Diss. ETH No. 24709

Formal Foundations for Access and Inference Control in Databases

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

Marco Guarnieri

Laurea Magistrale in Ingegneria Informatica,
Università degli Studi di Bergamo
born on 06.03.1988
citizen of Italy

accepted on the recommendation of

Prof. Dr. David Basin, examiner
Prof. Dr. Stephen Chong, co-examiner
Prof. Dr. Heiko Mantel, co-examiner

2017

Abstract

Databases often store and manage sensitive data. Regulating the access to databases is, therefore, essential. To this end, database security researchers have developed both *access control* and *inference control* mechanisms. The former limit direct access to sensitive data, whereas the latter prevent leaks caused by combining query results with external information such as prior knowledge or data dependencies. Ideally, all these mechanisms should come with security proofs clearly stating what attacks they are designed to thwart, as with security mechanisms in other domains. Unfortunately, this is far from reality. Existing protection mechanisms are implemented in an ad hoc fashion, with neither precise security guarantees nor the means to verify them. This has immediate consequences as existing mechanisms are inadequate to secure modern databases and are susceptible to attacks.

In this thesis, we develop theoretical foundations for access and inference control in databases. We leverage these foundations to design *provably secure* and *practical* protection mechanisms for modern database systems. In more detail, our thesis makes the following contributions.

First, we formalize Security-Aware Query Processing, the task of computing answers to `SELECT` queries in the presence of access control policies, and we study the existence of optimal Security-Aware Query Processing algorithms, i.e., those algorithms that provide both confidentiality and maximal data availability. We prove that there are no optimal algorithms for the relational calculus, whereas optimal algorithms exist for some of its fragments. We also investigate the connections between two different semantics for database access control that have been previously presented in the literature. In this respect, we show that, for optimal algorithms, these semantics are distinct for non-boolean `SELECT` queries, but they coincide for boolean queries.

Second, we develop foundations for database inference control in the presence of probabilistic data dependencies. Our foundations are based on ProbLog, a state-of-the-art probabilistic logic programming language. We leverage these foundations to develop Angerona, a database inference control mechanism that secures databases against attackers exploiting probabilistic dependencies. Angerona provides precise security, completeness, and tractability guarantees by exploiting a tractable inference algorithm for a practically relevant fragment of ProbLog. We empirically evaluate Angerona's performance and show that it scales to relevant security-critical problems.

Third, we develop a formal framework for reasoning about the security of database access control mechanisms. Our framework consists of (1) a formal operational semantics of databases supporting advanced features, such as triggers and views, (2) an attacker model formalizing how attackers infer information from the database system's behavior, and (3) precise security conditions accounting for dynamic security policies. Our attacker model and security conditions subsume the standard, limited attacker models and conditions considered in previous works, where both the database state and the policy are fixed and attackers are restricted to `SELECT` queries. Guided by our operational semantics and attacker model, we develop a provably secure enforcement mechanism that thwarts confidentiality and integrity attacks that existing mechanisms fail to prevent.

Finally, we reconcile database access control and information-flow control, two seemingly disparate research areas that share the same goal: protecting the confidentiality and integrity of sensitive information. We develop WhileSql, an imperative language with querying capabilities that relies on our database operational semantics, and we provide a framework for reasoning about the security of WhileSql programs. We reduce database access control to determining whether WhileSql programs leak information, thereby providing a way of applying existing information-flow techniques to database access control. We also develop a provably secure enforcement mechanism for WhileSql programs, which secures programs that existing mechanisms fail to secure. Our mechanism combines dynamic information-flow tracking with concepts from database theory, such as disclosure lattices and query determinacy, to provide end-to-end security guarantees for programs interacting with modern databases.

Riassunto

I moderni sistemi informativi spesso processano informazioni sensibili, che vengono memorizzate all'interno di basi di dati (*database*). Prevenire possibili attacchi aventi come obiettivo tali informazioni richiede quindi di controllare e limitare l'accesso alle basi di dati. Per questo motivo, sono stati proposti e implementati vari meccanismi che permettono il controllo degli accessi (*access control*) e delle inferenze (*inference control*). I primi limitano l'accesso diretto alle informazioni sensibili. I secondi, invece, regolano l'accesso indiretto a tali informazioni, essi cioè limitano le informazioni che possono essere ottenute combinando i risultati estratti dalle basi di dati tramite interrogazioni con informazioni aggiuntive come, per esempio, informazioni pregresse or dipendenze tra i dati. Idealmente, tutti questi meccanismi dovrebbero essere accompagnati da dimostrazioni formali che specificano e certificano quali attacchi vengono bloccati. Sfortunatamente, la realtà è diversa. I meccanismi di sicurezza esistenti sono stati sviluppati senza precise garanzie di sicurezza. Questo ha conseguenze immediate: i meccanismi esistenti non sono adeguati per proteggere le moderne basi di dati e sono vulnerabili a vari attacchi.

In questa tesi presentiamo i fondamenti teorici per il controllo degli accessi e delle inferenze. Basandoci su questi fondamenti teorici, abbiamo progettato e implementato nuovi meccanismi per controllare gli accessi e le inferenze all'interno delle moderne basi di dati. Questi meccanismi forniscono precise garanzie di sicurezza (certificate da dimostrazioni formali) e sono al contempo utilizzabili in pratica.

Il nostro primo contributo studia come eseguire interrogazioni di tipo `SELECT` in presenza di politiche di controllo degli accessi. Tale problema è anche chiamato Security-Aware Query Processing. In particolare, studiamo l'esistenza di algoritmi ottimali per effettuare Security-Aware Query Processing, cioè quegli algoritmi che garantiscono allo stesso tempo confidenzialità e disponibilità dei dati. In questa tesi, dimostriamo che non ci sono algoritmi ottimali in caso si utilizzi il calcolo relazionale come linguaggio di interrogazione, mentre tali algoritmi esistono per vari frammenti del calcolo relazionale. In aggiunta, investighiamo anche le relazioni tra due differenti semantiche per il controllo degli accessi nelle basi di dati. A questo proposito, dimostriamo che, per gli algoritmi ottimali, queste due semantiche sono distinte per le interrogazioni non booleane, mentre coincidono per le interrogazioni booleane.

Il nostro secondo contributo, invece, analizza il controllo delle inferenze in presenza di dipendenze probabilistiche tra i dati memorizzati nella base di dati. A questo proposito, sviluppiamo i fondamenti teorici per il controllo delle inferenze utilizzando ProbLog, un linguaggio di programmazione logica con aspetti probabilistici. Basandoci su tali fondamenti, sviluppiamo Angerona, un meccanismo per il controllo delle inferenze che protegge le basi di dati da attaccanti aventi accesso alle dipendenze probabilistiche tra i dati stessi. Angerona fornisce precise garanzie di sicurezza, completezza, e trattabilitá utilizzando un nuovo algoritmo per effettuare inferenze probabilistiche per un sottoinsieme praticamente rilevante dei programmi esprimibili in ProbLog. Infine, valutiamo empiricamente le prestazioni di Angerona e mostriamo che scalano a vari problemi d'interesse.

Il nostro terzo contributo consiste nello sviluppare i fondamenti teorici per il controllo degli accessi nelle basi di dati. A questo proposito, presentiamo (1) una semantica operazionale delle basi di dati che supporta funzionalità avanzate come viste e trigger, (2) un modello dell'attaccante che formalizza come un attaccante estrae informazioni sul contenuto della base di dati osservando il comportamento del sistema, e (3) precise definizioni di sicurezza che tengono conto di politiche del controllo degli accessi dinamiche. Sia il nostro modello dell'attaccante che le nostre definizioni di sicurezza sono più generali di quelle considerate in lavori precedenti, dove sia il contenuto della base di dati che la politica di controllo degli accessi sono fissi e gli attaccanti possono utilizzare solo interrogazioni di tipo `SELECT`. Utilizzando questi fondamenti teorici, infine, sviluppiamo un nuovo meccanismo per il controllo degli accessi che previene una serie di attacchi all'integrità e confidenzialità dei dati che non sono bloccati dai meccanismi esistenti.

L'ultimo nostro contributo consiste nel riconciliare il controllo degli accessi all'interno delle basi di dati con il controllo del flusso delle informazioni (*information-flow control*) all'interno dei programmi. Queste due aree di ricerca sembrano distinte, ma hanno lo stesso obiettivo: proteggere la confidenzialità e l'integrità di informazioni sensibili. In questa tesi, sviluppiamo WhileSql, un linguaggio di programmazione imperativo che permette anche di interrogare delle basi di dati, insieme con adeguate definizioni di sicurezza per i programmi espressi utilizzando WhileSql. Riduciamo il

controllo degli accessi per le basi di dati al problema di determinare se un programma WhileSql divulga informazioni sensibili. Questo permette di applicare varie tecniche già esistenti per il controllo del flusso delle informazioni al problema del controllo degli accessi nelle basi di dati. In aggiunta, sviluppiamo un meccanismo per controllare il flusso delle informazioni in programmi WhileSql e dimostriamo la sua sicurezza. Il nostro meccanismo combina il tracciamento dinamico del flusso delle informazioni con concetti derivati dalla teoria delle basi di dati per fornire precise garanzie di sicurezza per programmi che interagiscono con le moderne basi di dati.

Acknowledgements

First and foremost, I wish to thank my advisor David Basin for his support, education, and guidance in developing this work. During my years at ETH, David has always provided prompt, precise, and insightful comments on all my ideas and writings, and he has always found time in his busy schedule to discuss the many issues associated with my research. David taught me a lot from a research perspective: how to formulate clean and abstract problem statements, how to effectively organize and develop my ideas, and how to concisely communicate them. Thanks to him, I am a better researcher and person. I also wish to thank him for his continued effort in reminding me that simplicity and conciseness (in thoughts, words, and notation) lay at the hearth of clean and ultimately beautiful research.

I wish to express my appreciation and gratitude to my co-examiners—Stephen Chong and Heiko Mantel—for taking time to read through this thesis, providing insightful comments and criticisms, and for allowing me to defend my work.

The work presented in this thesis owes a large debt of gratitude to my collaborators—Srdjan Marinovic, Daniel Schoepe, Musard Balliu, and Andrei Sabelfeld—who shared greatly in its creation. It was a pleasure working with them and I enjoyed our discussions on database security.

I wish to express a special thank to Srdjan Marinovic, who, in addition to being a terrific collaborator, provided many creative ideas as well as insightful comments. Despite his busy schedule, Srdjan has always been available for lengthy discussions about our research as well as the pitfalls of academic life. These discussions allowed me to evaluate my research from new perspectives and, ultimately, greatly improved this thesis' quality.

I wish to extend my heartfelt gratitude to all those who commented on drafts of my papers, chapters, and presentations from which this thesis is composed. In alphabetical order, these persons are: Carlos Cotrini, Mohammad Torabi Dashti, Jannik Dreier, Erwin Fang, Arthur Gervais, Nikolaos Karapanos, Andreas Lochbihler, Ognjen Maric, Esfaandiar Mohammadi, Hubert Ritzdorf, Lara Schmid, David Sommer, Christoph Sprenger, Dmitriy Traytel, Petar Tsankov, Thilo Weghorn, Der-Yeuan Yu, and Eugen Zalinescu.

Our research group's secretary Barbara Pfänder deserves a special praise for how she easily, efficiently, and effectively handled the many bureaucratic issues–both academic and governmental– that arose during my stay at ETH.

I wish to express a special thank to Arthur Gervais, Nikolaos Karapanos, David Sommer, and Der-Yeuan Yu, who shared the office with me during my stay at ETH and endured the mess on my desk (and other office's furnitures) for so long. I also wish to thank Carlos Cotrini, Petar Tsankov, and Thilo Weghorn for the many discussions on various aspects of access control. I wish to thank all the members of the Institute of Information Security, who made my stay at ETH an intellectually stimulating experience. Furthermore, I would like to thank all my friends at ETH for all the after-work outings and movies, all the fun discussions during tea and coffee breaks, lunches, and dinners, as well as all the late nights spent together trying to meet a deadline. All these moments made my stay at ETH memorable and extremely pleasant.

I wish to thank my parents Angela and Ernesto and my brother Stefano for their support throughout my entire life. What I am now I owe to them, and this thesis is as much my achievement as theirs. They also deserve my gratitude for hosting me during my (rather frequent) visits to Italy, which represented an often much needed break from the stress of academic life as well as one of the few ways of getting access to delicious and reasonably priced food.

Finally, I would like to thank my girlfriend Letizia. She supported and encouraged me during the many ups and downs of my PhD for 5 years. She patiently waited me for all this time, and she put a lot of effort into making our relationship work even across the Alps. Without her support, this thesis would hardly exist.

Contents

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1

Introduction

Our society is interconnected and data-driven. According to a recent study [75], the digital universe, i.e., the amount of data generated, processed, and replicated, in the year 2020 will be about 40 zettabytes, which is 10^{21} bytes (this is equivalent approximately to 5,200 gigabytes for every human being on the planet). These data range from seemingly non-sensitive information, like public posts on social networks, to extremely sensitive ones, such as financial transactions or health-related information.

Protecting and regulating the access to this growing collection of data is one of the main challenges faced by information systems. On the one hand, many companies consider the data they process and collect as a critical asset. They are therefore interested in protecting data and preventing undesired leaks of sensitive information. On the other hand, these data can be used to infer sensitive information about individuals, and our society is therefore growing increasingly aware of the associated security and privacy risks. A recent survey [156] shows that security is one of the main concerns of European citizens. In particular, 88% of the interviewees say that data security is an important factor when choosing a company, and about 50% of them are willing to pay for better data protection. This increased interest in security is reflected, for example, by the recent European data protection regulation [130], which requires IT companies to properly secure customer data. For instance, "Personal data should be processed in a manner that ensures appropriate security and confidentiality of the personal data, including for preventing unauthorised access to or use of personal data and the equipment used for the processing" [130]. Satisfying such a requirement necessitates protection mechanisms that provide precise security guarantees and are secure against attacks.

Databases are one of the main components of information systems. Regulating the access to databases is, therefore, of utmost importance to prevent leaks of sensitive information. To this end, researchers have developed protection mechanisms offering *access control* and *inference control* capabilities. Existing mechanisms, however, are inadequate to secure modern systems. They are developed in ad hoc fashion, ignore many security-relevant database features, and do not provide precise security guarantees. It is, therefore, unclear what classes of attacks they prevent and in which sense they secure the database system. This has immediate consequences: attackers can exploit advanced database features to subvert protection mechanisms implemented in commercial databases and infer sensitive information. These mechanisms, therefore, cannot be used to fulfill the strict requirements, such as ensuring data confidentiality, imposed by recent regulations [130].

To address all these limitations, we propose the use of *provably secure protection mechanisms*. These mechanisms come with precise security guarantees and proofs specifying which classes of attacks they are designed to thwart. With provably secure mechanisms, securing a system amounts to (1) identifying the precise attacker model and the desired security guarantees, and (2) deploying and configuring the mechanisms that provide these guarantees. The accompanying proofs then certify that the desired guarantees are met and that the system is protected against specific classes of attacks.

In this thesis, we design provably secure mechanisms for access control and inference control. We first develop formal foundations for access and inference control in databases. We then leverage these foundations to construct *provably secure* and *practical* protection mechanisms for modern databases. The protection mechanisms we present in this thesis provide precise security guarantees, support advanced database features, and prevent attacks that existing mechanisms fail to stop. Observe that, in the context of this thesis, "provably secure" means that a protection mechanism's specification provably provides the desired security guarantees. However, we do not cover the verification of protection mechanisms' implementations, i.e., whether the implementation actually complies with the mechanism's specification.

In the following, we first introduce access control and inference control for database systems. Afterwards, we illustrate the limitations of existing approaches. We then present our contributions and, finally, we discuss this thesis' organization.

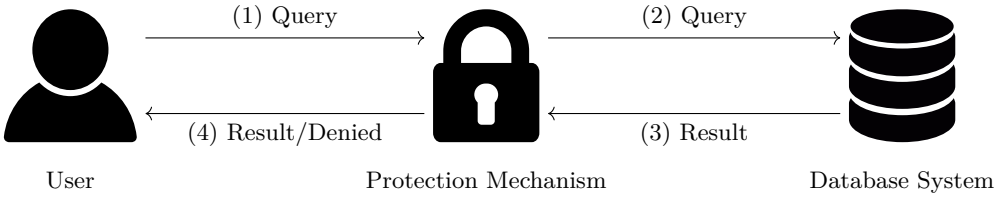


FIGURE 1.1: System model for DBAC and DBIC.

1.1 Regulating Access to Databases

It is essential to control the access to databases that store sensitive information. To this end, researchers have developed both database access control (DBAC) and database inference control (DBIC) mechanisms. We now summarize the key features of these approaches.

System Model. DBAC and DBIC share a common system model, depicted in Figure 1.1. Users interact with two components: a database system, which stores and manages the data, and a protection mechanism (be it a DBAC or DBIC mechanism), which intercepts the commands issued by the users and determines whether these commands are *secure*. The protection mechanism is parametrized by a security policy, which is designed by the security engineers and specifies which operations are secure. For simplicity, we assume that all communication between users and the components and between the components themselves is over secure channels.

When a user issues a command, the protection mechanism first checks whether the command is secure. If this is the case, it forwards the command to the database system. The database, then, executes the command and returns the command's result to the mechanism, which forwards it to the user. If instead the command is not secure, the protection mechanism blocks the command's execution, often returning an error message to the user.

Database Access Control. The main goal of DBAC is restricting the operations that each user can perform on the database. In general, a DBAC mechanism restricts both *read* and *write* access to the database. The former happens when a user acquires information about a portion of the database's content, for instance using a `SELECT` query. The latter instead happens when a user modifies the database content, e.g., using `INSERT` and `DELETE` commands, or the database configuration, for instance by modifying the policy or the database schema.

A DBAC mechanism is parametrized by a *security policy*, which expresses, often using a formal language, the operations each user is authorized to perform on the database. In the case of commercial databases, the policy is formalized using SQL, and it specifies (1) the database's content each user can *read*, and (2) the *write* operations each user can execute. A DBAC mechanism then intercepts all commands issued by the users and blocks all those commands that are not authorized, i.e., that do not comply with the given policy. Observe that in terms of read access, DBAC mechanisms mostly focus on restricting *direct access* [72]. Direct access to information stored in a database happens whenever a user issues a query, the database answers the query, and the user finally observes this query's result.

Database Inference Control. In contrast to DBAC, inference control restricts *indirect read access* to the database [72]. Indirect access happens whenever users infer sensitive information by combining the observed query results with external information, such as data dependencies or prior knowledge. Note that DBIC mechanisms usually ignore write access to the database.

In this setting, a DBIC mechanism is usually parametrized by (1) a security policy specifying the sensitive information, and (2) a description of the source of external information. Similarly to DBAC mechanisms, an inference control mechanism intercepts all user queries and authorizes only those that do not violate the security policy. DBIC has attracted considerable attention in recent years, and current research considers different sources of external information, such as the database schema [45, 91, 95, 128, 151, 152], statistical information [11, 47, 62, 65, 67], error messages [101], user-defined functions [101], and data dependencies [36, 40, 120, 121, 157, 158, 168].

Securing databases. Protecting the confidentiality of the data stored in databases requires protection from both *direct* and *indirect* access. As a result, it is essential to deploy both DBAC and DBIC protection mechanisms. Observe that the distinction between direct and indirect access is not always clear-cut. For instance, a leak through an exception caused by an `INSERT` command could be interpreted as direct access, as a user may learn information by directly observing the exception's result. However, it could also be classified as an indirect access since to infer information from the message a user would have to know which integrity constraint has been violated. Hence, some DBAC mechanisms may also include inference control aspects.

1.2 Provable Security for Databases – Challenges and Gaps

To secure a database system, security engineers first have to identify the attacker model and the desired security guarantees. Afterwards, they have to deploy some enforcement mechanisms that can provide these guarantees. Finally, they have to properly configure the deployed mechanisms to ensure that the guarantees are met.

Securing database systems is difficult. The aforementioned process may fail at any step, leading to insecure and vulnerable systems. For example, overseeing some aspects of the security analysis may lead to underestimating the attacker's capabilities and settling for security guarantees that are weaker than those needed in practice. Similarly, a misconfigured enforcement mechanism may grant too many permissions to users, thereby giving them access to sensitive information. These kinds of failures are well understood and researchers have been studying how to prevent them for decades. In particular, methodologies [119, 147, 159] exist to help security engineers in eliciting the attacker model and the security requirements, thereby reducing the likelihood of mistakes during the first step. Similarly, access control researchers have been developing techniques and tools [74, 100] for verifying the correctness of a given security policy with respect to a set of requirements, making it easier to identify misconfigured systems.

A third, subtler, way in which the above process may fail is in case the enforcement mechanisms do not offer the desired guarantees. This is particularly critical since security engineers often just re-use existing enforcement mechanisms; they do not design them. Hence, a mechanism that fails in delivering the desired guarantees would affect the security of all systems relying on it. Despite this, developing provably secure enforcement mechanisms is extremely hard, and even enforcement mechanisms implemented in commercial database systems fail in providing basic security guarantees, such as confidentiality. This is due to a lack of adequate and solid foundations for access and inference control in databases. In particular, existing formal models for DBAC and DBIC fail in meeting the following requirements, which are essential for reasoning about security of modern databases.

- **Realistic attacker models.** An attacker model specifies an attacker's capabilities, how he interacts with the database system, how he infers information from the system's behavior, and how he reasons about the acquired information. Reasoning about the security of an enforcement mechanism, therefore, requires an attacker model. In particular, the attacker model should be precise and formal to allow for security proofs, and it should capture the attacker's capabilities. However, existing DBAC and DBIC mechanisms, such as [13, 40, 131, 165], do not explicitly formalize their attacker models, thereby making it more difficult to understand the provided security guarantees. Attacker models should be first-class citizens in database security. The relevant models must be made explicit, just like when analyzing security in other domains. Solid foundations for DBAC and DBIC, therefore, must come with a family of *precise* and *realistic* attacker models.

- **Support for advanced, security-critical, database features.** Modern database systems provide users with much more than just `SELECT` queries to retrieve information. Indeed, users can modify the database content using `INSERT` and `DELETE` commands. They can also modify the database configuration or the security policy, for instance using `GRANT` commands to delegate permissions. Additionally, databases support advanced features like views, which allows the creation of virtual tables, or triggers, which automatically execute operations in response to user commands. All these features can be exploited by attackers to leak sensitive information. Reasoning about the security of modern database systems, therefore, requires formal models that account for all these advanced features. Unfortunately, existing formal models [131, 165] mostly consider only `SELECT` commands, thereby ignoring more advanced features.

- **Adequate security conditions.** Reasoning about the security of enforcement mechanisms requires one to first understand and formalize what it means to be *secure with respect to a given attacker model*. Existing research [131, 165] propose security conditions for `SELECT` queries based on well-known database theory problems. These conditions are, however, not adequate for a realistic setting involving modern databases, as they ignore attackers that interact with the database using more than just `SELECT` queries. As a result, existing conditions do not consider leaks involving features like `INSERT` and `DELETE` commands or triggers. They also ignore changes to the database and the security policy. All these features must be accounted for in any reasonable formal definition of security for modern databases.

Challenges

We identify four key challenges for DBAC and DBIC:

- **Developing foundations for DBAC and DBIC.** As explained above, existing formal models for DBAC and DBIC are not adequate for reasoning about the security of modern databases. Developing provably secure enforcement mechanisms, however, requires solid foundations. In more detail, we need (1) a formal operational semantics of database systems that accounts for advanced security-critical features, (2) precise attacker models capturing how malicious users may interact with the database, and (3) adequate security conditions.

- **Theoretical limitations.** A thorough understanding of DBAC and DBIC requires studying the theoretical limitations of these two tasks. This is important to drive the practical development of enforcement mechanisms. For instance, answering questions like *"What can and cannot be done in terms of DBAC and DBIC?"* or *"What is the trade-off between security and data availability?"* could clarify our understanding of these tasks as well as shed new light on settings where provably secure algorithms can be used efficiently without losing data availability.

- **Provably-secure enforcement mechanisms.** As mentioned before, existing protection mechanisms for databases fail in securing database systems. Attackers can infer sensitive information and modify the database's content in unauthorized ways. An open problem for DBAC and DBIC is, therefore, developing novel enforcement mechanisms that are tractable and provide precise security guarantees. Observe that tractability is a critical feature for an interactive setting like databases, where hundreds or thousands of queries per second are processed at runtime. Providing precise guarantees is essential to ensure that the developed mechanisms work as expected and thwart attacks. In this respect, enforcement mechanisms should be accompanied with precise attacker models and proofs certifying what security guarantees are met.

- **Security beyond the database.** In addition to databases, information systems consist of applications that process the data stored in databases. To prevent leaks, one also has to reason about how these applications process the data and how they interact with the databases. To this end, security researchers have developed information-flow control (IFC) techniques which track how the data flows inside programs and detect possible leaks of sensitive information. Existing IFC techniques, such as [141, 167], however, have severe limitations when it comes to dealing with databases. In particular, they consider only simplistic database models that ignore advanced security-critical features. They also ignore advances in DBAC and DBIC. Effectively securing modern information systems, however, requires combining information-flow control with DBAC and DBIC.

1.3 Contributions

This thesis makes the following contributions.

Limitations of Database Access Control. We characterize the limitations of database access control. In this respect, we formalize *Security-Aware Query Processing*, the task of computing answers to `SELECT` queries in the presence of access control policies, and we investigate the existence of optimal Security-Aware Query Processing algorithms, i.e., those algorithms that provide both confidentiality and maximal data availability. We present general impossibility results for the existence of optimal algorithms for Security-Aware Query Processing and classify query languages for which such algorithms exist. In particular, we show that for the relational calculus there are no optimal algorithms, whereas optimal algorithms exist for some of its fragments, such as the existential fragment. These results clarify the trade-offs between security and data availability and identify fragments of the relational calculus where it is possible to secure databases without reducing data availability due to false positives, i.e., without rejecting as insecure queries that are actually secure.

We also establish relationships between two different semantics for database access control, called Truman and Non-Truman models. These two models have different, and seemingly contradictory, goals, and their relationship was previously unclear. In the past, they were sometimes considered as distinct [131, 165]. For optimal Security-Aware Query Processing, we show that the Non-Truman model is a special case of the Truman model for boolean queries in the relational calculus. Moreover, the two models coincide for sufficiently powerful query languages, such as the relational calculus with aggregation operators. In contrast, for non-boolean queries, we prove that the two models are distinct.

Protecting databases from probabilistic inference. We study DBIC in the presence of probabilistic data dependencies, such as those arising in genomics [97, 105, 109], social networks [92],

and location tracking [117]. Attackers can exploit these dependencies to infer sensitive information with high confidence. Existing DBIC mechanisms, however, are inadequate to secure database systems in this settings. They either support very limited classes of probabilistic dependencies [44, 45, 91, 102, 120, 121, 166] or rely on intractable enforcement algorithms [116].

To address this, we develop foundations for DBIC in the presence of probabilistic data dependencies. We then use these foundations to build a tractable and practically useful DBIC mechanism based on probabilistic logic programming.

- We develop AtkLog, a language for formalizing attackers' beliefs and how they evolve while interacting with the system. AtkLog builds on ProbLog [60, 61, 73], a state-of-the-art probabilistic extension of Datalog, and extends its semantics by building on three key ideas from [52, 103, 116]: (1) users' beliefs can be represented as probability distributions, (2) belief revision can be performed by conditioning the probability distribution based on the users' observations, and (3) rejecting queries as insecure may leak information. By combining Datalog with probabilistic models and belief revision based on users' knowledge, AtkLog provides a natural and expressive language to model users' beliefs and thereby serves as a foundation for DBIC in the presence of probabilistic inferences.

- We identify acyclic ProbLog programs, a class of programs where the data complexity of probabilistic inference is PTime. We precisely characterize this class and develop a dedicated inference engine. Since ProbLog's inference is intractable in general, we see acyclic programs as an essential building block to effectively use AtkLog for DBIC.

- We present Angerona, a novel DBIC mechanism that secures databases against probabilistic inferences. We prove that Angerona is secure with respect to any AtkLog-attacker. In contrast to existing mechanisms, Angerona provides precise tractability and completeness guarantees for a practically relevant class of attackers. We empirically show that Angerona scales to relevant problems of interest.

Protecting modern database systems. We study the problem of protecting modern database systems that support advanced features like triggers and dynamic policies. We show that popular database systems are susceptible to two types of attacks. Integrity attacks allow an attacker to perform non-authorized changes to the database. Confidentiality attacks instead allow an attacker to learn sensitive data. These attacks exploit advanced SQL features, such as triggers, views, and integrity constraints, and they are easy to carry out.

Motivated by these attacks, we develop a comprehensive formal framework for the design and analysis of database access control. Our framework consists of a formal operational semantics of database systems and an attacker model formalizing how attackers infer information from the system's behavior, complemented with adequate security conditions. We use this framework to design and verify an access control mechanism that prevents confidentiality and integrity attacks that existing mechanisms fail to stop.

- Our operational semantics supports SQL's core features, as well as triggers, views, and integrity constraints. It models both the security-critical aspects of these features and the database's dynamic behavior at the level needed to capture realistic attacks. Our semantics is substantially more detailed than those used in previous works [131,165], which ignore the database's dynamics and advanced database features.

- In addition to SQL's core features, our attacker model incorporates advanced features such as triggers, views, and integrity constraints. It also accounts for how an attacker can infer information based on the semantics of these features. Our attacker model subsumes those considered in previous works [131, 165].

- Our security definitions—*database integrity* and *data confidentiality*—reflect two important security requirements for database access control. There is a natural and intuitive relationship between these definitions and the types of attacks that we identify. We thus argue that these definitions provide a strong measure of whether a given access control mechanism prevents attackers from exploiting modern SQL databases.

Guided by our operational semantics and attacker model, we build a database access control mechanism that is provably secure with respect to our attacker model and security definitions. In contrast to existing mechanisms, our solution prevents all the attacks that we identified.

Reconciling Database Access Control and Information-flow Control. Database access control and information-flow control (IFC) have the same objective: to protect the confidentiality and integrity of sensitive information. However, they have different foundations, security notions, and

enforcement mechanisms. We develop common foundations for IFC and DBAC. We leverage these foundations to (1) integrate advanced DBAC concepts into IFC, and (2) provide ways to re-use IFC techniques as building blocks for developing provably secure DBAC mechanisms.

- We develop common foundations for IFC and DBAC using WhileSql, a simple imperative language extended with querying capabilities. WhileSql is built on top of our database operational semantics and supports advanced database features like triggers and views. We propose two security conditions, one capturing *internal attackers* that directly interact with the database and the other capturing *external attackers* that interact with the database through an application. We use these conditions to clarify the differences between IFC and DBAC.

- We bridge IFC and DBAC. In particular, we reduce the latter to checking whether WhileSql programs leak information. We prove that sound IFC mechanisms for internal attackers can be used to construct provably secure DBAC mechanisms. Our reduction enables the direct application of existing IFC techniques to DBAC.

- We develop a novel dynamic security monitor for WhileSql programs and prove it sound with respect to our security condition for external attackers. Our monitor combines information-flow tracking with advanced DBAC concepts such as disclosure lattices [26] and query determinacy [124]. As a result, it can track fine-grained dependencies at the tuple-level. Additionally, our monitor supports advanced database features, such as triggers and policy changes, which are not supported by existing mechanisms [57, 59, 111, 141, 143, 167].

1.4 Publications

Much of the work presented in this thesis is based on the following co-authored articles:

- Marco Guarnieri and David Basin, *"Optimal Security-Aware Query Processing"*, in Proceedings of the 40th International Conference on Very Large Data Bases (**VLDB 2014**)

- Marco Guarnieri, Srdjan Marinovic, and David Basin, *"Strong and Provably Secure Database Access Control"*, in Proceedings of the 1st IEEE European Symposium on Security and Privacy (**EuroS&P 2016**)

- Marco Guarnieri, Srdjan Marinovic, and David Basin, *"Securing Databases from Probabilistic Inferences"*, in Proceedings of 30th IEEE Computer Security Foundations Symposium (**CSF 2017**)

- Marco Guarnieri, Daniel Schoepe, Musard Balliu, David Basin, and Andrei Sabelfeld, *"Reconciling Database Access Control and Information-flow Control"* (technical report)

1.5 Organization and Structure

The thesis is organized in 5 parts, 8 chapters, and 4 appendices.

Part *I* consists of this introduction and Chapter 2, which presents background about databases, query languages, logic programming, and Bayesian Networks. Chapter 2 also introduces relevant problems from database theory, such as query containment and determinacy, and disclosure lattices.

Part *II* focuses on protecting databases against attackers that can execute only `SELECT`-queries, i.e., those attackers that can just retrieve information from a database, but cannot modify the database's content or configuration.

- Chapter 3 covers DBAC in the presence of `SELECT`-only attackers, i.e., those that can only execute `SELECT` queries. The chapter introduces Security-Aware Query Processing and investigates the existence of optimal Security-Aware Query Processing algorithms. It also investigates the connections between the Truman model and Non-Truman model semantics.

- Chapter 4 targets DBIC for `SELECT`-only attackers that, additionally, have access to the probabilistic dependencies among the database's data. It introduces AtkLog as well as a dedicated inference engine for a fragment of ProbLog. It also presents Angerona, a tractable and provably secure DBIC mechanism.

Part *III* focuses on securing databases against more powerful attacker models, which can exploit the advanced features provided by modern databases.

- Chapter 5 introduces our operational semantics for database systems supporting advanced features like triggers and views.

- Chapter 6 focuses on DBAC in the presence of attackers that can exploit advanced database features. The chapter presents the formal foundations for DBAC, which consist of a formal attacker model complemented with adequate security conditions, together with a provably secure enforcement mechanism.
- Chapter 7 moves beyond DBAC by focusing on end-to-end security for applications interacting with databases. It presents WhileSql's syntax and semantics. Additionally, it presents a construction for building DBAC mechanisms using IFC solutions as building blocks, as well as a dynamic IFC monitor that exploits advanced DBAC concepts.

Part *IV*, which consists only of Chapter 8, draws our conclusions and discusses future work. Finally, Part *V* contains the proofs of all our results.

- Appendix A contains proofs of the results in Chapter 3.
- Appendix B contains proofs of the results in Chapter 4.
- Appendix C contains proofs of the results in Chapter 6.
- Appendix D contains proofs of the results in Chapter 7.

Chapter 2

Notation and Background

In this chapter, we introduce background about the main topics covered in the thesis. We first introduce some notation that we use throughout this thesis. Afterwards, we provide an overview of databases, query languages, logic programming, and Bayesian Networks.

Organization. We present notation in Section 2.1. We overview database theory concepts in Section 2.2. In more detail, we formalize database theory in Section 2.2.1 and we introduce the relational calculus query language in Section 2.2.2. Furthermore, we introduce relevant database theory decision problems in Section 2.2.3, and in Section 2.2.4 we present Disclosure Lattices, which we use in Chapter 7. In Section 2.3, we introduce logic programs and we present an introduction to Bayesian Networks in Section 2.4, both of which are relevant for our treatment of probabilistic logic programs in Chapter 4.

2.1 Notation

Here, we introduce some notation used throughout the thesis.

Sets. Given a set S, we denote by 2^S its power-set, i.e., the set of all subsets of S. As is standard, $x \in S$ denotes that x belongs to the set S and $S' \subseteq S$ denotes that S' is a subset of S. Furthermore, we write $x_1, \ldots, x_n \in S$ as a short-hand for $x_1 \in S, \ldots, x_n \in S$. Finally, we use standard notation for set comprehension. We write $\{x \in S \mid \phi\}$ to denote the set of all x in S that satisfies the predicate ϕ. Furthermore, given $x, y \in \mathbb{N}$ such that $x \leq y$, $\{x, \ldots, y\}$ denotes the set $\{z \in \mathbb{N} \mid x \leq z \wedge z \leq y\}$. Finally, given two real numbers x and y such that $x \leq y$, $[x, y]$ denotes the set $\{z \in \mathbb{R} \mid x \leq z \wedge z \leq y\}$.

Sequences. Let S be a set. We denote by S^k is the set of sequences over S of length $k \in \mathbb{N}$. Furthermore, $S^* = \bigcup_{k \in \mathbb{N}} S^k$ is the set of all finite sequences over S, S^ω is the set of all infinite sequences over S, and, finally, $S^\infty = S^* \cup S^\omega$ is the set of all finite and infinite sequences over S. Given a sequence s, we denote by $|s|$ the number of elements in it (note that $|s| = \infty$ if $s \in S^\omega$), by s^j, where $j \in \mathbb{N}$, its prefix of length j, and by $s(j)$ its j-th element (if it exists). We also denote by ϵ the empty sequence, by $s_1 \cdot s_2$, where $s_1 \in S^*$ and $s_2 \in S^\infty$, the concatenation of s_1 and s_2, and by $s_1 \preceq s_2$ that s_1 is a prefix of s_2.

Tuples. Let $n \in \mathbb{N}$ and $S_1, \ldots, S_n$ be sets. A *n-tuple with domains $S_1, \ldots, S_n$* is of the form $\langle v_1, \ldots, v_n \rangle$, where $v_i \in S_i$ for $1 \leq i \leq n$. We often overline an identifier to stress that it represents a tuple, e.g., we write $\bar{t}$ instead of t to stress that t represents a tuple.

Given an n-tuple $\bar{t}$, we denote by $\bar{t}(i)$, where $1 \leq i \leq n$, the i-th element in $\bar{t}$ and by $|\bar{t}|$ the number of elements in $\bar{t}$. Namely, $\langle v_1, \ldots, v_n \rangle(i) = v_i$ and $|\langle v_1, \ldots, v_n \rangle| = n$. Moreover, given a tuple $\bar{t}$ without repetitions and a value v in $\bar{t}$, we denote by $pos(\bar{t}, v)$ the position of v in $\bar{t}$, i.e., $pos(\bar{t}, v) = j$ iff $\bar{t}(j) = y$. Finally, given two tuples $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_m \rangle$, we denote by $\langle x_1, \ldots, x_n \rangle \circ \langle y_1, \ldots, y_m \rangle$ the tuple $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$.

The *cartesian product of $S_1, \ldots, S_n$* is the set $S_1 \times \ldots \times S_n = \{\langle v_1, \ldots, v_n \rangle \mid \bigwedge_{1 \leq i \leq n} v_i \in S_i\}$ containing all tuples with domains $S_1, \ldots, S_n$. Note that, given a set D, we denote by D^n the cartesian product of D by itself n times, e.g., D^3 is $D \times D \times D$.

Substitutions. Let A and B be two sets. A *substitution* from A to B is a (possibly partial) function $\theta : A \to B$. Given a tuple $\langle v_1, \ldots, v_n \rangle$ and a substitution θ, we denote by $\langle v_1, \ldots, v_n \rangle \theta$, denoted also by $\theta(\langle v_1, \ldots, v_n \rangle)$, the tuple $\langle v'_1, \ldots, v'_n \rangle$ obtained by applying θ, i.e., for each $1 \leq i \leq n$, $v'_i = \theta(v_i)$ if this is defined and $v'_i = v_i$ otherwise. Furthermore, given a n-tuple without repetitions $\langle a_1, \ldots, a_n \rangle \in A^n$ and an n-tuple $\langle b_1, \ldots, b_n \rangle \in B^n$, we denote by $[\langle a_1, \ldots, a_n \rangle \mapsto \langle b_1, \ldots, b_n \rangle]$ the substitution $A \to B$ assigning to each a_i the corresponding b_i. Finally, we denote by $\theta[a \mapsto b]$, where $a \in A$ and $b \in B$, the assignment θ' obtained as follows: $\theta'(a') = \theta(a')$ for all $a' \neq a$, and $\theta'(a) = b$.

Functions. Given a function $f : A \to B$, we denote by $dom(f)$ its domain and by $img(f)$ its image. Given two functions $f : A \to B$ and $g : B \to C$, we denote by $g \circ f$ their composition, namely the function $g \circ f : A \to C$ such that $g \circ f(x) = g(f(x))$. Finally, given two functions $f : A \to B$ and

$g : C \to D$ such that $dom(f) \cap dom(g) = \emptyset$, we denote by $f \uplus g$ the following function:

$$f \uplus g(x) = \begin{cases} f(x) & \text{if } x \in dom(f) \\ g(x) & \text{if } x \in dom(g) \end{cases}$$

We sometimes adopt a notation inspired by functional programming for defining functions via λ-terms. For instance, we may write $\lambda x \in \mathbb{N}.\ x+3$ for the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(x) = x+3$ for all $x \in \mathbb{N}$. Note that we often omit the typing information in case the latter is clear from the context, e.g., we write $\lambda x.\ x+3$ instead of $\lambda x \in \mathbb{N}.\ x+3$.

We also use standard (higher-order) functions from functional programming. Given a function $f : N \to M$ and a sequence $x_1 \cdot \ldots \cdot x_n \in N^*$ of elements in N, we denote by $map(f, x_1 \cdot \ldots \cdot x_n)$ the sequence $f(x_1) \cdot \ldots \cdot f(x_n) \in M^*$. Given a function $\oplus : (N \times M) \to O$, a sequence $x_1 \cdot \ldots \cdot x_n \in N^*$, and a sequence $y_1 \cdot \ldots \cdot y_n \in M^*$, $zipWith(\oplus, x_1 \cdot \ldots \cdot x_n, y_1 \cdot \ldots \cdot y_n)$ denotes the sequence $(x_1 \oplus y_1) \cdot \ldots \cdot (x_n \oplus y_n) \in O^*$.

Binary relations. Let A be a set. A binary relation over A is a set $R \subseteq A \times A$. Given a binary relation R, we denote by R^n the binary relation $\{\langle a, b\rangle \mid \exists c \in A.\ \langle a, c\rangle \in R^{n-1} \wedge \langle c, b\rangle \in R\}$. Observe that R^0 is the reflexive relation $\{\langle a, a\rangle \mid a \in A\}$ and R^1 is the relation R itself. Furthermore, we denote by R^* the reflexive and transitive closure of R, i.e., $R^* = \bigcup_{n \in \mathbb{N}} R^n$.

2.2 Databases

We now present background about databases, query languages, and database theory. We refer the reader to [10] for a detailed introduction to databases.

2.2.1 Modeling Databases

Here, we formalize databases. Instead of relying on definitions based on the relational model, such as [10], we present a formulation based on standard concepts from first-order logic (over finite structures). While both approaches are equivalent for the purposes of this thesis, our formulation allows us to simplify various aspects, such as many-sorted tuples, that are not interesting from a security point of view.

Let $\mathcal{R}$ be a countably infinite set representing identifiers of predicate symbols. A *database schema* D is a pair $\langle \Sigma, \mathbf{dom}\rangle$, where Σ is a first-order signature and **dom** is a fixed domain. For simplicity, we consider just a single domain. Extensions to the many-sorted case are straightforward [10]. The signature Σ consists of a set of *predicate symbols* $R \in \mathcal{R}$ (also called *relation schemas*), each symbol with arity $|R|$, and one constant symbol for each constant in **dom**. We interpret constants by themselves in the semantics.

A *database state* db of D, also called a state, is a finite Σ-structure with domain **dom** that interprets each predicate symbol R by a finite $|R|$-ary relation over **dom**, called *relation instance* or *table*. We denote by Ω_D the set of all states. Given a relation schema R in D, $db(R)$ denotes the set of tuples that belong to (the interpretation of) R in the state db. Observe that if the domain **dom** is a finite set, then the set of all possible states Ω_D is finite as well.

Given a database state db, a relation schema R, and a tuple $\bar{t} \in \mathbf{dom}^{|R|}$, we denote by $db[R \oplus \bar{t}]$ the database db' obtained by adding $\bar{t}$ to the relation defined by R, i.e., $db'(R) = db(R) \cup \{\bar{t}\}$ and $db'(R') = db(R')$ for all $R' \neq R$. Similarly, $db[R \ominus \bar{t}]$ denotes the database db' obtained by removing $\bar{t}$ from the relation defined by R, i.e., $db'(R) = db(R) \setminus \{\bar{t}\}$ and $db'(R') = db(R')$ for all $R' \neq R$.

2.2.2 Relational calculus

The *domain relational calculus*, or simply the *relational calculus* (RC), is a query language defined by Codd [55] built on top of function-free first-order logic (FOL). The relational calculus is, historically, one of the classical query languages studied in database theory, together with the relational algebra and languages based on logic programming. We first introduce the relational calculus' syntax and semantics. Afterwards, we formalize boolean and non-boolean relational calculus queries.

Syntax. We now formalize the relational calculus' syntax.

Definition 2.1. Let $D = \langle \Sigma, \mathbf{dom}\rangle$ be a database schema and *Vars* be a countably infinite set of variables. An *RC-formula over* D is inductively defined as follows:

- $R(x_1, \ldots, x_n)$ is an RC-formula over D, where R is a relation schema in D and $x_1, \ldots, x_n$ are variables in *Vars*.
- $x = y$ is an RC-formula over D, where x and y are variables in *Vars* or constants in **dom**.
- $Q\ x.\ \phi$ is an RC-formula over D, where ϕ is an RC-formula over D, $x \in Vars$, and $Q \in \{\exists, \forall\}$.
- $\neg\phi$ is an RC-formula over D, where ϕ is an RC-formula over D.

- ϕ *op* ψ is an *RC*-formula over D, where ϕ and ψ are *RC*-formulae over D and $op \in \{\wedge, \vee\}$. □

As is standard, $x \neq y$ stands for $\neg(x = y)$, $\phi \rightarrow \psi$ denotes $\neg\phi \vee \psi$, and $\phi \leftrightarrow \psi$ stands for $\phi \rightarrow \psi \wedge \psi \rightarrow \phi$. Furthermore, we support formulae of the form $\top$ and $\bot$ in the standard way. We also allow using constant symbols inside formulae of the form $R(x_1, \ldots, x_n)$ with the standard meaning. For instance, $R(1, 2, x)$ is equivalent to $\exists x_1, x_2.\ (R(x_1, x_2, x) \wedge x_1 = 1 \wedge x_2 = 2)$. Finally, given two tuples $\overline{x}$ and $\overline{y}$ of variables and constants such that $|\overline{x}| = |\overline{y}|$, we write $\overline{x} = \overline{y}$ to denote $\bigwedge_{1 \leq i \leq |\overline{x}|} \overline{x}(i) = \overline{y}(i)$ and $\overline{x} \neq \overline{y}$ to denote $\bigvee_{1 \leq i \leq |\overline{x}|} \overline{x}(i) \neq \overline{y}(i)$.

Let ϕ be a relational calculus formula over D. We denote by $\mathit{free}(\phi)$ the set of free variables in ϕ, and we say that ϕ is a *sentence* if $\mathit{free}(\phi) = \emptyset$. We write $\phi(\overline{x})$, where $\overline{x}$ is a tuple of variables without repetitions, to denote that the free variables in ϕ are exactly those in $\overline{x}$, i.e., $\mathit{free}(\phi) = \{\overline{x}(i) \mid 1 \leq i \leq |\overline{x}|\}$. Finally, given a tuple $\overline{t} = \langle t_1, \ldots, t_n \rangle \in (\mathit{Vars} \cup \mathbf{dom})^n$ and a relational calculus formula $\phi(\overline{x}, \overline{y})$ with free variables $\overline{x}$ and $\overline{y}$, where $\overline{x} = \langle x_1, \ldots, x_n \rangle$, we denote by $\phi[\overline{x} \mapsto \overline{t}]$ the formula obtained by replacing all the free occurrences of x_i with t_i for each $i \in \{1, \ldots, n\}$.

Semantics. The semantics of the relational calculus is similar to that of first-order logic.

Definition 2.2. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema, db be a D-database state in Ω_D, and ϕ be an *RC*-sentence. The satisfiability relation $\models$ is inductively defined as follows:

- $db \models R(v_1, \ldots, v_n)$, where R is a relation schema in D and $v_1, \ldots, v_n \in \mathbf{dom}$, holds iff $\langle v_1, \ldots, v_n \rangle \in db(R)$.
- $db \models k_1 = k_2$, where $k_1, k_2 \in \mathbf{dom}$, iff $k_1 = k_2$.
- $db \models \forall x.\, \phi$ iff $db \models \phi[x \mapsto v]$ for all $v \in \mathbf{dom}$.
- $db \models \exists x.\, \phi$ iff $db \models \phi[x \mapsto v]$ for some $v \in \mathbf{dom}$.
- $db \models \neg\phi$ iff $db \not\models \phi$.
- $db \models \phi \wedge \psi$ iff $db \models \phi$ and $db \models \psi$.
- $db \models \phi \vee \psi$ iff $db \models \phi$ or $db \models \psi$. □

Note that in the above definition we interpret the constant symbols by themselves. Similarly, we interpret the equality relation in the standard way. As a result, we do not need to explicitly keep track of the interpretation of constant symbols and of equality in the satisfiability relation, as it is usually done for first-order logic.

Queries. We are now ready to introduce boolean and non-boolean queries.

Definition 2.3. Let D be a database schema. A *non-boolean query over* D is of the form $\{\overline{x} \mid \phi\}$, where ϕ is a relational calculus sentence over D, $\overline{x}$ is a tuple of variables, $\mathit{free}(\phi) = \{\overline{x}(i) \mid 1 \leq i \leq |\overline{x}|\}$, and $\mathit{free}(\phi) \neq \emptyset$. A *boolean query over* D is a query $\{\mid \phi\}$, where ϕ is a relational calculus sentence. □

For simplicity, we only consider non-boolean queries $\{\overline{x} \mid \phi\}$ where the tuple $\overline{x}$ does not contain repeated variables. The extension to the general case is straightforward.

We now define the result of a query over a database state.

Definition 2.4. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema, $db \in \Omega_D$ be a database state, and q be a query over D. We denote the evaluation of q on the state db by $[q]^{db}$. For a boolean query $q = \{\mid \phi\}$, $[q]^{db} = \top$ if ϕ is satisfied on the state db, i.e., $db \models \phi$, and otherwise $[q]^{db} = \bot$. For a non-boolean query $q = \{\overline{x} \mid \phi(\overline{x})\}$, $[q]^{db}$ is the largest set $T \subseteq \mathbf{dom}^{|\overline{x}|}$ such that $[\{\mid \phi[\overline{x} \mapsto \overline{t}]\}]^{db} = \top$ for all $\overline{t} \in T$, i.e., $[q]^{db} = \{\overline{t} \in \mathbf{dom}^{|\overline{x}|} \mid db \models \phi[\overline{x} \mapsto \overline{t}]\}$. □

With a slight abuse of notation, we refer to boolean queries $\{\mid \phi\}$ using only the sentence ϕ. We therefore write $[\phi]^{db}$ instead of $[\{\mid \phi\}]^{db}$.

In the following, we denote by RC the set of all relational calculus queries and by RC_{bool} the set of all relational calculus boolean queries (i.e., all *RC*-sentences).

Integrity Constraints. An integrity constraint is an invariant that must be satisfied for a database state to be considered *valid*. Formally, an *integrity constraint over* D is a relational calculus sentence γ over D. Given a database state db, we say that db *satisfies the constraint* γ iff $[\gamma]^{db} = \top$. Given a set of constraints Γ, Ω_D^Γ denotes the set of all database states satisfying the constraints in Γ, i.e., $\Omega_D^\Gamma = \{db \in \Omega_D \mid \bigwedge_{\gamma \in \Gamma} [\gamma]^{db} = \top\}$. Note that if $\Gamma = \emptyset$ (or if all sentences in Γ are valid), then $\Omega_D^\Gamma = \Omega_D$, namely all database states comply with the integrity constraints.

Functional and inclusion dependencies are two well-known classes of integrity constraints [10]. The former capture primary-key constraints, whereas the latter capture foreign-key constraints. A *functional dependency* is a sentence of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\ ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}')$. An *inclusion dependency*, instead, is a sentence of the form $\forall \overline{x}, \overline{y}.\ (R(\overline{x}, \overline{y}) \rightarrow \exists \overline{z}.\ S(\overline{x}, \overline{z}))$.

Restrictions. As is standard, we consider only *domain independent* queries, which are queries that yield the same result on all possible underlying domains. Domain independent relational calculus is as expressive as relational algebra [10]. Although checking whether a query is domain independent is undecidable, there are various ways to obtain domain independent queries through syntactic

restrictions, such as safe-range queries and range-restricted queries [10]. Note that these syntactic approaches necessarily characterize proper subsets of the domain independent queries.

Assignments. An *assignment* is a substitution from the set of variables *Vars* to the domain **dom**. Given a relational calculus formula ϕ and an assignment ν, we say that ν *is well-formed for* ϕ iff ν is defined for all variables in $\mathit{free}(\phi)$. Furthermore, given a formula ϕ with free variables $\mathit{free}(\phi)$ and an assignment ν, we denote by $\phi\nu$ the formula ϕ' obtained by replacing, for each free variable $x \in \mathit{free}(\phi)$ such that $\nu(x)$ is defined, all the free occurrences of x with $\nu(x)$.

2.2.3 Relevant Database Theory Problems

We now review some relevant problems from database theory. These problems have a wide range of applications, such as query optimization and rewriting. They are relevant for this thesis as they are closely related to existing security conditions for DBAC and DBIC [131, 165].

Finite satisfiability and validity. Finite satisfiability and finite validity are the variants of the usual satisfiability and validity problems for first-order logic but restricted to finite structures. These problems have been studied in finite model theory [113]. Formally, given a database schema D, we say that a sentence ϕ is *finitely satisfiable* iff there is a database state $db \in \Omega_D$ such that $[\phi]^{db} = \top$. Similarly, we say that a sentence ϕ is *finitely valid* iff for all database states $db \in \Omega_D$, $[\phi]^{db} = \top$. Hence, the finite satisfiability (respectively validity) problem is the task of checking whether a given sentence is finitely satisfiable (respectively valid).

Query Containment. Query containment [10] is the task of determining, given two queries q and q', whether the result of q' is always contained in the result of q in any possible database state. Query containment is a fundamental problem in query evaluation and optimization. In particular, it can be used to check whether different rewritings of the same query are equivalent, and this can be done efficiently for a practically relevant fragment of the relational calculus (called Conjunctive Queries) [48]. Formally, given a database schema D and a set of integrity constraints Γ, the non-boolean query q contains the non-boolean query q' iff $[q']^{db} \subseteq [q]^{db}$ for all $db \in \Omega_D^\Gamma$.

Certain and Possible Answers. The certain answer problem is the task of determining, given a query q, a database state db, and a tuple $\overline{t}$, whether $\overline{t}$ belongs to q's results in all database states db' that agree with db on a certain criterion. The possible answer problem, instead, is its dual, i.e., it is the task of determining, given a query q, a database state db, and a tuple $\overline{t}$, whether there exists a database state db' that agrees with db on a certain criterion such that $\overline{t}$ belongs to q's results. These two problems have been thoroughly studied in the context of answering queries in the presence of incomplete information, for instance for incomplete databases [112] or for determining answers given a set of views [9].

We formalize here a general version of these problems inspired by [9]. Let D be a database schema, Γ be a set of integrity constraints over D, Q be a set of queries over D, $q = \{\overline{x} \mid \phi\}$ be a non-boolean query over D, $\overline{t} \in \mathbf{dom}^{|\overline{x}|}$ be a tuple, and $db \in \Omega_D^\Gamma$ be a database state. We say that $\overline{t}$ *is a certain answer for* q *given* db *and* Q iff for all database states db' in Ω_D^Γ, if $[q']^{db} = [q']^{db'}$ for all $q' \in Q$, then $\overline{t} \in [q]^{db}$. Similarly, we say that $\overline{t}$ *is a possible answer for* q *given* db *and* Q iff there exists a database state db' in Ω_D^Γ such that $[q']^{db} = [q']^{db'}$ for all $q' \in Q$ and $\overline{t} \in [q]^{db}$.

Query Determinacy. Query determinacy [124] is the task of determining, given two sets of queries Q and Q', if the results of the queries in Q are always *sufficient* to determine the result of the queries in Q'. This, therefore, means that the queries in Q contain more information than those in Q'. Note that query determinacy has many applications to query rewriting [124].

Let D be a database schema, Γ be a set of integrity constraints over D, and Q and Q' be sets of queries over D. We say that Q *determines* Q', written $D, \Gamma \vdash Q \twoheadrightarrow Q'$, iff for all database states db, db' in Ω_D^Γ, if $[q]^{db} = [q]^{db'}$ for all $q \in Q$, then $[q']^{db} = [q']^{db'}$ for all $q' \in Q'$. For instance, the set $Q = \{T(1), R(2)\}$ determines the query $T(1) \vee R(2)$.

Instance-based Determinacy. Instance-based determinacy is a variant of the determinacy problem restricted to a fixed database state. Namely, the instance-based determinacy problem [107] consists of checking whether, given a database state db, a set of queries Q, and a query q, the results of the queries in Q in the state db fully determine q's result in db.

Let D be a database schema, Γ be a set of integrity constraints over D, Q be a set of queries over D, q be a query over D, and db be a database state in Ω_D^Γ. We say that Q *determines* q *in the instance* db, written $D, \Gamma, db \vdash Q \twoheadrightarrow q$, iff for all database states db' in Ω_D^Γ, if $[q']^{db} = [q']^{db'}$ for all $q' \in Q$, then $[q]^{db} = [q]^{db'}$. Observe that if $D, \Gamma, db \vdash Q \twoheadrightarrow q$ holds for all $db \in \Omega_D^\Gamma$, then $D, \Gamma \vdash Q \twoheadrightarrow q$ holds as well. Observe also that instance-based determinacy can be easily extended to sets of queries. To illustrate, the set $Q = \{T(1)\}$ determines the query $T(1) \vee R(2)$ in any database state db where $T(1)$ is satisfied. Note, however, that $Q = \{T(1)\}$ does not determine the query $T(1) \vee R(2)$.

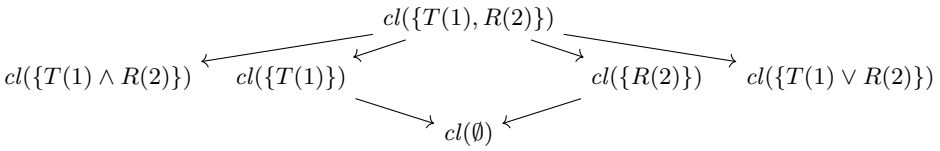


FIGURE 2.1: Portion of the disclosure lattice involving the queries $T(1) \wedge R(2)$, $T(1) \vee R(2)$, $T(1)$, and $R(2)$.

2.2.4 Disclosure Lattices

Disclosure lattices have been introduced by Bender et al. [26,27] to reason about the information disclosed by queries. Given two sets of queries Q_1 and Q_2, disclosure lattices provide a precise model for answering the following questions: (i) Does Q_1 reveal *more* information than Q_2? (ii) What *combined* information is revealed by Q_1 and Q_2? (iii) What *common* information is revealed by Q_1 alone and Q_2 alone?

Disclosure Orders. A *disclosure order* [26] is a binary relation $\preceq$ over 2^{RC} such that: (1) for all $Q, Q' \in 2^{RC}$, if $Q \subseteq Q'$, then $Q \preceq Q'$, (2) for all $Q, Q', Q'' \in 2^{RC}$, if $Q \preceq Q'$ and $Q' \preceq Q''$, then $Q \preceq Q''$, and (3) for all $Q, Q', Q'' \in 2^{RC}$, if $Q \preceq Q''$ and $Q' \preceq Q''$, then $Q \cup Q' \preceq Q''$.

Disclosure Lattices. Given a set of queries Q and a disclosure order $\preceq$, the *closure* of Q, written $cl(Q)$, is $\{q \in RC \mid \{q\} \preceq Q\}$. The $\preceq$-*disclosure lattice* [26] is a tuple $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ where (1) $\mathcal{L} = \{cl(Q) \mid Q \in 2^{RC}\}$, (2) $cl(Q) \sqsubseteq cl(Q')$ iff $Q \preceq Q'$, (3) $cl(Q) \sqcap cl(Q') = cl(Q) \cap cl(Q')$, (4) $cl(Q) \sqcup cl(Q') = cl(Q \cup Q')$, (5) $\bot = cl(\emptyset)$, and (6) $\top = cl(RC)$. Observe that $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ is a complete lattice [26].

Disclosure Lattices based on determinacy. Determinacy induces an ordering on the information content of queries. Hence, it is a good candidate for defining disclosure lattices. Bender et al. [26] present also other disclosure lattices, such as those based on query containment.

We are ready to define the relation $\preceq^{\rightarrow}_{D,\Gamma}$: given $Q, Q' \in 2^{RC}$, $Q \preceq^{\rightarrow}_{D,\Gamma} Q'$ iff $D, \Gamma \vdash Q' \twoheadrightarrow Q$. Observe that $Q \preceq^{\rightarrow}_{D,\Gamma} Q'$ means that Q is less informative than Q' (since the results of the queries in Q can be derived from the results of the queries in Q'). As shown in [26], $\preceq^{\rightarrow}_{D,\Gamma}$ is a disclosure order. As a result, the disclosure lattice defined by $\preceq^{\rightarrow}_{D,\Gamma}$ is a complete lattice [26].

To illustrate, consider the queries $T(1) \wedge R(2)$, $T(1) \vee R(2)$, $T(1)$, and $R(2)$. Figure 2.1 illustrates the portion of the lattice involving these queries, where arrows represent the relation $\sqsubseteq$. We can see that $cl(\{T(1)\}) \sqsubseteq cl(\{T(1), R(2)\})$, $cl(\{T(1)\}) \sqcup cl(\{R(2)\}) = cl(\{T(1), R(2)\})$, $cl(\{T(1)\}) \sqcap cl(\{R(2)\}) = \bot$. Observe also that $cl(\{T(1) \wedge R(2)\}) \sqsubseteq cl(\{T(1), R(2)\})$ and $cl(\{T(1) \vee R(2)\}) \sqsubseteq cl(\{T(1), R(2)\})$.

2.3 Logic Programming

We now introduce background on logic programs. In particular, we present DATALOG (extended with stratified negation), a declarative rule-based query language [10]. The key feature of DATALOG is that it supports the definition of recursive predicates.

Even though DATALOG is only a subset of more complex logic programming languages like PROLOG [43], with a slight abuse of notation in this thesis we use the term "logic programming" to refer to DATALOG with stratified negation. Note that DATALOG is relevant for our treatment of probabilistic logic programs in Chapter 4. Our treatment of logic programs follows [10]. We refer the reader to [10,43] for more details about DATALOG and PROLOG.

Syntax. Conventional DATALOG programs are constructed from atoms, literals, and rules. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema and *Vars* be a countably infinite set of variable identifiers. A $(\Sigma, \mathbf{dom})$-*atom* $R(v_1, \ldots, v_n)$ consists of a predicate symbol R in Σ and arguments $v_1, \ldots, v_n$ such that n is the arity of R and each v_i, for $1 \leq i \leq n$, is either a variable identifier in *Vars* or a constant in $\mathbf{dom}$. We denote by $\mathcal{A}_{\Sigma,\mathbf{dom}}$ the set $\{R(v_1, \ldots, v_{|R|}) \mid R \in \Sigma \wedge v_1, \ldots, v_{|R|} \in \mathbf{dom} \cup Vars\}$ of all $(\Sigma, \mathbf{dom})$-atoms. A $(\Sigma, \mathbf{dom})$-*literal* l is either a $(\Sigma, \mathbf{dom})$-atom a or its negation $\neg a$, where $a \in \mathcal{A}_{\Sigma,\mathbf{dom}}$. We denote by $\mathcal{L}_{\Sigma,\mathbf{dom}}$ the set $\mathcal{A}_{\Sigma,\mathbf{dom}} \cup \{\neg a \mid a \in \mathcal{A}_{\Sigma,\mathbf{dom}}\}$ of all $(\Sigma, \mathbf{dom})$-literals. Given a literal l, $vars(l)$ denotes the set of its variables, $args(l)$ the tuple containing its arguments, and $pred(l)$ the predicate symbol (i.e., the relation schema) used in l. As is standard, we say that a literal l is *positive* if it is an atom in $\mathcal{A}_{\Sigma,\mathbf{dom}}$ and *negative* if it is the negation of an atom. Furthermore, we say that a literal l is *ground* iff $vars(l) = \emptyset$. Finally, we denote by $atom(l)$ the atom used in l, i.e., given $a \in \mathcal{A}_{\Sigma,\mathbf{dom}}$, $atom(a) = a$ and $atom(\neg a) = a$.

A $(\Sigma, \mathbf{dom})$-*rule* is of the form $h \leftarrow l_1, \ldots, l_n, e_1, \ldots, e_m$, where $h \in \mathcal{A}_{\Sigma,\mathbf{dom}}$ is a $(\Sigma, \mathbf{dom})$-atom, $l_1, \ldots, l_n \in \mathcal{L}_{\Sigma,\mathbf{dom}}$ are $(\Sigma, \mathbf{dom})$-literals, and $e_1, \ldots, e_m$ are equality and inequality constraints

over the variables in $h, l_1, \ldots, l_m$.[1] Given a rule r, we denote by $head(r)$ the atom h, by $body(r)$ the literals $l_1, \ldots, l_n$, by $cstr(r)$ the constraints $e_1, \ldots, e_m$, and by $body(r, i)$ the i-th literal in r's body, i.e., $body(r, i) = l_i$. Furthermore, we denote by $body^+(r)$ (respectively $body^-(r)$) all positive (respectively negative) literals in $body(r)$. As is standard, we assume that the free variables in a rule's head are a subset of the free variables of the positive literals in the rule's body, i.e., $vars(head(r)) \subseteq \bigcup_{l \in body^+(r)} vars(l) \cup \bigcup_{(x=c) \in cstr(r) \wedge c \in \mathbf{dom}} \{x\}$. Finally, observe that a ground atom can be seen as a rule with an empty body.

We are now ready to define logic programs. Formally, a $(\Sigma, \mathbf{dom})$-*logic program* is a set of $(\Sigma, \mathbf{dom})$-ground atoms and $(\Sigma, \mathbf{dom})$-rules. Given a program p, we denote by $facts(p)$ the set of ground atoms in p and by $rules(p)$ all the rules that are not ground atoms, i.e., $rules(p) = p \setminus facts(p)$. Note that we ignore the distinction between extensional relation schemas (also called *edb*s), which are used only in ground atoms, and intensional relation schemas (also called *ibd*s), which are used in the head of the rules.

We consider only programs p that do not contain negative cycles in the rules as is standard for stratified Datalog with negation [10]. Namely, we consider only those programs p that admit a *stratification*. Following [10], we now formalize stratifications.

Definition 2.5. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema and p be a $(\Sigma, \mathbf{dom})$-logic program. A *stratification* of p is a sequence of $(\Sigma, \mathbf{dom})$-logic programs $p_1, \ldots, p_n$ such that for some mapping μ from the relation schemas in Σ to $\{0, \ldots, n\}$ the following conditions hold:

1. The programs $p_1, \ldots, p_n$ are a partition of $rules(p)$ (i.e., $\bigcup_{1 \leq i \leq n} p_i = p$ and $p_i \cap p_j = \emptyset$ for all distinct $i, j \in \{1, \ldots, n\}$).
2. For all rules $r \in rules(p)$ and all positive literals $l \in body^+(r)$, $\mu(pred(l)) \leq \mu(pred(head(r)))$.
3. For all rules $r \in rules(p)$ and all negative literals $l \in body^-(r)$, $\mu(pred(l)) < \mu(pred(head(r)))$.
4. For all rules $r \in rules(p)$, $r \in p_{\mu(pred(head(r)))}$ and $1 \leq \mu(pred(head(r))) \leq n$. □

A logic program p is *stratified* iff there is a stratification $p_1, \ldots, p_n$ for it, and we call each p_i a *stratum*.

Semantics. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema and r be a $(\Sigma, \mathbf{dom})$-rule. We denote by $ASGN(r)$ the set of all assignments from the free variables in r to values in $\mathbf{dom}$ that satisfy the equality and inequality constraints in r. Furthermore, given a mapping θ from variables in $Vars$ to values in $\mathbf{dom}$ and a literal l, we denote by $l\Theta$ the literal obtained by replacing all free variables in l with the corresponding values in θ.

One of the key concepts of the Datalog's semantics is the immediate consequence operator.

Definition 2.6. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema. The *immediate consequence operator* $\mathbb{T}$ takes as input a $(\Sigma, \mathbf{dom})$-logic program and returns another $(\Sigma, \mathbf{dom})$-logic program. The operator is defined as follows: $\mathbb{T}(p) = p \cup \{head(r)\theta \mid r \in p \wedge \theta \in ASGN(r) \wedge \forall l \in body^+(r).\ l\theta \in p \wedge \forall l \in body^-(r).\ atom(l)\theta \notin p\}$. □

The semantics of a Datalog program is the database state, which can be seen as a set of ground atoms, produced by executing the program. We first define the semantics of *semi-positive* programs [10], i.e., programs whose stratification contains just one stratum. The semantics of a semi-positive program is the least fixpoint of the immediate consequence operator. Note that such a fixpoint always exists [10]. Hence, the semantics is well-defined.

Definition 2.7. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema and p be a semi-positive $(\Sigma, \mathbf{dom})$-logic program. The semantics of p is as follows: $[\![p]\!] = facts\left(\bigcup_{i \in \mathbb{N}} \mathbb{T}^i(p)\right)$. □

We are now ready to introduce the semantics of stratified Datalog with negation.

Definition 2.8. Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema, p be a stratified $(\Sigma, \mathbf{dom})$-logic program with stratification $p_1, \ldots, p_n$, and $p_0 = facts(p)$. The semantics of p up to the i-th stratum is $[\![p]\!]_i = facts([\![p_i \cup [\![p]\!]_{i-1}]\!])$. The semantics of the program p is $[\![p]\!] = [\![p]\!]_n$. □

The stratified semantics simply executes the programs in the stratification sequentially, using the result of a stratum to derive additional facts for the next one. Note that the above semantics is well-defined since $p_i \cup [\![p]\!]_{i-1}$, for each $1 \leq i \leq n$, is a semi-positive program.

2.4 Bayesian Networks

A Bayesian Network (BN) is a graphical way of compactly representing a probability distribution by specifying a set of random variables and their dependencies as a directed acyclic graph. We now present an overview of BNs, which we use in Chapter 4. We refer the interested reader to [77, 105] for

[1] Without loss of generality, we assume that equality constraints involving a variable $x \in Vars$ and a constant $c \in \mathbf{dom}$ are of the form $x = c$.

additional information on BNs and other probabilistic graphical models. Our treatment of Bayesian Networks follows [105]. Here we focus only on BNs where each random variable has finitely many outcomes.

We first introduce probability distributions and random variables.

Definition 2.9. Let $V, V_1, \ldots, V_n$ be finite sets of possible outcomes. A *probability distribution over* V is a function $P : V \to [0, 1]$ such that $\Sigma_{v \in V} P(v) = 1$. A *conditional probability table over* V *indexed by* $V_1, \ldots, V_n$ is a function $CPT : V_1 \times \ldots \times V_n \to (V \to [0, 1])$ such that for all $\overline{v} \in V_1 \times \ldots \times V_n$, $CPT(\overline{v})$ is a probability distribution over V. A *random variable* X is a pair $\langle V, P \rangle$, where V is a finite domain and P is a probability distribution over V. □

Given a random variable $X = \langle V, P \rangle$, the probability that $v \in V$ is X's outcome, denoted $P(X = v)$, is $P(v)$.

Joint probability distributions specify the probabilities of multiple random variables.

Definition 2.10. Let $X_1 = \langle V_1, P_1 \rangle$, ..., $X_n = \langle V_n, P_n \rangle$ be random variables. A joint probability distribution over $X_1, \ldots, X_n$ is a probability distribution P over $V_1 \times \ldots \times V_n$ such that for each $1 \leq i \leq n$ and each $v_i \in V_i$, $\sum_{v_1 \in V_1} \cdots \sum_{v_{i-1} \in V_{i-1}} \sum_{v_{i+1} \in V_{i+1}} \cdots \sum_{v_n \in V_n} P(\langle v_1, \ldots, v_i - 1, v_i, v_{i+1}, \ldots v_n \rangle) = P_i(v_i)$. □

Let P be a joint probability distribution over $X_1 = \langle V_1, P_1 \rangle$, ..., $X_n = \langle V_n, P_n \rangle$. The probability that $\langle v_1, \ldots, v_n \rangle \in V_1 \times \ldots \times V_n$ is the outcome of $X_1, \ldots, X_n$, written $P(X_1 = v_1, \ldots, X_n = v_n)$, is $P(\langle v_1, \ldots, v_n \rangle)$. One can derive joint distributions over subsets of $X_1, \ldots, X_n$ by just summing out some of the variables. Let $\mathbb{X} = \{X_1, \ldots, X_n\}$ and $\mathbb{Z}_1 = \{X_1^1, \ldots, X_k^1\}$, $\mathbb{Z}_2 = \{X_1^2, \ldots, X_j^2\}$ be two sets defining a partition of $\mathbb{X}$, where each X_i^1 is of the form $\langle V_i^1, P_i^1 \rangle$ and each X_h^2 is of the form $\langle V_h^2, P_h^2 \rangle$ for $1 \leq i \leq k$ and $1 \leq h \leq j$. The probability distribution $\lambda v_1^1 \in V_1^1, \ldots, v_k^1 \in V_k^1 . \sum_{v_1^2 \in V_1^2} \cdots \sum_{v_j^2 \in V_j^2} P(X_1^1 = v_1^1, \ldots, X_k^1 = v_k^1, X_1^2 = v_1^2, \ldots, X_j^2 = v_j^2)$ is a joint probability distribution over $X_1^1, \ldots, X_k^1$. We denote the above probability distribution as $P_{X_1^1, \ldots, X_k^1}$.

A conditional probability distribution specifies the probability distribution of a random variable given a fixed outcome for other variables.

Definition 2.11. Let P be a joint probability distribution over $X_1 = \langle V_1, P_1 \rangle$, ..., $X_n = \langle V_n, P_n \rangle$. The *conditional probability of* X_i *given outcomes* $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ *for* $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n$, written $P(X_i \mid X_1 = v_1, \ldots, X_{i-1} = v_{i-1}, X_{i+1} = v_{i+1}, \ldots, X_n = v_n)$, is the function $\lambda v_i \in V_i . P(X_1 = v_1, \ldots, X_n = v_n) \cdot \left(\sum_{v_i \in V_i} P(X_1 = v_1, \ldots, X_n = v_n) \right)^{-1}$. □

With a slight abuse of notation, we write $P(X_i \mid X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)$ to denote the CPT assigning to each outcome of $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n$ the conditional probability of X_i, i.e., the function $\lambda v_1 \in V_1, \ldots, v_{i-1} \in V_{i-1}, v_{i+1} \in V_{i+1}, \ldots, v_n \in V_n . P(X_i \mid X_1 = v_1, \ldots, X_{i-1} = v_{i-1}, X_{i+1} = v_{i+1}, \ldots, X_n = v_n)$. We call $P(X_i \mid X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)$ the CPT of X_i indexed by $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n$.

Conditional probability distributions (and tables) can be simplified by exploiting the independence and conditional independence between random variables.

Definition 2.12. Let P be a joint probability distribution over $X_1 = \langle V_1, P_1 \rangle$, ..., $X_n = \langle V_n, P_n \rangle$. Furthermore, let $X_i = \langle V_i, P_i \rangle$, $X_j = \langle V_j, P_j \rangle$, and $X'_1 = \langle V'_1, P'_1 \rangle, \ldots, X'_k = \langle V'_k, P'_k \rangle$ be random variables in $\{X_1, \ldots, X_n\}$. Two random variables X_i and X_j are *independent* iff for all $v_i \in V_i$ and $v_j \in V_j$, $P_{X_i, X_j}(X_i = v_i, X_j = v_j) = P_{X_i}(X_i = v_i) P_{X_j}(X_j = v_j)$. Two random variables X_i and X_j are *conditionally independent given* $X'_1, \ldots, X'_k$ iff for all $v_i \in V_i$, $v_j \in V_j$, $v'_1 \in V'_1, \ldots, v'_k \in V'_k$, $P_{X_i, X_j, X'_1, \ldots, X'_k}(X_i = v_i, X_j = v_j \mid X'_1 = v'_1, \ldots, X'_k = v'_k) = P_{X_i, X'_1, \ldots, X'_k}(X_i = v_i \mid X'_1 = v'_1, \ldots, X'_k = v'_k) P_{X_j, X'_1, \ldots, X'_k}(X_j = v_j \mid X'_1 = v'_1, \ldots, X'_k = v'_k)$. □

We can, therefore, represent a joint probability distribution over $X_1, \ldots, X_n$ as a sequence of conditional probability tables. Furthermore, we can exploit independence and conditional independence among random variables to simplify these tables.

We are now finally ready to define Bayesian Networks.

Definition 2.13. A Bayesian Network bn is a tuple $\langle N, E, D, CPT, \preceq \rangle$, where N is the set of nodes, E is the set of directed edges, D is a function associating to each node $n \in N$ its domain, CPT is a function associating to each node $n \in N$ its conditional probability distribution, and $\preceq$ is a total ordering of the nodes in N. We denote by $parents(n)$ the set of n's parents, i.e., $parents(n) = \{n' \mid n' \to n \in E\}$. Furthermore, we denote by $ancestors(n)$ the set of n's ancestors, i.e., $ancestors(n) = parents(n) \cup \bigcup_{n' \in parents(n)} ancestors(n')$.

A Bayesian Network $\langle N, E, D, CPT, \preceq \rangle$ must satisfy the following constraints:

- $\langle N, E \rangle$ is a directed acyclic graph.

- For each node $n \in N$ such that $parents(n) = \emptyset$, $CPT(n)$ is a probability distribution over $D(n)$.
- For each node $n \in N$ such that $parents(n) \neq \emptyset$, $CPT(n)$ is a conditional probability table over $D(n)$ indexed by $D(n_1), \ldots, D(n_k)$, where $\langle n_1, \ldots, n_k \rangle$ is the tuple obtained by sorting $parents(n)$ according to $\preceq$. □

Finally, we formalize the BN's semantics, i.e., the probability distribution represented by the Bayesian Network.

Definition 2.14. Let $bn = \langle N, E, D, CPT, \preceq \rangle$ be a Bayesian Network, where $N = \{n_1, \ldots, n_k\}$. Furthermore, let $toTpl$ be a function that takes as input a subset S of N and returns the tuple obtained by sorting the nodes in S according to $\preceq$. Finally, let CPT^* be the function defined as follows: $CPT^*(n) = CPT(n)$ if $parents(n) = \emptyset$ and otherwise $CPT^*(n)$ is

$$\lambda v \in D(n). \sum_{v_{p_1} \in D(p_1)} \ldots \sum_{v_{p_k} \in D(p_k)} CPT(n)(\langle v_{p_1}, \ldots, v_{p_k} \rangle)(v) \cdot \prod_{p_i \in parents(n)} CPT^*(p_i)(v_{p_i}),$$

where $\langle p_1, \ldots, p_k \rangle = toTpl(parents(n))$. Note that $CPT^*(n)$ is a probability distribution over $D(n)$.

The BN bn defines a joint probability distribution $[\![bn]\!]$ over the random variables $X_1 = \langle D(n_1), CPT^*(n_1) \rangle, \ldots, X_k = \langle D(n_k), CPT^*(n_k) \rangle$, where

$$[\![bn]\!] = \lambda v_1 \in D(n_1), \ldots, v_k \in D(n_k). \prod_{n_i \in N'} CPT(n_i)(v_i) \cdot \prod_{n_i \in N''} CPT(n_i)(toTpl(parents(n_i))[\langle n_1, \ldots, n_k \rangle \mapsto \langle v_1, \ldots, v_k \rangle])(v_i),$$

where $N' = \{n \in N \mid parents(n) = \emptyset\}$, $N'' = \{n \in N \mid parents(n) \neq \emptyset\}$. □

When D and $\preceq$ are clear from the context, we refer to a Bayesian Network as a triple $\langle N, E, CPT \rangle$ instead of a 5-tuple $\langle N, E, CPT, D, \preceq \rangle$.

Before concluding, we introduce some notation to query Bayesian Networks. Let $bn = \langle N, E, D, CPT, \preceq \rangle$ be a Bayesian Network. A *bn-total assignment* is a total function ν that associates to each $n \in N$ a value in $v \in D(n)$, whereas a *bn-partial assignment* is a partial function ν that associates to each $n \in N'$, where $N' \subseteq N$, a value in $v \in D(n)$. We now lift the semantics of a BN to assignments. The probability defined by bn given a partial assignment ν, written $[\![bn]\!](\nu)$ is $\sum_{w_1 \in D(m_1)} \cdots \sum_{w_l \in D(m_l)} [\![bn]\!](toTpl(N)(\nu \uplus [\langle m_1, \ldots, m_l \rangle \mapsto \langle w_1, \ldots, w_l \rangle]))$, where $dom(\nu) = \{n_1, \ldots, n_k\}$ and $N \setminus dom(\nu) = \{m_1, \ldots, m_l\}$. Namely, $[\![bn]\!](\nu)$ returns the probability that the random variables associated with the nodes for which ν is defined have the outcome indicated by ν.

Part II

Protecting databases against `SELECT`-only attackers

Chapter 3

Database Access Control for SELECT-only attackers

> **Mike Michaelson**: *Christof, let me ask you, why do you think that Truman has never come close to discovering the true nature of his world until now?*
> **Christof**: *We accept the reality of the world with which we're presented. It's as simple as that.*
>
> The Truman Show (1998)

Retrieving data from a database is usually done via `SELECT` queries, which allow one to retrieve all tuples matching a given criterion. `SELECT` queries are one of the most common ways of interacting with a database. For this reason, many DBAC approaches target the so-called `SELECT`-only attacker model, where attackers can interact with the database only through `SELECT` queries. In this setting, therefore, both the database's content and the security policy are fixed.

In this chapter, we formalize and study *Security-Aware Query Processing*, the problem of computing answers to `SELECT` queries in the presence of security policies. Observe that Security-Aware Query Processing models DBAC in the `SELECT`-only setting. In the following, we present general impossibility results for the existence of optimal algorithms for Security-Aware Query Processing and classify query languages for which such algorithms exist. In particular, we show that for the relational calculus there are no optimal algorithms, whereas optimal algorithms exist for some of its fragments, such as the existential fragment.

We also establish relationships between two different models of Fine-Grained Access Control, called Truman and Non-Truman models, which have been previously presented in the literature as distinct. For optimal Security-Aware Query Processing, we show that the Non-Truman model is a special case of the Truman model for boolean queries in the relational calculus, moreover the two models coincide for more powerful query languages, such as the relational calculus with aggregation operators. In contrast, these two models are distinct for non-boolean queries. Note that this chapter is largely based on [84].

Organization. In Section 3.1, we introduce the Truman and Non-Truman models for database access control and the concept of optimal algorithms. In Section 3.2, we present the fragments of the relational calculus used in the rest of the chapter. In Section 3.3 we introduce our security policies, and in Section 3.4 we introduce Security-Aware Query Processing. We present our results in Section 3.5 for boolean queries and in Section 3.6 for non-boolean queries. In Section 3.7, we review related work and we draw conclusions in Section 3.8. Most of the proofs of our results are directly given in this chapter. For the sake of readability, some proofs are given in Appendix A.

3.1 Introduction

Computing answers to `SELECT` queries given a security policy is the main task of DBAC mechanisms in the `SELECT`-only attacker model. There is, however, no common terminology to refer to this problem. For example, "secure querying" [58], "enforcing data confidentiality" [125], and "Fine-Grained Access Control" [13, 131, 165] have all been used in this context. In the following, we refer to this problem as *Security-Aware Query Processing* (SAQP). This differs from *Secure Query Processing* since the latter usually refers to querying encrypted data. Security-Aware Query Processing algorithms are implemented in commercial databases [8, 41, 150], and solutions have been proposed for other settings like XML files [58] or RDF graphs [125].

Rizvi et al. [131] identified two distinct models, called *Truman* and *Non-Truman* models, that capture different approaches to the SAQP problem. The term *Truman* model comes from the protagonist of the movie *The Truman Show* who is unaware that he lives in an artificial environment where

Queries	SAQP	Truman Model	Non-Truman Model
Boolean	$\neg\exists optimal$	undecidable fragments (Theorem 3.2)	
	$\exists optimal$	sufficient conditions (Lemma 3.1) *ERC* (Theorem 3.3)	
Non-Boolean	$\neg\exists optimal$	undecidable fragments (Theorem 3.6)	undecidable fragments (Theorem 3.9)
	$\exists optimal$	sufficient conditions (Lemma 3.3) *ERC* (Theorem 3.7)	sufficient conditions (Lemma 3.4) *ERC* (Theorem 3.10)

TABLE 3.1: Summary of results. Additionally, Theorem 3.4 proves that the Non-Truman model is a special case of the Truman model for boolean queries, whereas Corollary 3.1 proves that the two models are distinct for non-boolean queries.

everything he experiences is externally controlled. Algorithms in the Truman model transparently modify all user queries to restrict the user's access to only the data authorized by the security policy. The query's result may differ from the unrestricted one to preserve confidentiality. For example, suppose we issue a query q asking for the names of all employees, but we are authorized only to read some of their records. If the company has 1000 employees and we have access to just 800 of them, the original query is modified, and we get just the names of the 800 employees we are authorized to read. However, such modifications may cause inconsistencies between user's expectations and the query's result. For instance, if we have access to the total number of employees and we know that there are actually 1000 employees, then the modified result is inconsistent with our knowledge.

In contrast to the above, the *Non-Truman* model solves the problem of inconsistencies. Algorithms in this model execute a query iff it can be answered using only information the user is authorized to read under the given policy (these queries are called *conditionally valid*) and otherwise the query is rejected. In our example, in the Non-Truman model the query q is rejected because it cannot be answered using only authorized information. In contrast, a query q' asking the names of the 800 employees we are authorized to read is executed. The two models target different needs. The Truman model ensures higher data availability at the price of partial results and inconsistencies, whereas the Non-Truman model ensures consistency at the expense of lower data availability.

Wang et al. [165] analyzed query processing algorithms in the Truman model and proposed three correctness criteria: *security*, *soundness*, and *maximality*. Security requires that a query's result does not depend on sensitive information, soundness requires that an algorithm returns only correct information, and maximality requires that an algorithm returns as much information as possible without violating the given policy. In the Truman model, we say that an algorithm is *optimal* iff it satisfies all three of these criteria, whereas in the Non-Truman model, an algorithm is *optimal* iff it executes exactly the conditionally valid queries. In both models, optimal algorithms are what we ideally want as they are the only algorithms that guarantee security, correctness, and data availability. For instance, in the Non-Truman model, an algorithm that rejects all queries is secure but useless, whereas an algorithm that executes all queries is useful but insecure. Similarly, in the Truman model, an algorithm that always returns $\emptyset$ as a query's result is secure and sound, an algorithm that always returns the original query's result is sound and maximal, and an algorithm that systematically introduces noise in the result can be secure and maximal.

Unfortunately, a thorough analysis of optimal Security-Aware Query Processing has been missing until now. For the Non-Truman model, Rizvi et al. [131] left open the decidability of the conditional validity problem. Although Zhang et al. [170] proved that it is decidable for conjunctive queries and Koutris et al. [107] extended this result to unions of conjunctive queries, there is no general characterization of the problem. For the Truman model, Wang et al. [165] claimed that "while the maximality property is desirable, it is difficult to achieve" for algorithms that are secure and sound. They presented an informal example supporting this claim but they did not give a proof.

In the following, we study the decidability of optimal Security-Aware Query Processing for boolean and non-boolean queries in both models and we provide answers to all the above open problems. We prove possibility and impossibility results for the relational calculus (RC) and for the existential fragment of RC (ERC). We also establish connections between Truman and Non-Truman models. Table 3.1 summarizes our main results and the associated theorems.

3.2 Fragments of the relational calculus

We now introduce the fragments of the relational calculus that we study in this chapter.

A *fragment F of the relational calculus* is a subset $F \subseteq RC$ of the relational calculus formulae. To stress that a formula ϕ belongs to a fragment F, we call it an F-formula. We say that a boolean query ϕ (respectively a non-boolean query $\{\overline{x} \mid \phi\}$) is in a fragment F iff $\phi \in F$.

Language	$FINSAT^F$	$FINVAL^F$
RC	Undecidable	Undecidable
$BSRRC$	Decidable	Undecidable
ERC	Decidable	Decidable

TABLE 3.2: Decidability of the $FINSAT^F$ and $FINVAL^F$ decision problems (taken from [37]).

We now introduce the existential fragment ERC of the relational calculus. Note that the ERC fragment strictly contains conjunctive queries (CRC).

Definition 3.1. Let D be a database schema. The formula $\phi(\overline{y}) = \exists \overline{x}.\, \psi(\overline{x}, \overline{y})$ is a *ERC-formula over D* iff ψ is a quantifier-free RC-formula over D. □

For technical reasons, we also consider the Bernays-Schönfinkel-Ramsey fragment $BSRRC$.

Definition 3.2. Let D be a database schema. The formula $\phi(\overline{z}) = \exists \overline{x}. \forall \overline{y}.\, \psi(\overline{x}, \overline{y}, \overline{z})$ is a *$BSRRC$-formula over D* iff ψ is a quantifier-free RC-formula over D. □

In the following, we use the term *query language* to refer to the relational calculus and its fragments, such as ERC. We assume that there is a unique encoding of formulae as natural numbers, and in our proofs we switch freely between formulae and their encodings.

3.2.1 Decision Problems

Here we define the finite satisfiability and finite validity decision problems for (fragments of) the relational calculus.

Definition 3.3. Given a query language F, $FINSAT^F$ denotes the following decision problem:
Input: A database schema D and a sentence $\phi \in F$.
Question: Is there a state $db \in \Omega_D$ such that $[\phi]^{db} = \top$? □

Definition 3.4. Given a query language F, $FINVAL^F$ denotes the following decision problem:
Input: A database schema D and a sentence $\phi \in F$.
Question: For all states $db \in \Omega_D$, is $[\phi]^{db} = \top$? □

Table 3.2 presents decidability results from [37] for these two problems for the query languages used in this chapter.

3.3 Security Policies

Various Fine-Grained Access Control models for databases have been proposed in the literature [8, 41, 110, 150]. Although each has its own features, the models share common characteristics: (1) they support access control constraints at the row or column level, and (2) access control constraints are given by formulae, e.g., expressed in SQL [110], referring to the database's current state or to the user's credentials. We now describe a security policy model that captures the main features of existing fine-grained access control models.

Let F be a query language. An *F-constraint* is a pair $\langle \{\overline{x} \mid \psi\}, \phi \rangle$, where $\{\overline{x} \mid \psi\}$ is a non-boolean F-query and ϕ is an F-formula such that $free(\phi) \subseteq free(\psi)$. A *row-level F-constraint* is just an F-constraint $\langle \{\overline{x} \mid \psi\}, \phi \rangle$ that specifies the conditions ϕ under which we are authorized to read a tuple in the result of the query $\{\overline{x} \mid \psi\}$. A *column-level F-constraint* is a triple $\langle \{\overline{x} \mid \psi\}, \phi, i \rangle$ that specifies the conditions ϕ under which we are authorized to read the i-th value of a tuple in the result of the query $\{\overline{x} \mid \psi\}$, where $\langle \{\overline{x} \mid \psi\}, \phi \rangle$ is an F-constraint and $i \in \{1, \ldots, |\overline{x}|\}$. Note that given an F-constraint $\langle \{\overline{x} \mid \psi\}, \phi \rangle$, a free variable z in ϕ refers to the i-th value of the tuple on which the constraint ϕ is evaluated, where $i = pos(\overline{x}, z)$. In the following, we consider only *domain independent* F-constraints, i.e., constraints $\langle \{\overline{x} \mid \psi\}, \phi \rangle$ such that $\{\overline{x} \mid \psi \wedge \phi\}$ is domain-independent. As a result, the access control decision does not depend on the underlying domain.

The constraints are expressed using formulae that can refer to the current state and to the values of the tuple being accessed. Note that the constraints can be defined over arbitrary non-boolean queries. We can therefore restrict access not only to the data in the relation schemas but also to derived information, such as tuples in views.

A security policy in our model is defined as follows.

Definition 3.5. Let F be a query language and D be a database schema. An *F-security policy S* over D is a pair $\langle ROW, COL \rangle$, where ROW is a set of row-level F-constraints and COL is a set of column-level F-constraints, such that for any non-boolean query q: (1) there is at most one constraint in ROW associated with q, and (2) there is at most one constraint in COL associated with the i-th value of the tuples in the result of q. □

Employee		
John	25	Clerk
Frank	17	Admin
Jack	36	SecretAgent

db_1

Employee		
John	25	Clerk
Frank	16	Admin
Jack	36	SecretAgent
Carl	26	SecretAgent

db_2

Employee		
John	25	Clerk
Frank	19	Admin
Jack	36	SecretAgent
Carl	26	SecretAgent

db_3

Employee		
John	25	Clerk
Frank	17	Admin

db_4

FIGURE 3.1: Some states.

Let D be a database schema, $S = \langle ROW, COL\rangle$ be a security policy over D, $\{\overline{x} \mid \psi\}$ be a non-boolean query, and $db \in \Omega_D$ be a state. We say that S discloses the tuple $\overline{t} \in [\{\overline{x} \mid \psi\}]^{db}$, denoted by $Disc_S(\overline{t}, \{\overline{x} \mid \psi\}, db)$, iff there is a constraint $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$ such that $[\phi[\overline{x} \mapsto \overline{t}]]^{db} = \top$. Similarly, we say that S discloses the i-th value of the tuple $\overline{t} \in [\{\overline{x} \mid \psi\}]^{db}$, denoted by $Disc_S(\overline{t}, \{\overline{x} \mid \psi\}, db, i)$, iff $Disc_S(\overline{t}, \{\overline{x} \mid \psi\}, db)$ and there is a constraint $\langle\{\overline{x} \mid \psi\}, \phi, i\rangle \in COL$ such that $[\phi[\overline{x} \mapsto \overline{t}]]^{db} = \top$, where $i \in \{1, \ldots, |\overline{x}|\}$. Note that the authorization to read a tuple does not imply the authorization to read any of its values. Given a constraint $\langle q, \phi\rangle \in ROW$, we denote by $Auth_{S,q}(db)$ the set of tuples in $[q]^{db}$ disclosed by S in the state db, i.e., $Auth_{S,q}(db) := \{\overline{t} \in [q]^{db} \mid Disc_S(\overline{t}, q, db)\}$.

Example 3.1. Let D be a database schema containing only one relation schema *Employee* with arity 3. We assume that the first column in the schema stores employees' names, the second column stores their ages, and the last one stores their jobs. We want to prevent the access to tuples representing secret agents. Although *name* is not protected, we want to disclose only the *age* of employees over 18, and only the *job* of the *Clerks*. We also want to disclose the set of the jobs in the database. Let q be the query $\{x, y, z \mid Employee(x, y, z)\}$, and r be the query $\{z \mid \exists x, y.\, Employee(x, y, z)\}$. The security policy $S = \langle ROW, COL\rangle$ is as follows: $ROW = \{\langle q, z \neq SecretAgent\rangle, \langle r, \top\rangle\}$, and $COL = \{\langle q, \top, 1\rangle, \langle q, \bigwedge_{i\in\{1,\ldots,17\}} y \neq i, 2\rangle, \langle q, z = Clerk, 3\rangle, \langle r, \top, 1\rangle\}$. ■

Note that a policy specifies which tuples and values we are authorized to read; it does not directly specify the result of a given query. The semantics of Security-Aware Query Processing will be introduced in the following sections.

We now introduce the concept of masked tuple from [165]. A *masked tuple* is a tuple where some values are replaced by the distinguished value †. This value prevents the disclosure of data that may be sensitive. A tuple $\overline{v}$ is *subsumed* by another tuple $\overline{t}$, written $\overline{v} \sqsubseteq \overline{t}$, iff $|\overline{v}| = |\overline{t}|$, and for all $i \in \{1, \ldots, |\overline{t}|\}$, $\overline{v}(i) = \overline{t}(i)$ or $\overline{v}(i) = \dagger$.

Example 3.2. Let q be the query $\{x, y, z \mid Employee(x, y, z)\}$ from Example 3.1. The result of q in the state db_1 in Figure 3.1 contains three tuples: $\overline{t} = \langle John, 25, Clerk\rangle$, $\overline{v} = \langle Frank, 17, Admin\rangle$, and $\overline{u} = \langle Jack, 36, SecretAgent\rangle$. The policy S discloses $\overline{t}$ and $\overline{v}$ but not $\overline{u}$ because Jack is a secret agent. Moreover, the tuple $\overline{v}$ is only partially disclosed. Indeed, we are authorized to read *Frank*'s name, but not his age or his job. The policy S also fully discloses the result of the query r, i.e., the set $\{\langle Clerk\rangle, \langle Admin\rangle, \langle SecretAgent\rangle\}$. Note that the tuple $\overline{z} = \langle Frank, \dagger, \dagger\rangle$ is subsumed by the tuple $\overline{v}$, i.e., $\overline{z} \sqsubseteq \overline{v}$. ■

Note that our security policy model uses a set semantics derived from the relational calculus, whereas related approaches use a multi-set semantics derived from SQL. The models in [8,150] support only row-level constraints, those in [41, 110] support only column-level constraints, and the model in [13] supports both. Our security policy model subsumes all these approaches as well as approaches where the security policy is expressed using views. Note that our security policies can represent, by combining column-level and row-level constraints, cell-level disclosure policies [13, 110, 165].

For simplicity, we ignore users and their credentials. This neither restricts nor limits our theoretical results. Users' credentials can be modeled as a mapping *cred* from the set $\mathcal{U}$ of users to the set $\mathcal{S}$ of security policies that assigns to each user $u \in \mathcal{U}$ a security policy $S \in \mathcal{S}$ where the users' credentials are hard-coded in S using constant values.

3.4 Security-Aware Query Processing

In this section, we introduce security-aware query processors, indistinguishable states, and correctness criteria for boolean queries in the Truman model.

Definition 3.6. Let D be a database schema, F be a query language, U be the set of all possible results, and $\mathcal{S}$ be the set of F-policies. An *F-Security-Aware Query Processor* (F-SAQP) is a function $\mathcal{M} : F \times \mathcal{S} \times \Omega_D \to U$. □

Note that the set U depends on the type of query, i.e., boolean or non-boolean. Although boolean queries can be only true ($\top$) or false ($\bot$) in a given state, a security-aware query processor may also return the third value †, i.e., $U = \{\top, \bot, \dagger\}$. This value is used to prevent the leakage of sensitive information, which is in contrast to [165] where † is used only to mask tuples in the query's result.

We now define indistinguishability, adapted from [165] to our setting.

Definition 3.7. Let D be a database schema, F be a query language, and $S = \langle ROW, COL\rangle$ be an F-policy over D. We say that two database states db_1 and db_2 in Ω_D are *indistinguishable* according to S, written $db_1 \cong_S db_2$, iff for all $\langle q, \phi\rangle \in ROW$, there is a bijection f from $Auth_{S,q}(db_1)$ to $Auth_{S,q}(db_2)$ such that for all $\bar{t} \in Auth_{S,q}(db_1)$ and all $i \in \{1, \ldots, |\bar{t}|\}$, (1) $Disc_S(\bar{t}, q, db_1, i)$ iff $Disc_S(f(\bar{t}), q, db_2, i)$, and (2) if $Disc_S(\bar{t}, q, db_1, i)$, then $\bar{t}(i) = f(\bar{t})(i)$. □

Example 3.3. The database states db_1 and db_2 in Figure 3.1 are indistinguishable according to the policy S given in Example 3.1. In contrast, $db_1 \ncong_S db_3$ because the value of the attribute *age* for *Frank* is protected in state db_1 but not in state db_3, and $db_1 \ncong_S db_4$ because the result of the query r is different in the two states. ■

Note that all the states are indistinguishable according to the empty policy $S = \langle \emptyset, \emptyset\rangle$. As stated in Proposition 3.1, given a database schema D and a policy S, the indistinguishability relation $\cong_S$ is an equivalence relation over Ω_D. Given a state $s \in \Omega_D$, the equivalence class of s defined by $\cong_S$ is denoted by $[\![s]\!]_{\cong_S}$.

Proposition 3.1. *Let D be a database schema and S be a security policy over D. The indistinguishability relation $\cong_S$ is an equivalence relation over Ω_D.*

Proof. Let D be a database schema and $S = \langle ROW, COL\rangle$ be a security policy over D. We now show that $\cong_S$ is reflexive, symmetric, and transitive.

Reflexivity: Let $db \in \Omega_D$. For each $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, the identity function $i : Auth_{S,\{\overline{x}\mid\psi\}}(db) \to Auth_{S,\{\overline{x}\mid\psi\}}(db)$ is a bijection on the set $Auth_{S,\{\overline{x}\mid\psi\}}(db)$ to itself such that for all $\bar{t} \in Auth_{S,\{\overline{x}\mid\psi\}}(db)$ and all $j \in \{1, \ldots, |\overline{x}|\}$, (1) $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db, j)$ iff $Disc_S(i(\bar{t}), \{\overline{x} \mid \psi\}, db, j)$, and (2) if $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, s, j)$, then $\bar{t}(j) = i(\bar{t})(j)$. Therefore $db \cong_S db$.

Symmetry: Let $db_1, db_2 \in \Omega_D$ be two database states such that $db_1 \cong_S db_2$. From the indistinguishability definition, it follows that for each $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, there is a bijection $f_{\{\overline{x}\mid\psi\}}$ from $Auth_{S,\{\overline{x}\mid\psi\}}(db_1)$ to $Auth_{S,\{\overline{x}\mid\psi\}}(db_2)$ such that for all $\bar{t} \in Auth_{S,\{\overline{x}\mid\psi\}}(db_1)$ and all $i \in \{1, \ldots, |\overline{x}|\}$, (1) $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_1, i)$ iff $Disc_S(f_{\{\overline{x}\mid\psi\}}(\bar{t}), \{\overline{x} \mid \psi\}, db_2, i)$, and (2) if $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_1, i)$, then $\bar{t}(i) = f_{\{\overline{x}\mid\psi\}}(\bar{t})(i)$. Therefore, for each $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, there is a bijection $f^{-1}_{\{\overline{x}\mid\psi\}}$ from $Auth_{S,\{\overline{x}\mid\psi\}}(db_2)$ to $Auth_{S,\{\overline{x}\mid\psi\}}(db_1)$ such that for all $\bar{t} \in Auth_{S,\{\overline{x}\mid\psi\}}(db_2)$ and all $i \in \{1, \ldots, |\overline{x}|\}$, (1) $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_2, i)$ iff $Disc_S(f^{-1}_{\{\overline{x}\mid\psi\}}(\bar{t}), \{\overline{x} \mid \psi\}, db_1, i)$, and (2) if $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_2, i)$, then $\bar{t}(i) = f^{-1}_{\{\overline{x}\mid\psi\}}(\bar{t})(i)$. Hence, $db_2 \cong_S db_1$.

Transitivity: Let $db_1, db_2, db_3 \in \Omega_D$ be three states such that $db_1 \cong_S db_2$ and $db_2 \cong_S db_3$. From $db_1 \cong_S db_2$, it follows that for each $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, there is a bijection $f^{1\to2}_{\{\overline{x}\mid\psi\}}$ from $Auth_{S,\{\overline{x}\mid\psi\}}(db_1)$ to $Auth_{S,\{\overline{x}\mid\psi\}}(db_2)$ such that for all $\bar{t} \in Auth_{S,\{\overline{x}\mid\psi\}}(db_1)$ and all $i \in \{1, \ldots, |\overline{x}|\}$, (1) $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_1, i)$ iff $Disc_S(f^{1\to2}_{\{\overline{x}\mid\psi\}}(\bar{t}), \{\overline{x} \mid \psi\}, db_2, i)$, and (2) if $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_1, i)$ holds, then $\bar{t}(i) = f^{1\to2}_{\{\overline{x}\mid\psi\}}(\bar{t})(i)$. From $db_2 \cong_S db_3$, it follows that for each $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, there is a bijection $f^{2\to3}_{\{\overline{x}\mid\psi\}}$ from $Auth_{S,\{\overline{x}\mid\psi\}}(db_2)$ to $Auth_{S,\{\overline{x}\mid\psi\}}(db_3)$ such that for all $\bar{t} \in Auth_{S,\{\overline{x}\mid\psi\}}(db_2)$ and all $i \in \{1, \ldots, |\overline{x}|\}$, (1) $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_2, i)$ iff $Disc_S(f^{2\to3}_{\{\overline{x}\mid\psi\}}(\bar{t}), \{\overline{x} \mid \psi\}, db_3, i)$, and (2) if $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_2, i)$ holds, then $\bar{t}(i) = f^{2\to3}_{\{\overline{x}\mid\psi\}}(\bar{t})(i)$. Therefore, for each $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, there is a bijection $f^{1\to3}_{\{\overline{x}\mid\psi\}} := f^{2\to3}_{\{\overline{x}\mid\psi\}} \circ f^{1\to2}_{\{\overline{x}\mid\psi\}}$ from $Auth_{S,\{\overline{x}\mid\psi\}}(db_1)$ to $Auth_{S,\{\overline{x}\mid\psi\}}(db_3)$ such that for all $\bar{t} \in Auth_{S,\{\overline{x}\mid\psi\}}(db_1)$ and all $i \in \{1, \ldots, |\overline{x}|\}$, (1) $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_1, i)$ iff $Disc_S(f^{1\to3}_{\{\overline{x}\mid\psi\}}(\bar{t}), \{\overline{x} \mid \psi\}, db_3, i)$, and (2) if $Disc_S(\bar{t}, \{\overline{x} \mid \psi\}, db_1, i)$ holds, then $\bar{t}(i) = f^{1\to3}_{\{\overline{x}\mid\psi\}}(\bar{t})(i)$. Hence, $db_1 \cong_S db_3$. □

We now adapt the correctness criteria given in [165] to boolean queries. Informally, a secure SAQP $\mathcal{M}$ is not influenced by data protected by a given security policy, i.e., $\mathcal{M}$'s result does not depend on undisclosed data.

$$\mathcal{M}_{opt}(\phi, S, db) = \begin{cases} \top & \text{if } \forall db' \in [\![db]\!]_{\cong_S}. [\phi]^{db'} = \top \\ \bot & \text{if } \forall db' \in [\![db]\!]_{\cong_S}. [\phi]^{db'} = \bot \\ \dagger & \text{otherwise} \end{cases}$$

FIGURE 3.2: An optimal SAQP for boolean queries.

Definition 3.8. Let D be a database schema and F be a query language. An F-security-aware query processor $\mathcal{M}$ is *secure* iff for all F-policies S over D, all F-queries q, and all $db, db' \in \Omega_D$, if $db \cong_S db'$, then $\mathcal{M}(q, S, db) = \mathcal{M}(q, S, db')$. □

A sound SAQP $\mathcal{M}$ returns only correct information, i.e., information already returned by the original query. Note that $\mathcal{M}(q, S, db)$ may return less information than $[q]^{db}$, but never more information. For boolean queries, $\mathcal{M}(q, S, db)$ can return either $[q]^{db}$ or $\dagger$.

Definition 3.9. Let D be a database schema and F be a query language. An F-security-aware query processor $\mathcal{M}$ is *sound* iff for all queries $q \in F$, all F-policies S over D, and all $db \in \Omega_D$, $\mathcal{M}(q, S, db) = [q]^{db}$ or $\mathcal{M}(q, S, db) = \dagger$. □

Finally, maximality formalizes that $\mathcal{M}$ returns as much information as possible. For boolean queries, a maximal SAQP, given a query q, returns $[q]^{db}$ for all states db where it is secure to do so.

Definition 3.10. Let D be a database schema and F be a query language. An F-security-aware query processor $\mathcal{M}$ is *maximal* iff for all F-security policies S over D, all F-queries q, and all $db \in \Omega_D$, if $[q]^{db} = [q]^{db'}$ for all $db' \in [\![db]\!]_{\cong_S}$, then $\mathcal{M}(q, S, db) = [q]^{db}$. □

We are now ready to define optimal algorithms for boolean queries in the Truman model.

Definition 3.11. Let F be a query language. An F-SAQP for boolean queries is *optimal in the Truman model* iff it satisfies the Definitions 3.8–3.10. □

Optimal algorithms are those algorithms that return as much information as possible without returning incorrect information or violating the security policy. Figure 3.2 describes an optimal SAQP for boolean queries. Observe that, depending on the query language, $\mathcal{M}_{opt}$ may not be computable.

Studying optimal algorithms is important as they are the best one can do in terms of SAQP. In particular, considering only two out of three correctness criteria is usually not enough. Indeed, a function $\mathcal{M}_1(q, S, db) = \dagger$ for all q, S, and db is secure and sound but is completely useless. Similarly, a function $\mathcal{M}_2(q, S, db) = [q]^{db}$ for all q, S, and db is sound and maximal but leaks sensitive information. Finally, a function $\mathcal{M}_3$ that satisfies the security and maximality criteria can systematically return arbitrary results without violating security and maximality. Indeed, let E be one of the partitions of Ω_D defined by $\cong_S$ such that $[q]^{db} \neq [q]^{db'}$ for some $db, db' \in E$, then $\mathcal{M}_3$ can return an arbitrary value for all states in E.

3.5 Boolean Queries

In this section, we study the existence of optimal Security-Aware Query Processing algorithms for boolean queries.

3.5.1 Preliminaries

We first define the query agreement decision problem. Afterwards, we show how this problem is related to optimal Security-Aware Query Processing.

Definition 3.12. Given a language F, $AGREE^F$ denotes the following decision problem:
Input: A database schema D, an F-security policy S over D, a boolean query $q \in F$, and a state $db \in \Omega_D$.
Question: For all states $db' \in [\![db]\!]_{\cong_S}$, is $[q]^{db'} = [q]^{db}$? □

Theorem 3.1 establishes that the decidability of $AGREE^F$ is related to the existence of optimal SAQP algorithms for the query language F.

Theorem 3.1. *Let F be a query language. There is a computable optimal F-SAQP algorithm $\mathcal{M}$ for boolean queries iff $AGREE^F$ is decidable.*

Proof. ($\Rightarrow$) Let $\mathcal{M}$ be a computable optimal F-SAQP algorithm for boolean queries. We use $\mathcal{M}$ as a subroutine in a decision procedure for $AGREE^F$. $AGREE^F$ takes as input a database schema D,

a policy S over D, a boolean F-query q, and a state db. If $\mathcal{M}(q, S, db) = \dagger$, then $AGREE^F(D, S, q, db) = \bot$, otherwise $AGREE^F(D, S, q, db) = \top$.

($\Leftarrow$) We use $AGREE^F$ to build an optimal F-SAQP $\mathcal{M}$. Let D be a database schema. $\mathcal{M}$ takes as input an F-policy S over D, a boolean F-query q, and a state $db \in \Omega_D$. If $AGREE^F(D, S, q, db) = \top$, then $\mathcal{M}(q, S, db) = [q]^{db}$, otherwise $\mathcal{M}(q, S, db) = \dagger$. It easy to see that $\mathcal{M}$ is optimal and computable. □

We next study the decidability of $AGREE^F$ for RC and ERC, and we extend the results to the existence of optimal algorithms for these query languages. In our theorem statements and proofs, we will ignore sub-recursive complexity bounds and all reductions will be Turing reductions.

3.5.2 Impossibility Results

We now show that there are no optimal SAQP algorithms for boolean RC queries. Since all query languages used in practice, such as SQL, are Codd-complete [10], i.e., they are at least as expressive as RC, it follows from our impossibility result that there are no optimal SAQP algorithms for the boolean query languages currently in use.

Theorem 3.2. *$AGREE^{RC}$ is undecidable.*

Proof. Let S be the empty policy $\langle \emptyset, \emptyset \rangle$ and D be a database schema. Then, $\cong_S$ defines just one equivalence class containing every state. An RC-sentence ϕ is finitely satisfiable iff $AGREE^{RC}(D, S, \phi, db) = \bot$ or $[\phi]^{db} = \top$, where db is the empty state. The reduction can be implemented by a total Turing machine. Since $FINSAT^{RC}$ is undecidable, so is $AGREE^{RC}$.

We now show the reduction's correctness. Assume that ϕ is finitely satisfiable. If $[\phi]^{db} = \top$, then $AGREE^{RC}(D, S, \phi, db) = \bot$ or $[\phi]^{db} = \top$ trivially holds. Assume, then, that $[\phi]^{db} = \bot$. From this and ϕ is satisfiable, it follows that there exists a database state $db' \in \Omega_D$ such that $[\phi]^{db} \neq [\phi]^{db'}$. From this and $S = \langle \emptyset, \emptyset \rangle$, $db' \in [\![db]\!]_{\cong_S}$. From this, there is a state $db' \in [\![db]\!]_{\cong_S}$ such that $[\phi]^{db} \neq [\phi]^{db'}$. Hence, $AGREE^{RC}(D, S, \phi, db) = \bot$. For the other direction, assume that $AGREE^{RC}(D, S, \phi, db) = \bot$ or $[\phi]^{db} = \top$. If $[\phi]^{db} = \top$, ϕ is trivially finitely satisfiable. In contrast, if $AGREE^{RC}(D, S, \phi, db) = \bot$, there is a database state $db' \in [\![db]\!]_{\cong_S}$ such that $[\phi]^{db'} \neq [\phi]^{db}$. Therefore, ϕ is satisfied in db or db'. □

From Theorem 3.2, it follows that there are no optimal SAQP algorithms for boolean RC-queries. Similarly to Theorem 3.2, we can prove an even stronger result: for any fragment F of RC such that $FINSAT^F$ (or $FINVAL^F$) is undecidable, then $AGREE^F$ is also undecidable. Therefore, from well-known undecidability results for fragments of RC, one can identify fragments for which there are no optimal SAQP algorithms. Note also that considering functional dependencies and other integrity constraints in a particular fragment of RC might cause the undecidability of the $AGREE$ problem, and therefore the impossibility of optimal SAQP.

3.5.3 Possibility Results

Although $AGREE^{RC}$ is undecidable, there are fragments of RC for which the problem is decidable. In this section, we present sufficient conditions for the decidability of $AGREE^F$, and we use these conditions to prove the decidability of $AGREE^{ERC}$. Note that we do not provide a complete classification of the fragments of RC for which $AGREE^F$ is decidable and there are fragments that meet neither the conditions for undecidability stated above nor the conditions of Lemma 3.1 below. Moreover, although we prove decidability, we do not derive complexity bounds for optimal SAQP algorithms.

We first introduce the notion of encoding the indistinguishability relation in a formula. Let D be a database schema, S be a security policy over D, and db be a state in Ω_D. We say that a sentence $\phi_{INDIST(S,db)}$ *encodes the indistinguishability relation* defined by the policy S on the state db iff $\phi_{INDIST(S,db)}$ is domain independent and for all $db' \in \Omega_D$, $[\phi_{INDIST(S,db)}]^{db'} = \top$ iff $db \cong_S db'$.

We now present sufficient conditions for the decidability of the $AGREE^F$ decision problem.

Lemma 3.1. *Let F be a query language. $AGREE^F$ is decidable if there is a query language F' such that:*

1. *$FINSAT^{F'}$ is decidable.*
2. *For any $db \in \Omega_D$ and any F-policy S, we can compute a sentence $\phi_{INDIST(S,db)}$ that encodes the indistinguishability relation.*
3. *We can compute sentences $\gamma, \gamma' \in F'$ equivalent to $\phi_{INDIST(S,db)} \wedge \psi$ and $\phi_{INDIST(S,db)} \wedge \neg\psi$ respectively, for any $db \in \Omega_D$, any F-policy S, and any sentence $\psi \in F$.*

Proof. Let F be a query language and F' be another query language satisfying the conditions (1)–(3). We prove the theorem by reducing $AGREE^F$ to $FINSAT^{F'}$. The inputs to the $AGREE^F$ problem are: a database schema D, an F-security policy S, an F-sentence ϕ, and a state $db \in \Omega_D$. From (2) and (3), we can compute two F'-formulae $\gamma_\top(S, db, \psi)$ and $\gamma_\bot(S, db, \psi)$ that are respectively equivalent to $\phi_{INDIST(S,db)} \wedge \neg\psi$ and $\phi_{INDIST(S,db)} \wedge \psi$ for any state $db \in \Omega_D$, any $\psi \in F$, and any F-policy S over D. The algorithm for $AGREE^F$ first computes $k = [\psi]^{db}$. Then, $AGREE^F(D, S, \psi, db) = \top$ iff $FINSAT^{F'}(\gamma_k(S, db, \phi)) = \bot$. This procedure can be implemented by a total Turing machine. Moreover, from (1) and (3), it follows that $\gamma_k(S, db, \phi) \in F'$ and $FINSAT^{F'}$ is decidable. Hence, $AGREE^F$ is decidable.

To show this reduction's correctness, we prove both directions.

($\Leftarrow$). Assume that $FINSAT^{F'}(\gamma_k(S, db, \phi)) = \bot$. Without loss of generality, we assume that $[\psi]^{db} = \bot$. Therefore, $\gamma_k(S, db, \phi)$ is $\phi_{INDIST(S,db)} \wedge \psi$. From this and $FINSAT^{F'}(\gamma_k(S, db, \phi)) = \bot$, it follows that there is no database state db' such that $[\phi_{INDIST(S,db)} \wedge \psi]^{db'} = \top$. From this, it follows that for all database states db' such that $[\phi_{INDIST(S,db)}]^{db'} = \top$, $[\psi]^{db'} = \bot$. From this and (2), it follows that for all database states $db' \in [\![db]\!]_{\cong_S}$, $[\psi]^{db'} = \bot$. Hence, $AGREE^F(D, S, \psi, db) = \top$.

($\Rightarrow$). Assume that $AGREE^F(D, S, \psi, db) = \top$. Without loss of generality, we assume $[\psi]^{db} = \top$. The proof for the other case is analogous. From $AGREE^F(D, S, \psi, db) = \top$ and $[\psi]^{db} = \top$, it follows that $[\psi]^{db'} = \top$ for all $db' \in [\![db]\!]_{\cong_S}$. From this and (2), $[\psi]^{db'} = \top$ for all db' such that $[\phi_{INDIST(S,db)}]^{db'} = \top$. From this, the formula $\gamma_\top(S, db, \psi) = \phi_{INDIST(S,db)} \wedge \neg\psi$ is unsatisfiable and, therefore, $FINSAT^{F'}(\gamma_k(S, db, \phi)) = \bot$. □

There are fragments of RC that are not expressive enough to encode the indistinguishability relation. For this reason, in Lemma 3.1, we introduced another fragment F' that is more expressive and allows such an encoding. For instance, in the ERC fragment there is no encoding of the indistinguishability relation, but it can be encoded in the more expressive $BSRRC$ fragment.

There is a similar set of preconditions for solving $AGREE^F$ using $FINVAL^{F'}$; we just need to consider the formulae $\phi_{INDIST(S,db)} \rightarrow \psi$ and $\phi_{INDIST(S,db)} \rightarrow \neg\psi$ instead of the formulae $\phi_{INDIST(S,db)} \wedge \psi$ and $\phi_{INDIST(S,db)} \wedge \neg\psi$.

Before proving the decidability of $AGREE^{ERC}$, we introduce some notation. Given a policy $S = \langle ROW, COL\rangle$, a state db, a constraint $\langle q, \phi\rangle \in ROW$, and a tuple $\overline{t} \in Auths_{,q}(db)$, we denote by $mask_{S,db,q}(\overline{t})$ the masked tuple $\overline{v}$ obtained from $\overline{t}$ by replacing all the undisclosed values with †, i.e., $|\overline{v}| = |\overline{t}|$ and for all $i \in \{1, \ldots, |\overline{t}|\}$, if $Disc_S(\overline{t}, q, db, i)$, then $\overline{v}(i) = \overline{t}(i)$, otherwise $\overline{v}(i) = \dagger$.

Let D be a database schema, $S = \langle ROW, COL\rangle$ be a security policy over D, $\langle q, \phi\rangle$ be a constraint in ROW, and db be a database state in Ω_D. The set $Ind_{S,q}(db)$ contains all the masked tuples $\overline{t}$ that can be obtained from tuples in $Auth_{S,q}(db)$ using the function $mask_{S,db,q}(\overline{t})$, i.e., $Ind_{S,q}(db) := \{mask_{S,db,q}(\overline{t}) \mid \overline{t} \in Auth_{S,q}(db)\}$. Given a tuple $\overline{t} \in Ind_{S,q}(db)$, we denote by $card_{S,db,q}(\overline{t})$ the number of tuples $\overline{t}' \in Auth_{S,q}(db)$ that cannot be distinguished from $\overline{t}$ according to the security policy S, i.e., $card_{S,db,q}(\overline{t}) := |\{\overline{t}' \in Auth_{S,q}(db) \mid mask_{S,db,q}(\overline{t}') = \overline{t}\}|$.

Let $S = \langle ROW, COL\rangle$ be a security policy, $i \in \mathbb{N}$, and $q = \{\overline{x} | \phi\}$ be a non-boolean query. We denote by ψ^S_q the condition in the constraint associated with q, i.e., $\langle q, \psi^S_q\rangle \in ROW$. Similarly, we denote by $\psi^S_{q,i}$ the condition associated with the i-th value of q, i.e., $\langle q, \psi^S_{q,i}, i\rangle \in COL$. If there is no such a constraint, then $\psi^S_q = \bot$ (respectively $\psi^S_{q,i} = \bot$).

We now use Lemma 3.1 to prove that $AGREE^{ERC}$ is decidable. Due to the limited expressiveness of the ERC fragment, we cannot encode the indistinguishability relation in it. However, we can prove the decidability of $AGREE^{ERC}$ using the more expressive $BSRRC$ fragment. The full proof of this result is given in Appendix A.

Theorem 3.3. *$AGREE^{ERC}$ is decidable.*

Proof Sketch. Let $S = \langle ROW, COL\rangle$ be an ERC-policy, db be a state, and $q = \{\overline{x} \mid \psi\}$ be a non-boolean ERC-query such that there is a constraint for q in S. The formula $\theta_{q,S}(\overline{y}, \overline{t})$, where $\overline{y}$ is a tuple of variables and $\overline{t}$ is a possibly masked tuple of values in **dom** such that $|\overline{y}| = |\overline{t}|$, is as follows:

$$\theta_{\{\overline{x}|\psi\},S}(\overline{y}, \overline{t}) := \bigwedge_{\substack{i\in\{1,\ldots,|\overline{t}|\}\\ \wedge\overline{t}(i)\neq\dagger}} (\psi^S_{q,i}[\overline{x} \mapsto \overline{y}] \wedge \overline{y}(i) = \overline{t}(i)) \wedge \bigwedge_{\substack{i\in\{1,\ldots,|\overline{t}|\}\\ \wedge\overline{t}(i)=\dagger}} \neg\psi^S_{q,i}[\overline{x} \mapsto \overline{y}].$$

The formula $\theta_{\{\overline{x}|\psi\},S}(\overline{y}, \overline{t})$ forces the masked version of the tuple associated with the values of the variables $\overline{y}$ to be $\overline{t}$.

In our encoding $\phi'_{INDIST(S,db)}$, we use counting quantifiers. A counting quantifier $\exists^{op\ m}\overline{x}.\,\phi(\overline{x})$, where $op \in \{=, \leq, \geq, <, >\}$ and $m \in \mathbb{N}$, is a quantifier that specifies the number of tuples that can be

mapped to $\overline{x}$ and satisfy $\phi(\overline{x})$. For instance, the formula $\exists^{=2}x_1.\,\phi(x_1)$ is true iff there are exactly two distinct elements x_1 satisfying $\phi(x_1)$. Note that counting quantifiers do not add expressive power to RC.

Given a tuple of variables $\overline{x}$, we denote by $\overline{y}_{\overline{x}}$ the tuple of variables $y_1, \ldots, y_{|\overline{x}|}$. The encoding $\phi'_{INDIST(S,db)}$ is:

$$\psi_{S,\{\overline{x}|\psi\},db} := \exists^{=|Auth_{S,\{\overline{x}|\psi\}}(db)|}\overline{y}_{\overline{x}}.\ (\psi[\overline{x} \mapsto \overline{y}_{\overline{x}}] \wedge \psi^S_{\{\overline{x}|\psi\}}[\overline{x} \mapsto \overline{y}_{\overline{x}}])$$

$$\gamma_{\{\overline{x}|\psi\},S,db,\overline{t}} := \exists^{\geq card_{S,db,\{\overline{x}|\psi\}}(\overline{t})}\overline{y}_{\overline{x}}.\ (\psi[\overline{x} \mapsto \overline{y}_{\overline{x}}] \wedge \psi^S_{\{\overline{x}|\psi\}}[\overline{x} \mapsto \overline{y}_{\overline{x}}] \wedge \theta_{\{\overline{x}|\psi\},S}(\overline{y}_{\overline{x}}, \overline{t}))$$

$$\phi'_{INDIST(S,db)} := \bigwedge_{\langle q,\phi\rangle \in ROW} \left(\psi_{S,q,db} \wedge \bigwedge_{\overline{t} \in Ind_{S,q}(db)} \gamma_{q,S,db,\overline{t}} \right).$$

The sentence $\psi_{S,\{\overline{x}|\psi\},db}$ states that, for any state db' such that $[\psi_{S,\{\overline{x}|\psi\},db}]^{db'} = \top$, the sets $Auth_{S,\{\overline{x}|\psi\}}(db')$ and $Auth_{S,\{\overline{x}|\psi\}}(db)$ have the same cardinality. The sentence $\gamma_{\{\overline{x}|\psi\},S,db,\overline{t}}$ states that, for any state db' such that $[\gamma_{\{\overline{x}|\psi\},S,db,\overline{t}}]^{db'} = \top$, there are at least $card_{S,db,\{\overline{x}|\psi\}}(\overline{t})$ tuples $\overline{v}$ in $Auth_{S,\{\overline{x}|\psi\}}(db')$ such that $mask_{S,s',\{\overline{x}|\psi\}}(\overline{v}) = \overline{t}$. The sentence $\phi'_{INDIST(S,db)}$ encodes the indistinguishability relation for the ERC-policy S and the state db.

Although $\phi'_{INDIST(S,db)}$ is not in $BSRRC$, it can be rewritten as a $BSRRC$-sentence. Furthermore, there are $BSRRC$-sentences equivalent to $\phi'_{INDIST(S,db)} \wedge \psi$ and $\phi'_{INDIST(S,db)} \wedge \neg\psi$ for any ERC-sentence ψ and any ERC-policy S. Therefore, $AGREE^{RC}$ is decidable by Lemma 3.1. □

From Theorem 3.3 and the fact that conjunctive queries (CRC) are a strict subset of ERC, it also follows that $AGREE^{CRC}$ is decidable. Therefore, there are optimal SAQP algorithms for boolean conjunctive queries.

3.5.4 Truman and Non-Truman models

In this section, we study the connections between Truman and Non-Truman models as defined in [131]. In the Non-Truman model, the security policy is expressed using a set of authorization views. Authorization views are standard database views extended with parameters referring to users' credentials. They can be used to specify the data a user is authorized to read. Authorization views in [131] are expressed in SQL, whereas here we study authorization views in the relational calculus. In the following, we ignore users' credentials, which does not limit our results.

Definition 3.13. Let D be a database schema and F be a query language. An *F-authorization view* is defined by assigning a relation identifier V not in D to a non-boolean F-query $\{\overline{x} \mid \phi(\overline{x})\}$, i.e., $V = \{\overline{x} \mid \phi(\overline{x})\}$. □

Let $s \in \Omega_D$ be a state and $V = \{\overline{x} \mid \phi(\overline{x})\}$ be a view. The *materialization* of V in db, denoted by $db(V)$, is $[\{\overline{x} \mid \phi(\overline{x})\}]^{db}$. Views can be used in RC-formulae in the same way as relation schemas, but in this case we consider the materialized views instead of the relation instances.

We now introduce the notion of equivalence with respect to a set of authorization views AV (AV-equivalence) defined in [131]. Two states are *AV-equivalent* iff their views' materializations are the same.

Definition 3.14. Let D be a database schema, AV be a set of authorization views over D, and $db, db' \in \Omega_D$ be two states. Then, db and db' are *AV-equivalent*, written $db \cong_{AV} db'$, iff for all $V \in AV$, $db(V) = db'(V)$. □

For any database schema D and any set of views AV, the relation $\cong_{AV}$ is an equivalence relation over Ω_D.

We now introduce *row-level policies*. Intuitively, a row-level policy does not disclose partial tuples.

Definition 3.15. Let D be a database schema and $S = \langle ROW, COL\rangle$ be a security policy over D. S is a *row-level policy* iff for all $\langle q, \phi\rangle \in ROW$, all $db \in \Omega_D$, and all $\overline{t} \in [q]^{db}$, if $Disc_S(\overline{t}, q, db)$, then $Disc_S(\overline{t}, q, db, i)$ for all $i \in \{1, \ldots, |\overline{t}|\}$. □

We say that a security policy S and a set of views AV are *equivalent* iff $\cong_{AV} = \cong_S$. We prove below that RC-authorization views are as expressive as row-level RC-policies. It is easy to see that row-level RC-policies are strictly less expressive than RC-policies, and therefore RC-authorization views are strictly less expressive than RC-policies.

Proposition 3.2. *Let D be a database schema. For each set of RC-authorization views over D, there is an equivalent row-level RC-security policy over D and vice versa.*

Proof. **(⇒).** Let AV be a set of RC-authorization views. We now show how to construct the row-level security policy S. The security policy $S = \langle ROW, COL\rangle$ is as follows:

- For each view $V = \{\overline{x} \mid \phi\}$ in AV, we define a constraint $\langle\{\overline{x} \mid \phi\}, \top\rangle \in ROW$.
- For each view $V = \{\overline{x} \mid \phi\}$ in AV and each $i \in \{1, \ldots, |\overline{x}|\}$, we define a constraint $\langle\{\overline{x} \mid \phi\}, \top, i\rangle \in ROW$.

It is easy to see that S is equivalent to AV, i.e., any two states AV-equivalent are also indistinguishable according to S and vice versa. Indeed, the data disclosed by S are exactly the materialization of the views in AV. It is also easy to check that S is row-level by construction. Finally, observe that if AV is a set of F-authorization views for some query language F, then S is an F-security policy.

(⇐). Let $S = \langle ROW, COL\rangle$ be a row-level RC-security policy. We can define a set of authorization views AV as follows. For each $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, we define the authorization view $V_{\{\overline{x}|\psi\}}$ as follows:

$$V_{\{\overline{x}|\psi\}} = \{\overline{x} \mid \psi \wedge \phi\}.$$

Given a state db and a constraint $\langle q, \phi\rangle \in ROW$, the set $Auth_{S,q}(db)$ is exactly the materialization of the view V_q in the state db. Note also that given a row-level policy S, then $Auth_{S,q}(db) = Ind_{S,q}(db)$ for all queries q and all states db. We now prove that S is equivalent to AV. Let db_1 and db_2 be two indistinguishable states according to S. Since the views in AV disclose only values that we are authorized to read according to S, then db_1 and db_2 are AV-equivalent. Similarly, let db_1 and db_2 be two AV-equivalent states. Given a constraint $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$ and a state db, then the set $Auth_{S,\{\overline{x}|\psi\}}(s)$ is exactly the materialization of the view $V_{\{\overline{x}|\psi\}}$ in the state db. Therefore, since db_1 and db_2 are AV-equivalent, i.e., the materialization of the views in AV are the same, then they are also indistinguishable. Note that if F is closed under conjunction, then given any F-policy, the resulting equivalent set of authorization views is in F. □

Example 3.4. Let AV be the set of views $\{q_1, q_2\}$, where q_1 is the query $\{z \mid \exists x, y.\, Employee(x, y, z)\}$ and q_2 is the query $\{y \mid \exists x, z.\, Employee(x, y, z)\}$. The equivalent row-level policy is $S = \langle\{\langle q_1, \top\rangle, \langle q_2, \top\rangle\}, \{\langle q_1, \top, 1\rangle, \langle q_2, \top, 1\rangle\}\rangle$.

Let S be the row-level policy $\langle\{\langle q_1, \phi\rangle, \langle q_2, \psi\rangle\}, \{\langle q_1, \top, 1\rangle, \langle q_2, \top, 1\rangle\}\rangle$, where ϕ and ψ are RC-formulae. The equivalent set of views is $\{q_1', q_2'\}$, where $q_1' = \{z \mid \exists x, y.\, Employee(x, y, z) \wedge \phi\}$ and $q_2' = \{y \mid \exists x, z.\, Employee(x, y, z) \wedge \psi\}$. ■

From Proposition 3.2, it follows that, given a set of authorization views AV, two states are AV-equivalent iff they are indistinguishable according to the equivalent security policy and vice versa.

In the relational calculus, authorization views are strictly less expressive than security policies. This is no longer the case for sufficiently powerful query languages. For instance, in the relational calculus extended with the **count** aggregation operator, authorization views are as expressive as security policies, as shown in Example 3.5. This also implies that, in contrast to the case of RC, for sufficiently powerful languages, row-level policies are as expressive as policies that combine both row-level and column-level constraints.

Example 3.5. In this example, we use the aggregation operator **count** [10]. Informally, $\mathbf{count}[\overline{x} \mid \psi(\overline{x})]$ returns the number of tuples in the result of the query $\{\overline{x} \mid \psi(\overline{x})\}$.

Let q be the query $\{x, z \mid \alpha\}$, where α is the RC-formula $\exists y.\, Employee(x, y, z)$, and let S be the policy $\langle\{\langle q, \phi\rangle\}, \{\langle q, \psi, 1\rangle, \langle q, \gamma, 2\rangle\}\rangle$, where ϕ, ψ, and γ are RC-formulae. The set of authorization views AV that is equivalent to S is:

$$\begin{aligned}
&\{x, z, s \mid \alpha \wedge \phi \wedge \psi \wedge \gamma \wedge s = 1\},\\
&\quad\{x, s \mid \exists z.\, (\alpha \wedge \phi \wedge \psi \wedge \neg\gamma) \wedge s = \mathbf{count}[z|\alpha \wedge \phi \wedge \psi \wedge \neg\gamma]\},\\
&\quad\{z, s \mid \exists x.\, (\alpha \wedge \phi \wedge \gamma \wedge \neg\psi) \wedge s = \mathbf{count}[x|\alpha \wedge \phi \wedge \gamma \wedge \neg\psi]\},\\
&\qquad\{s \mid \exists x, z.\, (\alpha \wedge \phi \wedge \neg\psi \wedge \neg\gamma) \wedge s = \mathbf{count}[x, z|\alpha \wedge \phi \wedge \neg\psi \wedge \neg\gamma]\}.
\end{aligned}$$

Let db be a state. The materialization of the views in AV in the state db discloses, for each $\bar{t} \in Ind_{S,q}(db)$, the values in $\bar{t}$ different from † and the value of $card_{S,db,q}(\bar{t})$, which is stored in the variable s. Therefore, $\cong_{AV} = \cong_S$. ■

The intuition behind Example 3.5 is formalized in Proposition 3.3. Note that we denote by RC^+ the relational calculus extended with the **count** aggregation operator [10].

Proposition 3.3. *Let D be a database schema. For each set of RC^+-authorization views over D, there is an equivalent RC^+-security policy over D, and vice versa.*

Proof. **(⇒).** The proof in this direction is exactly the same as the proof (in the same direction) of Proposition 3.2.

(⇐). Let $S = \langle ROW, COL\rangle$ be a RC^+-security policy. We define a set of RC^+-authorization views AV as follows. We denote by Q_S the set containing all the non-boolean queries on which there are constraints in ROW. Given a tuple of variables $\overline{x}$ and a finite set of variables A such that all $y \in A$ occur in $\overline{x}$, we denote by $var_{\overline{x}}(A)$ the tuple of variables $\overline{y}$ that contains exactly the variables in A ordered according to their position in $\overline{x}$. We also denote by $rem_{\overline{x}}(A)$ the result of $var_{\overline{x}}(\overline{x} \setminus A)$. For each $q = \{\overline{x} \mid \psi\}$ in Q_S, and each set $A \in 2^{free(\psi)}$, we define the authorization view $V_{\{\overline{x}|\psi\},A}$:

$$V_{q,A} := \{var_{\overline{x}}(A), s \mid \exists rem_{\overline{x}}(A).\ (\psi \wedge \psi_q^S \wedge \bigwedge_{y\in A} \psi^S_{q,pos(\overline{x},y)} \wedge \bigwedge_{y\in free(\psi)\setminus A} \neg\psi^S_{q,pos(\overline{x},y)}) \wedge$$
$$s = \mathbf{count}[rem_{\overline{x}}(A) \mid \psi \wedge \psi_q^S \wedge \bigwedge_{y\in A} \psi^S_{q,pos(\overline{x},y)} \wedge \bigwedge_{y\in free(\psi)\setminus A} \neg\psi^S_{q,pos(\overline{x},y)}]\}.$$

Note that if $|rem_{\overline{x}}(A)| = 0$, then we replace the **count** operator with 1, i.e., $s = 1$ in this case. Note that we assume, without loss of generality, that $s \notin free(\psi)$.

Given a constraint $\langle\{\overline{x} \mid \psi\}, \phi\rangle \in ROW$, a state db, and a set of variables $A \subseteq free(\psi)$, the materialization of the view $V_{\{\overline{x}|\psi\},A}$ in the state db contains a tuple $\overline{v} \circ \langle m\rangle$, where $\circ$ indicates the concatenation between two tuples and $m \in \mathbb{N}$, iff there is a tuple $\bar{t} \in Ind_{S,q}(db)$ such that $card_{S,db,q}(\bar{t}) = m$ and for all $y \in free(\psi)$, if $y \in A$, then $\overline{v}(pos(var_{\overline{x}}(A), y)) = \bar{t}(pos(\overline{x}, y))$ and otherwise $\bar{t}(pos(\overline{x}, y)) = \dagger$. From this, it follows that S is equivalent to AV. Indeed, let db_1 and db_2 be two indistinguishable states according to S. For all $\langle q, \phi\rangle \in ROW$, $Ind_{S,q}(db_1) = Ind_{S,q}(db_2)$ and $card_{S,db_1,q}(\bar{t}) = card_{S,db_2,q}(\bar{t})$ for all $\bar{t} \in Ind_{S,q}(db_1)$. Therefore, the materializations of the views in AV are the same in both states, and thus $db_1 \cong_{AV} db_2$. Similarly, let db_1 and db_2 be two AV-equivalent states. Then, for all $\langle q, \phi\rangle \in ROW$, we have that $Ind_{S,q}(db_1) = Ind_{S,q}(db_2)$ and also that $card_{S,db_1,q}(\bar{t}) = card_{S,db_2,q}(\bar{t})$ for all $\bar{t} \in Ind_{S,q}(db_1)$. Therefore, $db_1 \cong_S db_2$. □

We now introduce the concept of conditional validity [131], which is different from validity in FOL and RC. Afterwards, we define optimal algorithms in the Non-Truman model.

Definition 3.16. Let D be a database schema, $db \in \Omega_D$ be a state, and AV be a set of authorization views over D. An RC-query q is *conditionally valid* in db iff there is another RC-query q' written only in terms of the views in AV that is equivalent to q on all states db' such that $db \cong_{AV} db'$. □

Definition 3.17. Let F be a query language. An F-SAQP $\mathcal{M}$ is *optimal in the Non-Truman model* iff it executes exactly the conditionally valid queries. □

Before reconciling Truman and Non-Truman models, we state the following lemma.

Lemma 3.2. *Let D be a database schema, F be a query language, AV be a set of F-authorization views, $db \in \Omega_D$ be a state, and ϕ be a boolean F-query. Moreover, let S be the F-security policy equivalent to AV. The query ϕ is conditionally valid in the state db with respect to AV iff $AGREE^F(D, S, \phi, db) = \top$.*

Proof. **(⇒).** Let ϕ be a query that is conditionally valid in the state db with respect to AV. Since ϕ is conditionally valid, then there is another query ϕ' expressed only in terms of AV such that $[\phi]^{db'} = [\phi']^{db'}$ for all states db' AV-equivalent to db. Note that two states are AV-equivalent iff they have the same views' materializations. Since ϕ' is expressed only in terms of AV, it is domain independent, the query language is deterministic, and the views' materializations are fixed, then ϕ' returns always the same result $k \in \{\top, \bot\}$ on all states db' AV-equivalent to db. Therefore, $[\phi]^{db'} = [\phi']^{db'} = k$ for all states db' such that $db \cong_{AV} db'$. Since AV and S are equivalent, it follows that $db \cong_{AV} db'$ iff $db \cong_S db'$. It follows that $[\phi]^{db} = k$ for all states db' such that $db \cong_S db'$, and therefore $AGREE^F(D, S, \phi, db) = \top$.

(⇐). Let ϕ be a query such that $AGREE^F(D, S, \phi, db) = \top$. Moreover, let $k \in \{\top, \bot\}$ be the value $[\phi]^{db}$. Let S be the F-security policy equivalent to AV. Since $AGREE^F(D, S, \phi, db) = \top$, we know that the result of ϕ is k for all states indistinguishable from db according to S. Since S and AV are equivalent, it follows that the result of ϕ is k for all states AV-equivalent to db. Therefore, the query $\phi' = k$ is equivalent to ϕ for all states AV-equivalent to db. Thus, ϕ is conditionally valid in db according to AV. □

Finally, we can connect optimal SAQP in the Truman and Non-Truman model.

Theorem 3.4. *Let F be a query language. An optimal F-SAQP for boolean queries in the Truman model is an optimal F-SAQP for boolean queries in the Non-Truman model when $\dagger$ is interpreted as rejecting the query.*

Proof. The theorem follows from Theorem 3.1, Proposition 3.2, Lemma 3.2, and the definition of optimal SAQP in the Non-Truman model. □

In the past, Truman and Non-Truman models have been presented as two distinct and alternative approaches: the former concerned with returning as much information as possible and the latter concerned with avoiding inconsistencies. For boolean *RC*-queries, Theorem 3.4 shows that an optimal SAQP in the Truman model can be used as an optimal SAQP in the Non-Truman model. The reason is that, for boolean queries, the only way to protect sensitive information is to return † and there is no way to return partial results. Therefore, for boolean *RC*-queries, the Non-Truman model is a special case of the Truman model. Indeed, *RC*-authorization views are strictly less expressive than *RC*-policies. For sufficiently powerful query languages, such as the relational calculus extended with the **count** operator, the two models coincide for boolean queries because authorization views are as expressive as security policies.

The practical consequence of Theorem 3.4 is that we can straightforwardly use an optimal algorithm in the Truman model to compute the answer to a query in the Non-Truman model. It also follows that the results in Sections 3.5.2 and 3.5.3 apply to the Non-Truman model and to the conditional validity problem.

3.6 Non-boolean queries

In this section, we study the existence of optimal SAQP algorithms for non-boolean queries and the connections between optimal SAQP algorithms in the Truman and Non-Truman models.

3.6.1 Correctness Criteria

The answer to a non-boolean query under a given security policy is a multi-set of masked tuples and not a set of tuples as is standard in database theory. Indeed, some values may be set to † because we are not authorized to read them. However, not all multi-sets of tuples are valid results for SAQP algorithms. We are interested only in those multi-sets that can be obtained from a set of unmasked tuples by replacing values with †. The set M is the set of all multi-sets of tuples V such that there is a set of unmasked tuples T and a bijection f from V to T such that $\bar{t} \sqsubseteq f(\bar{t})$ for all $\bar{t} \in V$. In the following, we always refer just to multi-sets in M.

Following [165], we define a subsumption relation $\preceq$ on multi-sets, which is a partial order on M. A multi-set $K \in M$ is subsumed by another multi-set $K' \in M$, written $K \preceq K'$, iff there is an injective mapping $f : K \to K'$ such that for all $\bar{t} \in K$, $\bar{t} \sqsubseteq f(\bar{t})$.

Example 3.6. Let $\bar{t}$ and $\bar{z}$ be the tuples $\langle John, 25, Clerk\rangle$ and $\langle Frank, \dagger, \dagger\rangle$ respectively. The multi-set $\{\bar{t}, \bar{t}, \bar{z}\}$ is not in M because there are two occurrences of the unmasked tuple $\bar{t}$. In contrast, the multi-set $J = \{\bar{z}, \bar{z}, \bar{t}\}$ is in M. Let T be the set $\{\langle Frank, 46, Clerk\rangle, \langle Frank, 27, SecretAgent\rangle, \langle John, 25, Clerk\rangle\}$. Then, $J \preceq T$. ■

We now introduce the correctness criteria for non-boolean queries. Security is the same as in Definition 3.8 and we state here only soundness and maximality. Wang et al. [165] studied only secure and sound algorithms and considered only algorithms that return a unique multi-set of masked tuples as a query's result. However, there are cases in which the optimal answer is not unique, i.e., there are finitely many different multi-sets of masked tuples that are all optimal and incomparable. In the following, we consider algorithms that might return as a query's result a set of multi-sets. Therefore, for non-boolean queries, the set U, which contains all possible results of optimal SAQPs, is 2^M. For this reason, we must modify the criteria given in [165].

For non-boolean queries, a sound algorithm must return a result subsumed by the original query's result.

Definition 3.18. Let D be a database schema and F be a query language. An F-security-aware query processor $\mathcal{M}$ is *sound* iff for all non-boolean queries $q \in F$, all F-policies S over D, all $db \in \Omega_D$, and all $V \in \mathcal{M}(q, S, db)$, $V \preceq [q]^{db}$. □

A maximal algorithm must return a result that subsumes any multi-set $T \in M$ that is subsumed by the original query's result in all indistinguishable states.

Definition 3.19. Let D be a database schema and F be a query language. An F-security-aware query processor $\mathcal{M}$ is *maximal* iff for all F-policies S over D, all non-boolean queries $q \in F$, all $db \in \Omega_D$, and all $T \in M$, if $T \preceq [q]^{db'}$ for all $db' \in [\![db]\!]_{\cong_S}$, then there is a $V \in \mathcal{M}(q, S, db)$ such that $T \preceq V$. □

We now define optimal algorithms for non-boolean queries in the Truman model.

Definition 3.20. Let F be a query language. An F-SAQP for non-boolean queries is *optimal in the Truman model* iff it satisfies the Definitions 3.8, 3.18, and 3.19. □

$$\mathcal{M}_{opt}(q, S, db) = \{V \in M \mid \forall db' \in [\![db]\!]_{\cong_S}.\ (V \preceq [q]^{db'})\}$$

FIGURE 3.3: An optimal SAQP for non-boolean queries

Figure 3.3 describes an optimal SAQP for non-boolean queries. Depending on the query language, this function may not be computable.

Example 3.7. Let D be a database schema with only one relation schema R of arity 2. For simplicity, we call a and b the two attributes in R, and we assume that **dom** is $\mathbb{N}$. Moreover, let $\phi(n, m)$ be the formula $x = n \wedge y = m \wedge \forall y.\,(R(n, y) \rightarrow y = m) \wedge \forall x.\,(R(x, m) \rightarrow x = n)$. Let q be the query $\{x, y \mid R(x, y)\}$ and S be the security policy $\langle\{\langle q, \top\rangle\}, \{\langle q, \psi, 1\rangle, \langle q, \psi, 2\rangle\}\rangle$, where ψ is the formula $\neg\phi(1, 3) \wedge \neg\phi(1, 4) \wedge \neg\phi(2, 3) \wedge \neg\phi(2, 4)$. One of the equivalence classes defined by S contains exactly the two states db and db' such that $db(R) = \{\langle 1, 3\rangle, \langle 2, 4\rangle\}$ and $db'(R) = \{\langle 1, 4\rangle, \langle 2, 3\rangle\}$. In this case, $\mathcal{M}_{opt}(q, S, db) = \mathcal{M}_{opt}(q, S, db') = K$ and $K = \{V \in M \mid V \preceq \{\langle 1, \dagger\rangle, \langle 2, \dagger\rangle\} \vee V \preceq \{\langle \dagger, 3\rangle, \langle \dagger, 4\rangle\}\}$. Therefore, in db and db', we are authorized to read the values of the attributes a and b separately, but not together. ■

In the Truman model, boolean queries are not a special case of non-boolean queries. For fragments that are not closed under negation, we cannot use an optimal SAQP $\mathcal{M}$ for non-boolean queries to distinguish whether a sentence returns $\bot$ or $\dagger$ because $\mathcal{M}$ returns $\{\emptyset\}$ in both cases.

We now define a decision problem, denoted as $SUBSUME^F$, that will be used in a way similar to $AGREE^F$.

Definition 3.21. Given a query language F, $SUBSUME^F$ denotes the problem:
Input: A database schema D, an F-security policy S, a non-boolean F-query $q = \{\overline{x} \mid \phi(\overline{x})\}$, a multi-set $T \in M$ such that $|\bar{t}| = |\overline{x}|$ for all $\bar{t} \in T$, and a state $db \in \Omega_D$.
Question: For all states $db' \in [\![db]\!]_{\cong_S}$, is $T \preceq [q]^{db'}$? □

The $SUBSUME^F$ decision problem is related to the existence of optimal SAQP algorithms.

Theorem 3.5. *Let F be a query language. There is a computable optimal F-SAQP algorithm $\mathcal{M}$ for non-boolean queries iff $SUBSUME^F$ is decidable.*

Proof. **(⇒).** Let $\mathcal{M}$ be a computable optimal F-SAQP algorithm for non-boolean queries. We use $\mathcal{M}$ as a subroutine in a decision procedure for $SUBSUME^F$. Let D, S, q, T, and db be the inputs of the $SUBSUME^F$ problem. Then, $SUBSUME^F(D, S, q, T, db) = \top$ iff there is a $V \in \mathcal{M}(q, S, db)$ such that $T \preceq V$.

(⇐). We use $SUBSUME^F$ to build an optimal F-SAQP $\mathcal{M}$. Let D be a database schema, S be an F-security policy over D, $db \in \Omega_D$ be a state, and q be a non-boolean F-query. $\mathcal{M}(q, S, db)$ is $\{T \in M \mid T \preceq [q]^{db} \wedge SUBSUME^F(D, S, q, T, db) = \top\}$. It is easy to see that $\mathcal{M}$ is optimal and computable. □

The $SUBSUME^F$ decision problem plays the same role for optimal algorithms for non-boolean queries as $AGREE^F$ does for boolean queries. We now analyze the decidability of $SUBSUME^F$.

3.6.2 Impossibility Results

In this section, we extend the impossibility results in Section 3.5.2 from boolean RC-queries to non-boolean RC-queries.

Theorem 3.6. *$SUBSUME^{RC}$ is undecidable.*

Proof. Let D be a database schema. We define a new schema D' obtained from D by adding a new relation schema T with arity 1. Let v be a value in **dom**, db be the state in $\Omega_{D'}$ such that $db(T) = \{\langle v\rangle\}$ and $db(R) = \emptyset$ for all relation schemas R in D, and S be the RC-policy $\langle\{\langle\{x \mid T(x)\}, \top\rangle\}, \{\langle\{x \mid T(x)\}, \top, 1\rangle\}\rangle$. Then, an RC-sentence ϕ over D is finitely valid iff $SUBSUME^{RC}(D', S, \{x \mid T(x) \wedge \phi\}, \{\langle v\rangle\}, db) = \top$. To show the correctness of this reduction, consider that the equivalence class $[\![db]\!]_{\cong_S}$ defined by the policy S contains a database state db' for each database state $db'' \in \Omega_D$ such that db' and db'' agree on all relation schemas in S, i.e., $[\![db]\!]_{\cong_S} = \{db' \in \Omega_{D'} \mid \exists db'' \in \Omega_D.\ db'(T) = \{\langle v\rangle\} \wedge \bigwedge_{R \in D} db'(R) = db''(R)\}$. Then, $SUBSUME^{RC}(D', S, \{x \mid T(x) \wedge \phi\}, \{\langle v\rangle\}, db) = \top$ iff ϕ holds in all database states in $[\![db]\!]_{\cong_S}$, i.e., iff ϕ is finitely valid. This reduction can be implemented by a total Turing machine. Therefore, since $FINVAL^{RC}$ is undecidable, so is $SUBSUME^{RC}$. □

From Theorem 3.6, it follows that, for non-boolean queries, there are no computable optimal SAQP algorithms in the Truman model for the relational calculus. Therefore, it is impossible to securely process queries without either violating the security policy, losing some information, or returning incorrect results.

Note that, following the proof of Theorem 3.6, we can prove an even stronger result: for any fragment F of RC such that (1) F is closed under conjunction, and (2) $FINVAL^F$ is undecidable, then $SUBSUME^F$ is also undecidable. Therefore, from well-known undecidability results for fragments of first-order logic, we can identify fragments of RC for which there are no optimal algorithms for non-boolean queries, such as the $BSRRC$ fragment.

3.6.3 Possibility Results

We now present a general criterion for identifying fragments of RC where $SUBSUME^F$ is decidable. We first introduce the notion of encoding the subsumption relation $\preceq$ in a formula. Let D be a database schema, $\psi(\overline{x})$ be a formula with free variables $\overline{x}$, and $T \in M$ be a multi-set such that $|\overline{t}| = |\overline{x}|$ for all $\overline{t} \in T$. We say that a sentence $\phi_{T,\psi(\overline{x})}$ *encodes the subsumption relation between* T *and* $\psi(\overline{x})$ iff $\phi_{T,\psi(\overline{x})}$ is domain independent and for all $db \in \Omega_D$, $[\phi_{T,\psi(\overline{x})}]^{db} = \top$ iff $T \preceq [\{\overline{x} | \psi(\overline{x})\}]^{db}$.

Lemma 3.3 presents sufficient conditions for the decidability of the $SUBSUME^F$ decision problem.

Lemma 3.3. *Let F be a query language. $SUBSUME^F$ is decidable if there is a query language F' such that:*

1. *$F \subseteq F'$,*
2. *$AGREE^{F'}$ is decidable for all F-policies, and*
3. *for any multi-set $T \in M$ and any F-formula $\psi(\overline{x})$ such that $|\overline{t}| = |\overline{x}|$ for all $\overline{t} \in T$, we can compute a sentence $\phi_{T,\psi(\overline{x})} \in F'$ that encodes the subsumption relation between T and $\psi(\overline{x})$.*

Proof. Let D be a database schema, $db \in \Omega_D$ be a state, $\{\overline{x} \mid \psi\}$ be an F-query, S be an F-policy, and $T \in M$ be a multi-set of tuples such that $|\overline{t}| = |\overline{x}|$ for all $\overline{t} \in T$. Let $\phi_{T,\psi}$ be the F'-sentence encoding the subsumption relation. Then, $SUBSUME^F(D, S, \{\overline{x} \mid \psi\}, T, db) = \top$ iff $[\phi_{T,\psi}]^{db} = \top$ and $AGREE^{F'}(D, S, \phi_{T,\psi}, db) = \top$. This reduction can be implemented by a Turing machine. The decidability of $SUBSUME^F$ directly follows from (1)–(3).

We now prove the correctness of the above reduction. If $[\phi_{T,\psi(\overline{x})}]^{db} = \top$ and $AGREE^{F'}(D, S, \phi_{T,\psi(\overline{x})}, db) = \top$, then for all states db' indistinguishable from db, we have that $T \preceq [\{\overline{x} \mid \psi(\overline{x})\}]^{db'}$. Hence, $SUBSUME^F(D, S, \{\overline{x} \mid \psi(\overline{x})\}, T, db) = \top$. If $[\phi_{T,\psi(\overline{x})}]^{s} = \bot$, then $T \not\preceq [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$ and thus $SUBSUME^F(D, S, \{\overline{x} \mid \psi(\overline{x})\}, T, db) = \bot$. Similarly, if $AGREE^{F'}(D, S, \phi_{T,\psi(\overline{x})}, db) = \bot$, then there is at least a state db' indistinguishable from db such that $T \not\preceq [\{\overline{x} \mid \psi(\overline{x})\}]^{db'}$. Therefore, also in this case $SUBSUME^F(D, S, \{\overline{x} \mid \psi(\overline{x})\}, T, db) = \bot$. □

We now use Theorem 3.3 and Lemma 3.3 to prove the decidability of $SUBSUME^{ERC}$. From Theorem 3.7, whose full proof is given in Appendix A, it follows that there are optimal SAQP algorithms for the existential fragment of the relational calculus.

Theorem 3.7. *$SUBSUME^{ERC}$ is decidable.*

Proof Sketch. Given a multi-set of tuples T in M and a tuple $\overline{t} \in T$, let $K_{\overline{t},T}$ be the multi-set $\{\overline{t}' \mid \overline{t}' \in T \wedge \overline{t} \sqsubseteq \overline{t}'\}$. The encoding is given by:

$$\phi_{T,\psi(\overline{x})} := \bigwedge_{\overline{t} \in T} \exists^{\geq |K_{\overline{t},T}|} \overline{x}. \left(\psi(\overline{x}) \wedge \bigwedge_{i \in \{1,\ldots,|\overline{x}|\} \wedge \overline{t}(i) \neq \dagger} \overline{x}(i) = \overline{t}(i)\right).$$

$\phi_{T,\psi(\overline{x})}$ can be equivalently rewritten as an ERC-sentence. □

Since ERC strictly contains conjunctive queries, also $SUBSUME^{CRC}$ is decidable.

3.6.4 Truman and Non-Truman models

We now study the connections between Truman and Non-Truman models for non-boolean queries. We first introduce the notion of a *strongly-optimal* security-aware query processor. Using strongly-optimal SAQPs, we generalize the Non-Truman model approach for non-boolean queries [131] to our security policies, which are more expressive than authorization views. Afterwards, we study the relationships between strongly-optimal SAQPs and optimal SAQPs in the Non-Truman model and extend our decidability results.

A strongly-optimal SAQP for non-boolean queries returns the original query's result whenever it is secure to do so, and otherwise returns †. Let L be the set of all finite sets of unmasked tuples. A security-aware query processor for non-boolean queries is called *strongly-optimal* if it satisfies the

$$\mathcal{M}_{s\text{-}opt}(q, S, db) = \begin{cases} [q]^{db} & \text{if } \forall db' \in [\![db]\!]_{\cong_S}. [q]^{db'} = [q]^{db} \\ \dagger & \text{otherwise} \end{cases}$$

FIGURE 3.4: A strongly-optimal SAQP for non-boolean queries

Definitions 3.8–3.10 and the set U, containing all the possible results, is $L \cup \{\dagger\}$. Figure 3.4 describes a strongly optimal SAQP for non-boolean queries. Depending on the query language, this function may not be computable.

Optimal SAQPs and strongly-optimal SAQPs are distinct. For instance, optimal SAQPs can return partial results while strongly-optimal SAQPs cannot. This holds even if we consider just row-level policies. Note too that, in the Non-Truman model, boolean queries are a special case of non-boolean queries.

We first define a new decision problem corresponding to $SUBSUME^F$.

Definition 3.22. Given a query language F, $EQUAL^F$ denotes the problem:
Input: A database schema D, an F-security policy S, a non-boolean F-query $q = \{\overline{x} \mid \phi(\overline{x})\}$, a set of tuples $T \in L$ such that $|\bar{t}| = |\overline{x}|$ for all $\bar{t} \in T$, and a state $db \in \Omega_D$.
Question: For all states $db' \in [\![db]\!]_{\cong_S}$, is $T = [q]^{db'}$? □

Similarly to Theorems 3.1 and 3.5, we can prove:

Theorem 3.8. *Let F be a query language. There is a computable strongly-optimal F-SAQP algorithm $\mathcal{M}$ for non-boolean queries iff $EQUAL^F$ is decidable.*

Proof. **(⇒).** Let $\mathcal{M}$ be a computable strongly-optimal F-SAQP $\mathcal{M}$ for non-boolean queries. We use $\mathcal{M}$ as a subroutine in a decision procedure for $EQUAL^F$, which takes as input a database schema D, a policy S, a set of tuples T, a non-boolean F-query q, and a database state db. Then, $EQUAL^F(D, S, q, T, db) = \top$ iff $T = \mathcal{M}(q, S, db)$.
(⇐). We use $EQUAL^F$ to construct a strongly-optimal F-SAQP $\mathcal{M}$. Let D be a database schema. $\mathcal{M}$ takes as input a security policy S over D, a non-boolean query $q \in F$, and a state $db \in \Omega_D$. We first compute the value $T = [q]^{db}$. If $EQUAL^F(D, S, q, T, db) = \top$, then $\mathcal{M}(q, S, db) = T$, otherwise $\mathcal{M}(q, S, db) = \dagger$. It is easy to see that $\mathcal{M}$ is strongly-optimal and computable. □

We can easily adapt the proof in Section 3.6.2 to obtain the following undecidability result:

Theorem 3.9. *$EQUAL^{RC}$ is undecidable.*

Proof. To prove the theorem, we reduce $AGREE^{RC}$ to $EQUAL^{RC}$. The undecidability of $EQUAL^{RC}$ then directly follows from Theorem 3.2.

Let D be a database schema, ϕ be a boolean formula, S be a security policy over D, and $db \in \Omega_D$ be a state. We define a database schema D' from D by adding a new relation schema T with arity 1. Let v be a value in **dom** and $db' \in \Omega_{D'}$ be the database state such that $db'(T) = \{\langle v \rangle\}$ and $db'(R) = db(R)$ for all relation schemas R in D. Finally, we extend the security policy $S = \langle ROW, COL \rangle$ on D to a new security policy $S' = \langle ROW', COL' \rangle$ on D', where $ROW' = ROW \cup \{\{x \mid T(x)\}, \top\rangle\}$ and $COL' = COL \cup \{\langle\{x \mid T(x)\}, \top, 1\rangle\}$. The equivalence class $[\![db']\!]_{\cong_{S'}}$ contains a state db_1 for each state $db_2 \in [\![db]\!]_{\cong_S}$ such that the relation instances in db_1 and db_2 are identical for all relation schema in D. Note that there is a bijection from the set $[\![db]\!]_{\cong_S}$ to the set $[\![db']\!]_{\cong_{S'}}$. We define the query q' as $\{x \mid T(x) \wedge \phi\}$. The set K' is $\{\langle v \rangle\}$, whereas $K'' = \emptyset$. Then, $AGREE^F(D, S, \phi, db) = \top$ iff $EQUAL^F(D', S', q', \{\langle v \rangle\}, db') = \top$ or $EQUAL^F(D', S', q', \emptyset, db') = \top$. The reduction described above can be implemented by a total Turing machine. We now prove its correctness. If $EQUAL^F(D', S', q', \{\langle v \rangle\}, db') = \top$, then $[\phi]^{db''} = \top$ for all $db'' \in [\![db]\!]_{\cong_S}$ and thus $AGREE^F(D, S, \phi, db) = \top$. Similarly, if $EQUAL^F(D', S', q', \emptyset, db') = \top$, then $[\phi]^{db''} = \bot$ for all $db'' \in [\![db]\!]_{\cong_S}$ and therefore $AGREE^F(D, S, \phi, db) = \top$. Otherwise, if $EQUAL^F(D', S', q', \{\langle v \rangle\}, db') = \bot$ and $EQUAL^F(D', S', q', \emptyset, db') = \bot$ then there are two states $db_1, db_2 \in [\![db]\!]_{\cong_S}$ such that $[\phi]^{db_1} \neq [\phi]^{db_2}$ and therefore $AGREE^F(D, S, \phi, db) = \bot$. □

Therefore, even strongly-optimal Security-Aware Query Processing is impossible for the relational calculus. Furthermore, $EQUAL^F$ is undecidable for any fragment F (closed under conjunction) where $FINVAL^F$, $FINSAT^F$, or $AGREE^F$ are undecidable.

We can also provide some decidability results. We first define a new encoding. Let D be a database schema, $\psi(\overline{x})$ be a formula with free variables $\overline{x}$, and $T \in L$ be a finite set of tuples such that $|\bar{t}| = |\overline{x}|$ for all $\bar{t} \in T$. We say that a sentence $\phi_{T,\psi(\overline{x})}$ *encodes the property* $\psi(\overline{x})$ *satisfied by the set* T iff $\phi_{T,\psi(\overline{x})}$ is domain independent and for all $db \in \Omega_D$, $[\phi_{T,\psi(\overline{x})}]^{db} = \top$ iff $T = [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$.

We can now give sufficient conditions for the decidability of the $EQUAL^F$ problem.

Lemma 3.4. *Let F be a query language. $EQUAL^F$ is decidable if there is a query language F' such that:*

1. *$F \subseteq F'$,*
2. *$AGREE^{F'}$ is decidable for all F-policies, and*
3. *for any finite set $T \in L$ and any formula $\psi(\overline{x}) \in F$ such that $|\overline{x}| = |\overline{t}|$ for all $\overline{t} \in T$, we can compute a sentence $\phi_{T,\psi(\overline{x})} \in F'$ that encodes the property $\psi(\overline{x})$ satisfied by the set T.*

Proof. Let D be a database schema, S be an F-security policy over D, $db \in \Omega_D$ be a database state, $q = \{\overline{x} \mid \psi(\overline{x})\}$ be an F-query, and $T \in L$ be a finite set of tuples such that $|\overline{x}| = |\overline{t}|$ for all $\overline{t} \in T$. From (3), it follows that we can compute the encoding $\phi_{T,\psi(\overline{x})}$. Then, $EQUAL^F(D, S, q, T, db) = \top$ iff $[\phi_{T,\psi(\overline{x})}]^{db} = \top$ and $AGREE^{F'}(D, S, \phi_{T,\psi(\overline{x})}, db) = \top$. The decidability of $EQUAL^F$ directly follows from (1) and (2).

We now show the correctness of our reduction. In case $[\phi_{T,\psi(\overline{x})}]^{db} = \top$ and $AGREE^{F'}(D, S, \phi_{T,\psi(\overline{x})}, db) = \top$, then for all states db' indistinguishable from db, we have that $T = [\{\overline{x} \mid \psi(\overline{x})\}]^{db'}$, and therefore $EQUAL^F(D, S, \{\overline{x} \mid \psi(\overline{x})\}, T, db) = \top$. If $[\phi_{T,\psi(\overline{x})}]^s = \bot$, then $T \neq [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$ and thus $EQUAL^F(D, S, \{\overline{x} \mid \psi(\overline{x})\}, T, db) = \bot$. Similarly, if $AGREE^{F'}(D, S, \phi_{T,\psi(\overline{x})}, db) = \bot$, then there is at least a state db' indistinguishable from db such that $T \neq [\{\overline{x} \mid \psi(\overline{x})\}]^{db'}$ and thus $EQUAL^F(D, S, \{\overline{x} \mid \psi(\overline{x})\}, T, db) = \bot$. □

Using Lemma 3.4, we derive the following result, whose proof is given in Appendix A.

Theorem 3.10. *$EQUAL^{ERC}$ is decidable.*

Similarly to Section 3.5.4, we have:

Lemma 3.5. *Let D be a database schema, F be a query language (that supports equality and is closed under conjunction and disjunction), AV be a set of F-authorization views, $db \in \Omega_D$ be a state, and q be a non-boolean F-query. Moreover, let S be the F-security policy equivalent to AV. The query q is conditionally valid in the state db with respect to AV iff $EQUAL^F(D, S, q, [q]^{db}, db) = \top$.*

Proof. **(⇒).** Let q be a query conditionally valid in the state db with respect to AV. From this, it follows that there is a query q' expressed only in terms of AV such that $[q]^{db'} = [q']^{db'}$ for all states db' AV-equivalent to db. Note that two states are AV-equivalent iff they have the same views' materializations. Since q' is expressed only in terms of AV, it is domain independent, the query language is deterministic, and the views' materializations are fixed, then q' returns always the same result K on all states db' AV-equivalent to db. Therefore, $[q]^{db'} = [q']^{db'} = K$ for all states db' such that $db \cong_{AV} db'$. Since AV and S are equivalent, it follows that $db \cong_{AV} db'$ iff $db \cong_S db'$. It follows that $[q]^{db'} = K$ for all states db' such that $db \cong_S db'$. Note that $[q]^{db} = K$, and therefore, $EQUAL^F(D, S, q, [q]^{db}, db) = \top$.

(⇐). Let $q = \{\overline{x} \mid \phi(\overline{x})\}$ be a query such that $EQUAL^F(D, S, q, [q]^{db}, db) = \top$ holds. Moreover, let $K = [q]^{db}$. Let S be the F-security policy equivalent to AV. Since $EQUAL^F(D, S, q, [q]^{db}, db) = \top$, the result of q is K for all states indistinguishable from db according to S. Since AV and S are equivalent, it follows that the result of q is K for all states AV-equivalent to db. We can now define the query q' as

$$\{y_1, \ldots, y_{|\overline{x}|} \mid \bigvee_{\overline{t} \in K} \bigwedge_{i \in \{1,\ldots,|\overline{x}|\}} y_i = \overline{t}(i)\}$$

Observe that (1) the result of q' for all states db' AV-equivalent to db is exactly K, and (2) q' does not depend on database tables. This means that q' is equivalent to q for all states AV-equivalent to db, and therefore q is conditionally valid in db according to AV. □

Theorem 3.11. *Let F be a query language. A strongly-optimal F-SAQP for non-boolean queries is an optimal F-SAQP for non-boolean queries in the Non-Truman model when † is interpreted as rejecting the query.*

Proof. The theorem follows from Proposition 3.2, Theorem 3.8, Lemma 3.5, and the definition of optimal SAQP in the Non-Truman model. □

For non-boolean queries, strongly-optimal algorithms and optimal algorithms are distinct. Therefore, from Theorem 3.11, we have:

Corollary 3.1. *There is a non-boolean RC-query q, a state db, a set AV of RC-views, and the equivalent row-level RC-policy S, such that the result of an optimal SAQP for non-boolean queries in the Non-Truman model is different from the result of an optimal SAQP for non-boolean queries in the Truman model.*

For boolean *RC*-queries, optimal algorithms in the Non-Truman model are a special case of optimal algorithms in the Truman model. Corollary 3.1 states that this result does not hold for non-boolean queries. In this case, the two models are distinct.

The reason for this difference is that in the Truman model we can return partial results, whereas in the Non-Truman model we either return the query's result or reject the query. Moreover, given the same inputs, the result of an optimal SAQP in one model does not provide any insights about the result of an optimal SAQP in the other model. For instance, if an optimal SAQP in the Non-Truman model rejects a query q, we do not know anything about the result in the Truman model. Similarly, if an optimal SAQP in the Truman model returns $\{\emptyset\}$ as a query's result, we do not know whether an optimal SAQP in the Non-Truman model accepts the query or rejects it. Hence, we cannot use optimal SAQPs in the Truman model as optimal SAQPs in the Non-Truman model and vice versa.

Theorem 3.11 shows that, for *RC*, optimal SAQPs in the Non-Truman model are a special case of strongly-optimal SAQPs. From this, it follows that Lemma 3.4 and Theorem 3.10 apply to optimal SAQPs in the Non-Truman model, and to the conditional validity problem. Observe too that, for sufficiently powerful query languages (such as the relational calculus extended with **count** operators), optimal SAQPs in the Non-Truman model and strongly-optimal SAQPs are equivalent modulo the interpretation of †.

3.7 Related Work

Security-Aware Query Processing. Security-Aware Query Processing algorithms are implemented in commercial databases [8, 41, 150]. Despite that, only limited work has been done on the theoretical aspects of this problem. In [131], Rizvi et al. proposed the notions of Truman and Non-Truman models. They provided inference rules, which are sound but not complete, for determining whether a query is conditionally valid. The undecidability of the unconditional validity problem follows from well-known results on query rewriting using views [124]. Zhang et al. [170] studied the conditional validity problem for conjunctive queries and showed that it is decidable. We improve these results in that we provide sufficient conditions for the decidability of conditional validity. We also show that this problem is decidable for the existential fragment of the relational calculus, which contains conjunctive queries, and for our security policies, which are more expressive than authorization views.

Wang et al. [165] were the first to propose correctness criteria for algorithms in the Truman model. They proposed a secure and sound SAQP algorithm. Other secure and sound algorithms have been proposed since then, such as [90, 145]. Our results prove the claim of Wang et al. that optimal Security-Aware Query Processing is difficult. We also prove that optimal SAQP in the Truman model is possible for the existential fragment of the relational calculus.

Instance-based Determinacy. The instance-based determinacy problem [107] consists of checking whether, given a database state *db*, a set of views V, and a query q, the materialization of the views in V in the state *db* fully determines the result of q in *db*. Koutris et al. [107] proved that the problem of instance-based determinacy for unions of conjunctive queries under the set semantics is decidable and is *coNP-complete* in terms of data complexity.

When restricted to row-level policies, which are equivalent to *RC*-views, the *AGREE* and *EQUAL* problems are equivalent to the instance-based determinacy problem for boolean and non-boolean queries respectively. In the unrestricted case, however, these equivalences do not hold because security policies are more expressive than *RC*-views and, therefore, *AGREE* and *EQUAL* are more general than instance-based determinacy under the set semantics. Indeed, masked tuples introduce a kind of bag semantics that cannot be captured using *RC*-views under the set semantics. We are not aware of any work exploring instance-based determinacy under the bag semantics.

Certain Answers. The problem of computing *certain answers* using views under the closed world assumption [9] shares similarities with optimal SAQP. But it also has important differences. For instance, the result of a boolean query ϕ according to optimal SAQP is one of $\{\top, \bot, \dagger\}$ whereas the certain answer is one of $\{\top, \bot\}$. Indeed, one must compute both the certain answer and the possible answer to compute the result of boolean optimal SAQP.

For non-boolean queries, optimal algorithms in the Truman model return a set of results, whereas the certain answer is unique. Moreover, while optimal SAQP in the Truman model considers masked tuples, the certain answer problem considers only unmasked tuples. This seriously limits Fine-Grained Access Control. For example, suppose we have a policy with a constraint on the i-th value of a query $q = \{\overline{x} \mid \psi\}$, for some $i \in \{1, \ldots, |\overline{x}|\}$. The certain answer of q will be $\emptyset$, whereas the result of an optimal SAQP will be a multi-set of masked tuples where the i-th value is replaced with †. In the Non-Truman model, an algorithm either returns the query's result or rejects the query. In contrast, the certain answer to a query is generally different from the query's result.

The main difference between optimal SAQP and the certain answer problem is that security policies are more expressive than views in the relational calculus. As we previously noted, the presence of masked tuples introduces a kind of bag semantics. However, while the problem of querying views has been studied extensively under the set semantics [9, 124], only limited work have been done under the bag-set and bag semantics [12]. Note that our work can be viewed in the general setting of querying views under the bag-set semantics. Despite that, we decided to keep the terminology of Security-Aware Query Processing for consistency with previous works on Fine-Grained Access Control in databases, e.g., [131, 165].

Another related problem is computing the certain answer to a query in an incomplete database [112]. The same considerations for certain answers using views apply to this case. Observe also that the notion of indistinguishability appears related to the notion of semantics of incomplete databases [112].

3.8 Conclusions

We presented the first analysis of optimal Security-Aware Query Processing. Our results show that (1) it is impossible to build optimal SAQP algorithms for Codd-complete query languages, such as SQL, and (2) this impossibility is not due to specific characteristics of these query languages but rather to the undecidability of the relational calculus and several of its fragments. Note that our results also do not depend on the particular characteristics of our security model. We showed that there are interesting fragments of RC for which optimal Security-Aware Query Processing is possible, such as the existential fragment of RC. Our results may be used to prove the decidability of optimal Security-Aware Query Processing for other fragments of RC, such as the monadic fragment and the guarded fragment, and other query languages.

For boolean queries, we showed that, for the relational calculus, optimal SAQP in the Non-Truman model is a special case of optimal SAQP in the Truman model, and that optimal algorithms in the two models coincide for the relational calculus extended with aggregation operators. In contrast, for non-boolean queries, optimal algorithms in the two models are distinct. This has direct consequences for developing algorithms for those models.

Optimal Security-Aware Query Processing is a difficult problem: it is intractable even for conjunctive queries. Indeed, it is *coNP-complete* in terms of data complexity for boolean conjunctive queries and row-level security policies [107]. Despite that, optimal SAQP can still have practical applications. For instance, there are fragments of conjunctive queries for which optimal SAQP in the Non-Truman model is in *PTIME* in terms of data complexity for row-level policies [107]. This suggests that there might be other fragments of conjunctive queries for which optimal SAQP is tractable.

Non-optimal SAQP algorithms can benefit from efficient optimal algorithms for some special cases. Namely, we can use tractable optimal algorithms when it is possible, and fall back to efficient non-optimal algorithms for the cases where optimal SAQP is undecidable or intractable. Furthermore, the study of optimal SAQP may shed some light on the trade-offs between efficiency and optimality, and can therefore lead to improvements for non-optimal algorithms.

Chapter 4

Securing Databases from Probabilistic Inference

Probability is expectation founded upon partial knowledge. A perfect acquaintance with all the circumstances affecting the occurrence of an event would change expectation into certainty, and leave nether room nor demand for a theory of probabilities.

George Boole

Databases can leak confidential information when users combine query results with probabilistic data dependencies, such as those found in genomics [97, 105, 109], social networks [92], and location tracking [117]. Attackers can exploit these dependencies to infer sensitive information with high confidence. To effectively prevent probabilistic inferences, DBIC mechanisms should (1) support a large class of probabilistic dependencies, and (2) have tractable runtime performance. The former is needed to express different attacker models. The latter is necessary for mechanisms to scale to real-world databases. Most existing DBIC mechanisms, however, support only precise data dependencies [36, 40, 157, 158, 168] or just limited classes of probabilistic dependencies [44, 45, 91, 102, 120, 121, 166]. As a result, they cannot reason about the complex probabilistic dependencies that exist in many realistic settings. Mardziel et al.'s mechanism [116] instead supports arbitrary probabilistic dependencies, but no complexity bounds have been established and their algorithm appears to be intractable.

In this chapter, we propose foundations for DBIC based on ProbLog, a probabilistic logic programming language. We leverage these foundations to develop Angerona[1], a provably secure enforcement mechanism that prevents information leakage in the presence of probabilistic dependencies. We then provide a tractable inference algorithm for a practically relevant fragment of ProbLog. Finally, we empirically evaluate Angerona's performance showing that it scales to relevant security-critical problems. Note that this chapter is largely based on [88].

Structure. In Section 4.1, we illustrate the security risks associated with probabilistic data dependencies. In Section 4.2, we present our system model, which we formalize in Section 4.3. In Section 4.4, we introduce AtkLog, a language, based on ProbLog, for formalizing attacker models in DBIC, whereas in Section 4.5 we present our dedicated inference engine for acyclic ProbLog programs. In Section 4.6, we present Angerona. We discuss related work in Section 4.7 and draw conclusions in Section 4.8. Finally, in Section 4.9 we present some extensions, technical details, and additional examples. The proof of all our results are given in Appendix B, and a prototype of our enforcement mechanism is available at [85].

4.1 Motivating Example

Hospitals and medical research centres store large quantities of health-related information for purposes ranging from diagnosis to research. As this information is extremely sensitive, the databases used must be carefully secured [7, 130]. This task is, however, challenging due to the dependencies between health-related data items. For instance, information about someone's hereditary diseases or genome can be inferred from information about her relatives. Even seemingly non-sensitive information, such as someone's job or habits, may leak sensitive health-related information such as her predisposition to diseases. Most of these dependencies can be formalized using probabilistic models developed by medical researchers.

Consider a database storing information about the smoking habits of patients and whether they have been diagnosed with lung cancer. The database contains the tables *patient*, *smokes*, *cancer*,

[1] Angerona is the Roman goddess of silence and secrecy. She is the keeper of the city's sacred, and secret, name.

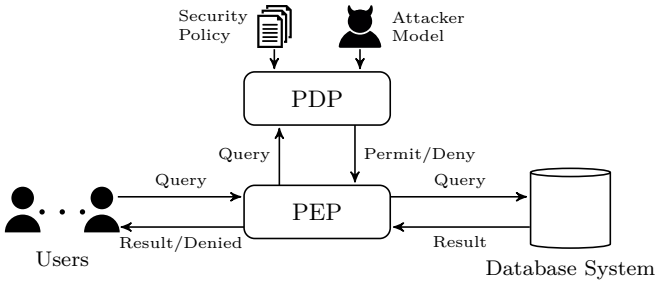


FIGURE 4.1: System model.

father, and *mother*. The first table contains all patients, the second contains all regular smokers, the third contains all diagnosed patients, and the last two associate patients with their parents. Now consider the following probabilistic model: (a) every patient has a 5% chance of developing cancer, (b) for each parent with cancer, the likelihood that a child develops cancer increases by 15%, and (c) if a patient smokes regularly, his probability of developing cancer increases by 25%. We intentionally work with a simple model since, despite its simplicity, it illustrates the challenges of securing data with probabilistic dependencies. We refer the reader to medical research for more realistic probabilistic models [14, 161].

The database is shared between different medical researchers, each conducting a research study on a subset of the patients. All researchers have access to the *patient*, *smokes*, *father*, and *mother* tables. Each researcher, however, has access only to the subset of the *cancer* table associated with the patients that opted-in to his research study. We want to protect our database against a malicious researcher whose goal is to infer the health status of patients not participating in the study. This is challenging since restricting direct access to the *cancer* table is insufficient. Sensitive information may be leaked even by queries involving only authorized data. For instance, the attacker may know that the patient *Carl*, which has not disclosed his health status, smokes regularly. From this, he can infer that *Carl*'s probability of developing lung cancer is, at least, 30%. If, additionally, *Carl*'s parents opted-in to the research study and both have cancer, the attacker can directly infer that the probability of *Carl* developing lung cancer is 60% by accessing his parents' information.

Security mechanisms that ignore such probabilistic dependencies allow attackers to infer sensitive information. An alternative is to use standard DBIC mechanisms and encode all dependencies as precise, non-probabilistic, dependencies. This, however, would result in an unusable system. Medical researchers, even honest ones, would be able to access the health-related status only of those patients whose relatives also opted-in to the user study, independently of the amount of leaked information, which may be negligible. Hence, to secure the database and retain usability, it is essential to reason about the probabilistic dependencies.

4.2 System Model

Figure 4.1 depicts our system model. Users interact with two components: a database system and an inference control system, which consists of a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). We assume that all communication between users and the components and between the components themselves is over secure channels.

Database System. The database system manages the system's data. Its state is a mapping from tables to sets of tuples.

Users. Each user has a unique account used to retrieve information from the database system by issuing `SELECT` commands. Note that these commands do not change the database state. This reflects settings where users have only read-access to a database. Each command is checked by the inference control system and is executed if and only if the command is authorized by the security policy.

Security policy. The system's security policy consists of a set of *negative permissions* specifying information to be kept secret. These permissions express bounds on users' beliefs, formalized as probability distributions, about the actual database content. Negative permissions are formalized using commands of the form `SECRET` q `FOR` u `THRESHOLD` l, where q is a query, u is a user identifier, and l is a rational number, $0 \leq l \leq 1$. This represents the requirement that "A user u's belief in the result of q must be less than l." Namely, the probability assigned by u's belief to q's result must be less than l. Requirements like "A user u is not authorized to know the result of q" can be formalized as `SECRET` q `FOR` u `THRESHOLD` 1. The system also supports commands of the form `SECRET` q `FOR USERS NOT IN` $\{u_1, \ldots, u_n\}$ `THRESHOLD` l, which represents the requirement that "For all users $u \notin \{u_1, \ldots, u_n\}$, u's belief in the result of q must be less than l."

patient
Alice
Bob
Carl

smokes
Bob
Carl

father	
Bob	Carl

mother	
Alice	Carl

FIGURE 4.2: The template for all database states, where the content of the *cancer* table is left unspecified.

Attacker. An attacker is a system user with an assigned user account, and each user is a potential attacker. An attacker's goal is to violate the security policy, that is, to read or infer information about one of the `SECRET`s with a probability of at least the given threshold.

An attacker can interact with the system and observe its behavior in response to his commands. Furthermore, he can reason about this information and infer information by exploiting domain-specific relationships between data items. We assume that attackers know the database schema as well as any integrity constraints on it.

Attacker Model. An attacker model represents each user's initial beliefs about the actual database state and how he updates his beliefs by interacting with the system and observing its behavior in response to his commands. These beliefs may reflect the attacker's knowledge of domain-specific relationships between the data items or prior knowledge.

Inference Control System. The inference control system protects the confidentiality of database data. It consists of a PEP and a PDP, configured with a security policy P and an attacker model ATK. For each user, the inference control system keeps track of the user's beliefs according to ATK.

The system intercepts all commands issued by the users. When a user u issues a command c, the inference control system decides whether u is authorized to execute c. If c complies with the policy, i.e., the users' beliefs still satisfy P even after executing c, then the system forwards the command to the database, which executes c and returns its result to u. Otherwise, it raises a *security exception* and rejects c.

4.3 Formal Model

Here we formalize the various components of our system model.

4.3.1 Database Model

We use the database model presented in Section 2.2, and we use the relational calculus as a query language (see Section 2.2.2). Observe that we support arbitrary integrity constraints. Finally, we assume that the domain **dom** is finite, as is standard for many application areas combining databases and probabilistic reasoning [60, 77, 105, 153]. In this case, the set of all states Ω_D is finite.

Example 4.1. The database associated with the example in Section 4.1 consists of five relational schemas *patient*, *smokes*, *cancer*, *father*, and *mother*, where the first three schemas have arity 1 and the last two have arity 2. We assume that there are only three patients `Alice`, `Bob`, and `Carl`, so the domain **dom** is $\{\texttt{Alice}, \texttt{Bob}, \texttt{Carl}\}$. The integrity constraints are as follows:

- `Alice`, `Bob`, and `Carl` are patients.

$$\mathit{patient}(\texttt{Alice}) \wedge \mathit{patient}(\texttt{Bob}) \wedge \mathit{patient}(\texttt{Carl})$$

- `Alice` and `Bob` are `Carl`'s parents.

$$\forall x, y.\ (\mathit{father}(x, y) \leftrightarrow (x = \texttt{Bob} \wedge y = \texttt{Carl})) \wedge$$
$$\forall x, y.\ (\mathit{mother}(x, y) \leftrightarrow (x = \texttt{Alice} \wedge y = \texttt{Carl}))$$

- `Alice` does not smoke, whereas `Bob` and `Carl` do.

$$\neg \mathit{smokes}(\texttt{Alice}) \wedge \mathit{smokes}(\texttt{Bob}) \wedge \mathit{smokes}(\texttt{Carl})$$

Given these constraints, there are just 8 possible database states in Ω_D^Γ, which differ only in their *cancer* relation. The content of the *cancer* relation is a subset of $\{\texttt{Alice}, \texttt{Bob}, \texttt{Carl}\}$, whereas the content of the other tables is shown in Figure 4.2. We denote each possible world as s_C, where the set $C \subseteq \{\texttt{Alice}, \texttt{Bob}, \texttt{Carl}\}$ denotes the users having cancer. ■

4.3.2 Security Policies

Existing access control models for databases are inadequate to formalize security requirements capturing probabilistic dependencies. For example, SQL cannot express statements like "A user u's belief that ϕ holds must be less than l." We present a simple framework, inspired by knowledge-based policies [116], for expressing such requirements.

A *D-secret* is a tuple $\langle U, \phi, l\rangle$, where U is either a finite set of users in $\mathcal{U}$ or a co-finite set of users, i.e., $U = \mathcal{U} \setminus U'$ for some finite $U' \subset \mathcal{U}$, ϕ is a relational calculus sentence over D, and l is rational number $0 \leq l \leq 1$ specifying the uncertainty threshold. Abusing notation, when U consists of a single user u, we write u instead of $\{u\}$. Informally, $\langle U, \phi, l\rangle$ represents that for each user $u \in U$, u's belief that ϕ holds in the actual database state must be less than l. Therefore, a command of the form `SECRET` q `FOR` u `THRESHOLD` l can be represented as $\langle u, q, l\rangle$, whereas a command `SECRET` q `FOR USERS NOT IN` $\{u_1, \ldots, u_n\}$ `THRESHOLD` l can be represented as $\langle \mathcal{U} \setminus \{u_1, \ldots, u_n\}, q, l\rangle$. Finally, a *$D$-security policy* is a finite set of D-secrets. Given a D-security policy P, we denote by $\mathit{secrets}(P, u)$ the set of D-secrets associated with the user u, i.e., $\mathit{secrets}(P, u) = \{\langle u, \phi, l\rangle \mid \langle U, \phi, l\rangle \in P \wedge u \in U\}$. Note that the function *secrets* is computable since the set U is always either finite or co-finite.

Our framework also allows the specification of lower bounds. Requirements of the form "A user u's belief that ϕ holds must be greater than l" can be formalized as $\langle u, \neg\phi, 1 - l\rangle$ (since the probability of $\neg\phi$ is $1 - P(\phi)$, where $P(\phi)$ is ϕ's probability). Security policies can be extended to support secrets over non-boolean queries. A secret $\langle u, \{\overline{x} \mid \phi(\overline{x})\}, l\rangle$ can be seen as a shorthand for the set $\{\langle u, \phi[\overline{x} \mapsto \overline{t}], l\rangle \mid \overline{t} \in \bigcup_{s \in \Omega_D^\Gamma} [\{\overline{x} \mid \phi(\overline{x})\}]^s\}$, i.e., u's belief in any tuple $\overline{t}$ being in the query's result must be less than l.

Example 4.2. Let *Mallory* denote the malicious researcher from Section 4.1 and D be the database schema in Example 4.1. Consider the requirement from Section 4.1: *Mallory*'s belief in a patient having cancer must be less than 50%. This can be formalized as $\langle \mathit{Mallory}, \mathit{cancer}(\texttt{Alice}), 1/2\rangle$, $\langle \mathit{Mallory}, \mathit{cancer}(\texttt{Bob}), 1/2\rangle$, and $\langle \mathit{Mallory}, \mathit{cancer}(\texttt{Carl}), 1/2\rangle$, or equivalently as $\langle \mathit{Mallory}, \{p \mid \mathit{cancer}(p)\}, 1/2\rangle$. In contrast, the requirement "For all users u that are not *Carl*, u's belief in *Carl* having cancer must be less than 50%" can be formalized as $\langle \mathcal{U} \setminus \{\mathit{Carl}\}, \mathit{cancer}(\texttt{Carl}), 1/2\rangle$, where *Carl* denotes the user identifier associated with Carl. ■

4.3.3 Formalized System Model

We now formalize our system model. We first define a system configuration, which describes the database schema and the integrity constraints. Afterwards, we define the system's state. Finally, we define a system run, which represents a possible interaction of users with the system.

A *system configuration* is a tuple $\langle D, \Gamma\rangle$, where D is a database schema and Γ is a set of integrity constraints over D. Let $C = \langle D, \Gamma\rangle$ be a system configuration. A *C-system state* is a tuple $\langle db, U, P\rangle$, where $db \in \Omega_D^\Gamma$ is a database state, $U \subset \mathcal{U}$ is a finite set of users (representing all active users), and P is a D-security policy. A *C-query* is a pair $\langle u, \phi\rangle$ where $u \in \mathcal{U}$ is a user and ϕ is a relational calculus sentence over D.[2] We denote by Ω_C the set of all system states and by $\mathcal{Q}_C$ the set of all queries.

A *C-event* is a triple $\langle q, a, \mathit{res}\rangle$, where q is a C-query in $\mathcal{Q}_C$, $a \in \{\top, \bot\}$ is a security decision, where $\top$ stands for "authorized query" and $\bot$ stands for "unauthorized query", and $\mathit{res} \in \{\top, \bot, \dagger\}$ is the query's result, where $\top$ and $\bot$ represent the usual boolean values and $\dagger$ represents that the query was not executed as access was denied. Given a C-event $e = \langle q, a, \mathit{res}\rangle$, we denote by $q(e)$ (respectively $a(e)$ and $\mathit{res}(e)$) the query q (respectively the decision a and the result *res*). A *C-history* is a finite sequence of C-events. We denote by $\mathcal{H}_C$ the set of all possible C-histories.

We now formalize Policy Decision Points. A *C-PDP* is a function $f : \Omega_C \times \mathcal{Q}_C \times \mathcal{H}_C \to \{\top, \bot\}$ taking as input a system state, a query, and a history and returning the security decision, accept ($\top$) or deny ($\bot$).

Let C be a system configuration, $s = \langle db, U, P\rangle$ be a C-state, and f be a C-PDP. A C-history h is *compatible with s and f* iff for each $1 \leq i \leq |h|$, (1) $f(s, q(h(i)), h^{i-1}) = a(h(i))$, (2) if $a(h(i)) = \bot$, then $\mathit{res}(h(i)) = \dagger$, and (3) if $a(h(i)) = \top$, then $\mathit{res}(h(i)) = [\phi]^{db}$, where $q(h(i)) = \langle u, \phi\rangle$. In other words, h is compatible with s and f iff it was generated by the PDP f starting in state s.

A *(C, f)-run* is a pair $\langle s, h\rangle$, where s is a system state in Ω_C and h is a history in $\mathcal{H}_C$ compatible with s and f. Since all queries are `SELECT` queries, the system state does not change along the run. Hence, our runs consist of a state and a history instead of e.g., an alternating sequence of states and actions (as is standard for runs). We denote by $\mathit{runs}(C, f)$ the set of all (C, f)-runs. Furthermore,

[2] Without loss of generality, we focus only on boolean queries. We can support non-boolean queries as follows. Given a query $q := \{\overline{x} \mid \phi\}$ and a database state db, we first compute the set $\mathbb{R}$ of all possible results of q, i.e., $\mathbb{R} = \{[q]^{db} \mid db \in \Omega_D^\Gamma\}$. Then, we authorize the query q and return its result $[q]^{db}$ iff for all $R \in \mathbb{R}$, the inference control mechanism authorizes the boolean query $\bigwedge_{\overline{t} \in R} \phi[\overline{x} \mapsto \overline{t}] \wedge (\forall \overline{x}.\, \phi \to \bigvee_{\overline{t} \in R} \overline{x} = \overline{t})$.

given a run $r = \langle\langle db, U, P\rangle, h\rangle$, we denote by r^i the run $\langle\langle db, U, P\rangle, h^i\rangle$, and we use a dot notation to access to r's components. For instance, $r.db$ denotes the database state db and $r.h$ denotes the history.

Example 4.3. Consider the run $r = \langle\langle db, U, P\rangle, h\rangle$, where the database state db is the state $s_{\{\texttt{A},\texttt{B},\texttt{C}\}}$, where `Alice`, `Bob`, and `Carl` have cancer, the policy P is defined in Example 4.2, the set of users U contains only *Mallory*, and the history h is as follows (here we assume that all queries are authorized):

1. *Mallory* checks whether *Carl* smokes. Thus, $h(1) = \langle\langle \mathit{Mallory}, \mathit{smokes}(\texttt{Carl})\rangle, \top, \top\rangle$.
2. *Mallory* checks whether *Carl* is *Alice*'s and *Bob*'s son. Therefore, $h(2)$ is $\langle\langle \mathit{Mallory}, \mathit{father}(\texttt{Bob}, \texttt{Carl}) \wedge \mathit{mother}(\texttt{Alice}, \texttt{Carl})\rangle, \top, \top\rangle$.
3. *Mallory* checks whether *Alice* has cancer. Thus, $h(3) = \langle\langle \mathit{Mallory}, \mathit{cancer}(\texttt{Alice})\rangle, \top, \top\rangle$.
4. *Mallory* checks whether *Bob* has cancer. Thus, $h(4) = \langle\langle \mathit{Mallory}, \mathit{cancer}(\texttt{Bob})\rangle, \top, \top\rangle$. ■

4.3.4 Attacker Model

To reason about DBIC, it is essential to precisely define (1) how users interact with the system, (2) how they reason about the system's behavior, (3) their initial beliefs about the database state, and (4) how these beliefs change by observing the system's behavior. We formalize this in an attacker model.

Each user has an initial belief about the database state. Following [52, 53, 71, 116], we represent a user's beliefs as a probability distribution over all database states. Furthermore, users observe the system's behavior and derive information about the database content. We formalize a user's observations as an equivalence relation over runs, where two runs are equivalent iff the user's observations are the same in both runs, as is standard in information-flow control [18, 20]. A user's knowledge is the set of all database states that he considers possible given his observations. Finally, we use Bayesian conditioning to update a user's beliefs given his observations.

Let $C = \langle D, \Gamma\rangle$ be a system configuration and f be a C-PDP. A *C-probability distribution* is a discrete probability distribution given by a function $P : \Omega_D^\Gamma \to [0, 1]$ such that $\sum_{db \in \Omega_D^\Gamma} P(db) = 1$. Given a set $E \subseteq \Omega_D^\Gamma$, $P(E)$ denotes $\sum_{s \in E} P(s)$. Furthermore, given two sets $E', E'' \subseteq \Omega_D^\Gamma$ such that $P(E') \neq 0$, $P(E'' \mid E')$ denotes $P(E'' \cap E')/P(E')$ as is standard. We denote by $\mathcal{P}_C$ the set of all possible C-probability distributions. Abusing notation, we extend probability distributions to formulae: $P(\psi) = P(\llbracket\psi\rrbracket)$, where $\llbracket\psi\rrbracket = \{db \in \Omega_D^\Gamma \mid [\psi]^{db} = \top\}$.

We now introduce *indistinstinguishability*, an equivalence relation used in information-flow control [93]. Let C be a system configuration and f be a C-PDP. Given a history h and a user $u \in \mathcal{U}$, $h|_u$ denotes the history obtained from h by removing all C-events from users other than u, namely $\epsilon|_u = \epsilon$, and if $h = \langle\langle u', q\rangle, a, res\rangle \cdot h'$, then $h|_u = h'|_u$ in case $u \neq u'$, and $h|_u = \langle\langle u, q\rangle, a, res\rangle \cdot h'|_u$ if $u = u'$. Given two runs $r = \langle\langle db, U, P\rangle, h\rangle$ and $r' = \langle\langle db', U', P'\rangle, h'\rangle$ in $\mathit{runs}(C, f)$ and a user $u \in \mathcal{U}$, we say that r and r' are *indistinguishable for* u, written $r \sim_u r'$, iff $h|_u = h'|_u$. This means that r and r' are indistinguishable for a user u iff the system's behavior in response to u's commands is the same in both runs. Note that $\sim_u$ depends on both C and f, which we generally leave implicit. Given a run r, $[r]_{\sim_u}$ is the equivalence class of r with respect to $\sim_u$, i.e, $[r]_{\sim_u} = \{r' \in \mathit{runs}(C, f) \mid r' \sim_u r\}$, whereas $\llbracket r\rrbracket_{\sim_u}$ is set of all databases associated with the runs in $[r]_{\sim_u}$, i.e., $\llbracket r\rrbracket_{\sim_u} = \{db \mid \exists U, P, h.\ \langle\langle db, U, P\rangle, h\rangle \in [r]_{\sim_u}\}$.

Definition 4.1. Let $C = \langle D, \Gamma\rangle$ be a configuration and f be a C-PDP. A *(C, f)-attacker model* is a function $ATK : \mathcal{U} \to \mathcal{P}_C$ associating to each user $u \in \mathcal{U}$ a C-probability distribution representing u's initial beliefs. Additionally, for all users $u \in \mathcal{U}$ and all states $db \in \Omega_D^\Gamma$, we require that $ATK(u)(db) > 0$. The *semantics of* ATK is $\llbracket ATK\rrbracket(u, r) = \lambda db \in \Omega_D^\Gamma.\ ATK(u)(db \mid \llbracket r\rrbracket_{\sim_u})$, where $u \in \mathcal{U}$ and $r \in \mathit{runs}(C, f)$. □

The semantics of an attacker model ATK associates to each user u and each run r the probability distribution obtained by updating u's initial beliefs given his knowledge with respect to the run r. We informally refer to $\llbracket ATK\rrbracket(u, r)(\llbracket\phi\rrbracket)$ as u's beliefs in a sentence ϕ (given a run r).

Example 4.4. The attacker model for our motivating example from Section 4.1 is as follows. Let $X_{\texttt{Alice}}$, $X_{\texttt{Bob}}$, and $X_{\texttt{Carl}}$ be three boolean random variables, representing the probability that the corresponding patient has cancer. They define the following joint probability distribution, which represents a user's initial beliefs about the actual database state: $P(X_{\texttt{Alice}}, X_{\texttt{Bob}}, X_{\texttt{Carl}}) = P(X_{\texttt{Alice}}) \cdot P(X_{\texttt{Bob}}) \cdot P(X_{\texttt{Carl}} \mid X_{\texttt{Alice}}, X_{\texttt{Bob}})$. The probability distributions of these variables are given in Figure 4.3 and they are derived from the probabilistic model in Section 4.1. We associate each outcome $\langle x, y, z\rangle$ of $X_{\texttt{Alice}}, X_{\texttt{Bob}}, X_{\texttt{Carl}}$ with the corresponding database state s_C, where C is the set of patients such that the outcome of the corresponding variable is $\top$. For each user $u \in \mathcal{U}$, the distribution P_u is defined as $P_u(s_C) = P(X_{\texttt{Alice}} = x, X_{\texttt{Bob}} = y, X_{\texttt{Carl}} = z)$, where x (respectively y and z) is $\top$ if `Alice` (respectively `Bob` and `Carl`) is in C and $\bot$ otherwise. Figure 4.4 shows the probabilities associated with each state in Ω_D^Γ, i.e., a user's initial beliefs. Finally, the attacker model is $ATK = \lambda u \in \mathcal{U}.\ P_u$. ■

	$X_{\texttt{Alice}}$
$\top$	$1/20$
$\bot$	$19/20$

	$X_{\texttt{Bob}}$
$\top$	$6/20$
$\bot$	$14/20$

$X_{\texttt{Alice}}$	$X_{\texttt{Bob}}$	$X_{\texttt{Carl}}$ $\top$	$X_{\texttt{Carl}}$ $\bot$
$\top$	$\top$	$12/20$	$8/20$
$\top$	$\bot$	$9/20$	$11/20$
$\bot$	$\top$	$9/20$	$11/20$
$\bot$	$\bot$	$6/20$	$14/20$

FIGURE 4.3: Probability distribution for the random variables $X_{\texttt{Alice}}$, $X_{\texttt{Bob}}$, and $X_{\texttt{Carl}}$ from Example 4.4.

State	Probability
$s_{\emptyset}$	0.4655
$s_{\{\texttt{A}\}}$	0.01925
$s_{\{\texttt{B}\}}$	0.15675
$s_{\{\texttt{C}\}}$	0.1995

State	Probability
$s_{\{\texttt{A,B}\}}$	0.006
$s_{\{\texttt{A,C}\}}$	0.01575
$s_{\{\texttt{B,C}\}}$	0.12825
$s_{\{\texttt{A,B,C}\}}$	0.009

FIGURE 4.4: Probability distribution over all database states. Each state is denoted as s_C, where C is the content of the *cancer* table. Here we denote the patients' names with their initials.

4.3.5 Confidentiality

We first define the notion of a secrecy-preserving run for a secret $\langle u, \phi, l\rangle$ and an attacker model ATK. Informally, a run r is secrecy-preserving for $\langle u, \phi, l\rangle$ iff whenever an attacker's belief in the secret ϕ is below the threshold l, then there is no way for the attacker to increase his belief in ϕ above the threshold. Our notion of secrecy-preserving runs is inspired by existing security notions for query auditing [71].

Definition 4.2. Let $C = \langle D, \Gamma\rangle$ be a configuration, f be a C-PDP, and ATK be a (C, f)-attacker model. A run r is *secrecy-preserving* for a secret $\langle u, \phi, l\rangle$ and ATK iff for all $0 \leq i < |r|$, $[\![ATK]\!](u, r^i)(\phi) < l$ implies $[\![ATK]\!](u, r^{i+1})(\phi) < l$. □

We now formalize our confidentiality notion. A PDP provides data confidentiality for an attacker model ATK iff all runs are secrecy-preserving for ATK. Note that our security notion can be seen as a probabilistic generalization of opacity [142] for the database setting. Our notion is also inspired by the semantics of knowledge-based policies [116].

Definition 4.3. Let $C = \langle D, \Gamma\rangle$ be a system configuration, f be a C-PDP, and ATK be a (C, f)-attacker model. We say that the PDP f *provides data confidentiality with respect to* C *and* ATK iff for all runs $r = \langle\langle db, U, P\rangle, h\rangle$ in $runs(C, f)$, for all users $u \in U$, for all secrets $s \in secrets(P, u)$, r is secrecy-preserving for s and ATK. □

A PDP providing confidentiality ensures that if an attacker's initial belief in a secret ϕ is below the corresponding threshold, then there is no way for the attacker to increase his belief in ϕ above the threshold by interacting with the system. However, this guarantee does not apply to *trivial non-secrets*, i.e., those secrets an attacker knows with a probability at least the threshold even before interacting with the system. No PDP can prevent their disclosure since the disclosure does not depend on the attacker's interaction with the database.

Example 4.5. Let r be the run given in Example 4.3, ATK be the attacker model in Example 4.4, and u be the user *Mallory*. In the following, ϕ_1, ϕ_2, and ϕ_3 denote $cancer(\texttt{Carl})$, $cancer(\texttt{Bob})$, and $cancer(\texttt{Alice})$ respectively, i.e., the three secrets from Example 4.2. Furthermore, we assume that the policy contains an additional secret $\langle Mallory, \phi_4, 1/2\rangle$, where $\phi_4 := \neg cancer(\texttt{Alice})$.

Figure 4.5 illustrates *Mallory*'s beliefs about $\phi_1, \ldots, \phi_4$ and whether the run is secrecy-preserving for the secrets $\phi_1, \ldots, \phi_4$. The probabilities in the tables can be obtained by combining the states in $[\![r^i]\!]_{\sim_u}$, for $0 \leq i \leq 4$, and $[\![\phi_j]\!]$, for $1 \leq j \leq 4$, with the probabilities from Figure 4.4. As shown in Figure 4.5, the run is not secrecy-preserving for the secrets ϕ_1 and ϕ_2 as it completely discloses that *Alice* and *Bob* have cancer, in the third and fourth steps respectively. Secrecy-preservation is also violated for the secret ϕ_1, even though r does not directly disclose any information about *Carl*'s health status. Indeed, in the last step of the run, *Mallory*'s belief in ϕ_1 is 0.6, which is higher than the threshold $1/2$, even though his belief in ϕ_1 before learning that *Bob* had cancer was below the threshold. Note that ϕ_4 is a trivial non-secret: even before interacting with the system, *Mallory*'s belief in ϕ_4 is 0.95. ■

4.3.6 Discussion

Our approach assumes that the attacker's capabilities are well-defined. While this, in general, is a strong assumption, there are many domains where such information is known. There are, however,

i	$[\![ATK]\!](u, r^i)([\![\phi]\!])$				$[\![ATK]\!](u, r^{i+1})([\![\phi]\!])$				Secrecy			
	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_1	ϕ_2	ϕ_3	ϕ_4
0	0.3525	0.3	0.05	0.95	0.3525	0.3	0.05	0.95	✓	✓	✓	$*$
1	0.3525	0.3	0.05	0.95	0.3525	0.3	0.05	0.95	✓	✓	✓	$*$
2	0.3525	0.3	0.05	0.95	0.495	0.3	1	0	✓	✓	$\mathcal{X}$	$*$
3	0.495	0.3	1	0	0.6	1	1	0	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$	$*$
4	0.6	1	1	0	–	–	–	–	–	–	–	–

FIGURE 4.5: Evolution of *Mallory*'s beliefs in the secrets $\phi_1, \ldots, \phi_4$ for the run r and the attacker model ATK from Example 4.5. In the table, $\mathcal{X}$ and ✓ denote that secrecy-preservation is violated and satisfied respectively, whereas $*$ denotes trivial secrets.

domains where this information is lacking. In these cases, security engineers must (1) determine the appropriate beliefs capturing the desired attacker models, and (2) formalize them. The latter can be done, for instance, using AtkLog (see Section 4.4). Note however that precisely eliciting the attackers' capabilities is still an open problem in DBIC.

4.4 AtkLog

We now describe AtkLog, a language for formalizing a large class of attacker models. AtkLog is built on top of ProbLog [60, 61, 73], a state-of-the-art probabilistic logic programming language. We first briefly review ProbLog's syntax and semantics. Afterwards, we present AtkLog and use it to formalize the attacker model associated with our motivating example from Section 4.1.

4.4.1 Probabilistic Logic Programming

ProbLog [60, 61, 73] is a probabilistic logic programming language with associated tool support. An exact inference engine for ProbLog is available at [6].

Here we introduce ProbLog by building on top of our logic programming's foundations from Section 2.3. Thus, following our treatment of logic programs in Section 2.3, we restrict ourselves to function-free stratified ProbLog programs with negation. As a result, in our setting ProbLog is a probabilistic extension of stratified Datalog with negation. In the following, let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that $\mathbf{dom}$ is a finite set.

Syntax. To reason about probabilities, ProbLog extends logic programs with probabilistic atoms. A $(\Sigma, \mathbf{dom})$-*probabilistic atom* is a $(\Sigma, \mathbf{dom})$-atom a annotated with a rational value $0 \leq v \leq 1$, denoted $v{::}a$. If $v = 1$, then we write $a(\overline{c})$ instead of $1{::}a(\overline{c})$. A $(\Sigma, \mathbf{dom})$-ProbLog *program* is a finite set of ground $(\Sigma, \mathbf{dom})$-probabilistic atoms and $(\Sigma, \mathbf{dom})$-rules. Note that ground atoms $a \in \mathcal{A}_{\Sigma,\mathbf{dom}}$ are represented as $1{::}a$. Given a ProbLog program p, we denote by $prob(p)$ the set of all probabilistic ground atoms $v{::}a$ in p, i.e., $prob(p) := \{v{::}a \in p \mid 0 \leq v \leq 1 \wedge a \in \mathcal{A}_{\Sigma,\mathbf{dom}}\}$, and by $rules(p)$ the (non-probabilistic) rules in p, i.e., $rules(p) := p \setminus prob(p)$. Observe that ProbLog programs are subject to all restrictions we introduced in Section 2.3 for logic programs. For instance, we consider only programs p that admit a stratification. Finally, we say that a ProbLog program p is a *logic program* iff $v = 1$ for all $v{::}a \in prob(p)$, i.e., p does not contain probabilistic atoms.

Semantics. Given a $(\Sigma, \mathbf{dom})$-ProbLog program p, a *p-grounded instance* is a logic program $A \cup R$, where the set of ground atoms A is a subset of $\{a \mid \exists v.\ v{::}a \in prob(p)\}$ and $R = rules(p)$. Informally, a grounded instance of p is one of the logic programs that can be obtained by selecting some of the probabilistic atoms in p and keeping all rules in p. A *p-probabilistic assignment* is a total function associating to each probabilistic atom $v{::}a$ in $prob(p)$ a value in $\{\top, \bot\}$. We denote by $A(p)$ the set of all p-probabilistic assignments. The *probability* of a p-probabilistic assignment f is $prob(f) = \prod_{v::a \in \{v::a \in prob(p) \mid f(v::a) = \top\}} v \cdot \prod_{v::a \in \{v::a \in prob(p) \mid f(v::a) = \bot\}} (1 - v)$. Given a p-probabilistic assignment f, $instance(p, f)$ denotes the p-grounded instance $\{a \mid \exists v.\ f(v{::}a) = \top\} \cup rules(p)$.

The semantics of a $(\Sigma, \mathbf{dom})$-ProbLog program p is defined as a probability distribution over all possible p-grounded instances. Note that ProbLog's semantics relies on the *closed world assumption*, namely every fact that is not in a given model is considered false. The semantics of p, denoted by $[\![p]\!]$, is as follows: $[\![p]\!](p') = \sum_{f \in F_{p,p'}} prob(f)$, where $F_{p,p'} = \{f \in A(p) \mid p' = instance(p, f)\}$. We remark that a $(\Sigma, \mathbf{dom})$-ProbLog program p implicitly defines a probability distribution over $(\Sigma, \mathbf{dom})$-structures. Indeed, the probability of a given $(\Sigma, \mathbf{dom})$-structure s is the sum of the probabilities of all p-grounded instances p' such that $[\![p']\!] = s$.[3] With a slight abuse of notation, we extend the

[3] With a slight abuse of notation, we treat a set of ground $(\Sigma, \mathbf{dom})$-atoms as a $(\Sigma, \mathbf{dom})$-structure. This is without loss of generality. Under the closed world assumption, a set of ground atoms directly defines a structure (and vice versa).

semantics of p to $(\Sigma, \mathbf{dom})$-structures and sentences as follows: $[\![p]\!](s) = \sum_{f \in \mathcal{M}(p,s)} prob(f)$, where s is a $(\Sigma, \mathbf{dom})$-structure and $\mathcal{M}(p, s)$ is the set of all assignments f such that $[\![instance(p, f)]\!] = s$. Finally, p's semantics can be lifted to sentences as follows: $[\![p]\!](\phi) = \Sigma_{s \in [\![\phi]\!]} [\![p]\!](s)$, where $[\![\phi]\!]$ is the set of all $(\Sigma, \mathbf{dom})$-structures satisfying ϕ.

Evidence. ProbLog supports expressing *evidence* inside programs [60]. To express evidence, i.e., to condition a probability distribution on some event, we use statements of the form *evidence*(a, v), where a is a ground atom and $v \in \{\texttt{true}, \texttt{false}\}$. Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program p with evidence $evidence(a_1, v_1), \ldots, evidence(a_n, v_n)$, and p' be the program without the evidence statements. Furthermore, let $POX(p')$ be the set of all $(\Sigma, \mathbf{dom})$-structures s complying with the evidence, i.e., the set of all states s such that a_i holds in s iff $v_i = \texttt{true}$. Then, $[\![p]\!](s)$, for a $(\Sigma, \mathbf{dom})$-structure $s \in POX(p)$, is $[\![p']\!](s) \cdot \left(\sum_{s' \in POX(p)} [\![p']\!](s')\right)^{-1}$.

Syntactic Sugar. Following [60, 61, 73], we extend ProbLog programs with two additional constructs: probabilistic rules and annotated disjunctions. As shown in [60], these constructs are just syntactic sugar. A *probabilistic rule* is a ProbLog rule whose head is a probabilistic atom. The probabilistic rule $v{::}h \leftarrow l_1, \ldots, l_n$ can be encoded using the additional probabilistic atoms $v{::}sw(\_)$ and the rule $h \leftarrow l_1, \ldots, l_n, sw(\overline{x})$, where sw is a fresh predicate symbol, $\overline{x}$ is the tuple containing the variables in $vars(h) \cup \bigcup_{1 \leq i \leq n} vars(l_i)$, and $v{::}sw(\_)$ is a shorthand representing the fact that there is a probabilistic atom $v{::}sw(\overline{t})$ for each tuple $\overline{t} \in \mathbf{dom}^{|\overline{x}|}$.

An *annotated disjunction* $v_1{::}a_1(\overline{t}_1); \ldots; v_n{::}a_n(\overline{t}_n)$, where $a_1(\overline{t}_1), \ldots, a_n(\overline{t}_n)$ are ground atoms and $\left(\sum_{1 \leq i \leq n} v_i\right) \leq 1$, denotes that $a_1, \ldots, a_n$ are mutually exclusive probabilistic events happening with probabilities $v_1, \ldots, v_n$. This annotated disjunction can be encoded as:

$$
\begin{aligned}
& p_1{::}sw_1(\_) \\
& \vdots \\
& p_n{::}sw_n(\_) \\
& a_1(\overline{t}_1) \leftarrow sw_1(\overline{t}_1) \\
& a_2(\overline{t}_2) \leftarrow \neg sw_1(\overline{t}_1), sw_2(\overline{t}_2) \\
& \vdots \\
& a_n(\overline{t}_n) \leftarrow \neg sw_1(\overline{t}_1), \ldots, \neg sw_{n-1}(\overline{t}_{n-1}), sw_n(\overline{t}_n),
\end{aligned}
$$

where each p_i, for $1 \leq i \leq n$, is $v_i \cdot \left(1 - \sum_{1 \leq j < i} v_j\right)^{-1}$. Probabilistic rules can be easily extended to support annotated disjunctions in their heads.

Example 4.6. Let Σ be a first-order signature with two predicate symbols V and W, both with arity 1, $\mathbf{dom}$ be the domain $\{a, b\}$, and p be the program consisting of the facts $1/4{::}T(a)$ and $1/2{::}T(b)$, the annotate disjunction $1/4{::}W(a); 1/2{::}W(b)$, and the rule $1/2{::}T(x) \leftarrow W(x)$. The probability associated to each $(\Sigma, \mathbf{dom})$-structure by the program p is shown in the following table.

		W			
		$\emptyset$	$\{a\}$	$\{b\}$	$\{a, b\}$
T	$\emptyset$	$3/32$	$3/64$	$3/32$	0
	$\{a\}$	$1/32$	$5/64$	$1/32$	0
	$\{b\}$	$3/32$	$3/64$	$9/32$	0
	$\{a, b\}$	$1/32$	$5/64$	$3/32$	0

The empty structure has probability $3/32$. The only grounded instance producing the empty database is the instance i_1 that does not contain ground atoms. Its probability is $3/32$ because the probability that $T(a)$ is not in i_1 is $3/4$, the probability that $T(b)$ is not in i_1 is $1/2$, and the probability that neither $W(a)$ nor $W(b)$ are in i_1 is $1/4$ and all these events are independent.

The probability of some structures is determined by more than one grounded instance. For example, the probability of the structure s such that $s(T) = \{a, b\}$ and $s(W) = \{a\}$ is $5/64$. There are two grounded instances i_2 and i_3 that produce s. The instance i_2 has probability $1/16$ and it consists of the atoms $T(b), W(a), sw(a)$ and the rule $T(x) \leftarrow W(x), sw(x)$, whereas the instance i_3 has probability $1/64$ and it consists of the atoms $T(a), T(b), W(a)$ and the rule $T(x) \leftarrow W(x), sw(x)$. Note that before computing the ground instances, we translated probabilistic rules and annotated disjunctions into standard ProbLog rules. ■

Medical Data. We formalize the probability distribution from Example 4.4 as a ProbLog program. We use the database schema from Example 4.1 as the database schema for the ProbLog program.

We encode the template shown in Figure 4.2 using ground atoms: *patient*(`Alice`), *patient*(`Bob`), *patient*(`Carl`), *smokes*(`Bob`), *smokes*(`Carl`), *father*(`Bob`, `Carl`), and *mother*(`Alice`, `Carl`). Finally, we encode the probability distribution associated with the possible values of the *cancer* table using the following probabilistic rules:

$$
\begin{aligned}
1/20&::cancer(x) \leftarrow patient(x)\\
5/19&::cancer(x) \leftarrow smokes(x)\\
3/14&::cancer(y) \leftarrow father(x,y), cancer(x), mother(z,y), \neg cancer(z)\\
3/14&::cancer(y) \leftarrow father(x,y), \neg cancer(x), mother(z,y), cancer(z)\\
3/7&::cancer(y) \leftarrow father(x,y), cancer(x), mother(z,y), cancer(z)
\end{aligned}
$$

The coefficients in the above example are derived from Section 4.1. For instance, the probability that a smoking patient x whose parents are not in the *cancer* relation has cancer is 30%. The coefficient in the first rule is $1/20$ since each patient has a 5% probability of having cancer. The coefficient in the second rule is $5/19$, which is $(6/20 - 1/20) \cdot (1 - 1/20)^{-1}$, i.e., the probability that $cancer(x)$ is derived from the second rule given that it has not been derived from the first rule. This ensures that the overall probability of deriving $cancer(x)$ is $6/20$, i.e., 30%. The coefficients for the last two rules are derived analogously.

Informally, a probabilistic ground atom $1/2::cancer(\texttt{Bob})$ expresses that *cancer*(`Bob`) holds with a probability $1/2$. Similarly, the rule $1/20::cancer(x) \leftarrow patient(x)$ states that, for any x such that $patient(x)$ holds, then $cancer(x)$ can be derived with probability $1/20$. This program yields the probability distribution shown in Figure 4.4.

4.4.2 AtkLog's Foundations

We first introduce belief programs, which formalize an attacker's initial beliefs. Afterwards, we formalize AtkLog.

Belief Programs. A belief program formalizes an attacker's beliefs as a probability distribution over the database states.

A database schema $D' = \langle \Sigma', \mathbf{dom} \rangle$ extends a schema $D = \langle \Sigma, \mathbf{dom} \rangle$ iff Σ' contains all relation schemas in Σ. The extension D' may extend Σ with additional predicate symbols necessary to encode probabilistic dependencies. Given an extension D', a D'-state s' *agrees* with a D-state s iff $s'(R) = s(R)$ for all R in D. Given a D-state s, we denote by $EXT(s, D, D')$ the set of all D'-states that agree with s.

A $(\Sigma', \mathbf{dom})$-ProbLog program p, where $D' = \langle \Sigma', \mathbf{dom} \rangle$ extends D, is a *belief program over* D. The *D-semantics of* p is $[\![p]\!]_D = \lambda s \in \Omega_D . \sum_{s' \in EXT(s,D,D')} [\![p]\!](s')$. Given a system configuration $C = \langle D, \Gamma \rangle$, a belief program p over D *complies with* C iff $[\![p]\!]_D$ is a C-probability distribution. With a slight abuse of notation, we lift the semantics of belief programs to sentences: $[\![p]\!]_D = \lambda \phi \in \mathcal{RC}_{bool} . \sum_{s' \in \{s \in \Omega_D \mid [\phi]^s = \top\}} [\![p]\!]_D(s')$.

AtkLog. An AtkLog model specifies the initial beliefs of all users in $\mathcal{U}$ using belief programs.

Let D be a database schema and $C = \langle D, \Gamma \rangle$ be a system configuration. A C-AtkLog *model* ATK is a function associating to each user $u \in U$, where $U \subset \mathcal{U}$ is a finite set of users, a belief program p_u and to all users $u \in \mathcal{U} \setminus U$ a belief program p_0, such that for all users $u \in \mathcal{U}$, $[\![ATK(u)]\!]_D$ complies with C and for all database states $db \in \Omega_D^\Gamma$, $[\![ATK(u)]\!]_D(db) > 0$, i.e., all database states satisfying the integrity constraints are possible. Informally, a C-AtkLog model associates a distinct belief program to each user in U, and it associates to each user in $\mathcal{U} \setminus U$ the same belief program p_0.

Given a C-PDP f, a C-AtkLog model ATK defines the (C, f)-attacker model $\lambda u \in \mathcal{U} . [\![ATK(u)]\!]_D$ that associates to each user $u \in \mathcal{U}$ the probability distribution defined by the belief program $ATK(u)$. The semantics of this (C, f)-attacker model is: $\lambda u \in \mathcal{U} . \lambda r \in runs(C, f) . \lambda db \in \Omega_D^\Gamma . [\![ATK(u)]\!]_D(db \mid [\![r]\!]_{\sim_u})$. Informally, given a C-AtkLog model ATK, a C-PDP f, and a user u, u's belief in a database state db, given a run r, is obtained by conditioning the probability distribution defined by the belief program $ATK(u)$ given the set of database states corresponding to all runs $r' \sim_u r$.

4.5 Tractable Inference for ProbLog programs

Probabilistic inference in ProbLog is intractable in general. Its data complexity, i.e., the complexity of inference when only the programs' probabilistic ground atoms are part of the input and the rules are considered fixed and not part of the input, is $\#P$-hard; see Appendix B. This limits the practical applicability of ProbLog (and AtkLog) for DBIC. To address this, we define acyclic ProbLog programs, a class of programs where the data complexity of inference is PTime.

Given a ProbLog program p, our inference algorithm consists of three steps: (1) we compute all of p's derivations, (2) we compile these derivations into a Bayesian Network (BN) bn, and (3) we

perform the inference over bn. To ensure tractability, we leverage two key insights. First, we exploit guarded negation [25] to develop a sound over-approximation, called the relaxed grounding, of all derivations of a program that is independent of the presence (or absence) of the probabilistic atoms. This ensures that whenever a ground atom can be derived from a program (for a possible assignment to the probabilistic atoms), the atom is also part of this program's relaxed grounding. This avoids grounding p for each possible assignment to the probabilistic atoms. Second, we introduce syntactic constraints (acyclicity) that ensure that bn is a forest of poly-trees. This ensures tractability since inference for poly-tree BNs can be performed in polynomial time in the network's size [105].

We also precisely characterize the expressiveness of acyclic ProbLog programs. In this respect, we prove that acyclic programs are as expressive as forests of poly-tree BNs, one of the few classes of BNs with tractable inference.

As mentioned in Section 4.4, probabilistic rules and annotated disjunctions are just syntactic sugar. Hence, in the following we consider ProbLog programs consisting just of probabilistic ground atoms and non-probabilistic rules. Note also that we treat ground atoms as rules with an empty body.

In the following, let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that $\mathbf{dom}$ is a finite domain and p be a $(\Sigma, \mathbf{dom})$-program. Without loss of generality, we assume that there are no distinct v_1 and v_2 such that $v_1{::}a(\overline{c}) \in p$ and $v_2{::}a(\overline{c}) \in p$.

4.5.1 Preliminaries

Negation-guarded Programs. A rule r is *negation-guarded* [25] iff all the variables occurring in negative literals also occur in positive literals, namely for all negative literals l in $body^-(r)$, $vars(l) \subseteq \bigcup_{l' \in body^+(r)} vars(l')$. To illustrate, the rule $C(x) \leftarrow A(x), \neg B(x)$ is negation-guarded, whereas the rule $C(x) \leftarrow A(x), \neg B(x,y)$ is not since the variable y does not occur in any positive literal. We say that a program p is *negation-guarded* if all rules $r \in p$ are. Observe that negation-guarded rules ensure domain independence, i.e., the result of a negation-guarded program does not depend on the domain $\mathbf{dom}$.

Relaxed Grounding. The relaxed grounding of a program p is obtained by considering all probabilistic atoms as certain and by grounding all positive literals. For all negation-guarded programs, the relaxed grounding of p is a sound over-approximation of all possible derivations in p. Given a program p and a rule $r \in p$, $ground(p)$ denotes p's relaxed grounding and $ground(p,r)$ denotes the set of r's ground instances.

Formally, $ground(p)$ and $ground(p,r)$ are defined as follows. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that $\mathbf{dom}$ is a finite domain, p be a $(\Sigma, \mathbf{dom})$-negation guarded ProbLog program, and $i \in \mathbb{N}$ be an natural number. The function $ground(p,i)$ is inductively defined as follows: $ground(p,0) = \{a(\overline{c}) \mid a(\overline{c}) \in p \vee \exists v.\, v{::}a(\overline{c}) \in p\}$, and $ground(p,i) = ground(p,i-1) \cup \{h\Theta \mid \exists r \in p.\ (\Theta \in ASGN(r) \wedge h = head(r) \wedge \forall l \in body^+(r).\ l\Theta \in ground(p,i-1) \wedge \forall i,j.\ consistent(body(r,i)\Theta, body(r,j)\Theta) = \top)\}$, where $consistent(a(\overline{v}), \neg a(\overline{v})) = consistent(\neg a(\overline{v}), a(\overline{v})) = \bot$ and $consistent(l,l') = \top$ otherwise. The function $ground(p,r,i)$, where $r \in p$, is inductively defined as follows: $ground(p, a(\overline{c}), i) = \{a(\overline{c})\}$, $ground(p, v{::}a(\overline{c}), i) = \{a(\overline{c})\}$, $ground(p,r,0) = \emptyset$, and $ground(p,r,i)$ is the set $\{h\Theta \leftarrow l_1\Theta, \ldots, l_n\Theta \mid \Theta \in ASGN(r) \wedge \forall l_i \in body^+(r).\ l_i\Theta \in ground(p,i-1) \wedge \forall i,j.\ consistent(l_i\Theta, l_j\Theta) = \top\}$, where $r = h \leftarrow l_1, \ldots, l_n$. Finally, the relaxed grounding of p, denoted by $ground(p)$, is the set $ground(p) = \bigcup_{i \in \mathbb{N}} ground(p,i)$, whereas $ground(p,r) = \bigcup_{i \in \mathbb{N}} ground(p,r,i)$.

Example 4.7. Let p be the program consisting of the facts $1/2{::}A(1)$, $A(2)$, $A(3)$, $D(1)$, $E(2)$, $F(1)$, $O(1,2)$, and $2/3{::}O(2,3)$, and the rules $r_a = B(x) \leftarrow A(x), D(x)$, $r_b = B(x) \leftarrow A(x), E(x)$, and $r_c = B(y) \leftarrow B(x), \neg F(x), O(x,y)$. The relaxed grounding of p consists of the initial facts together with $B(1)$, $B(2)$, and $B(3)$, whereas $ground(p,r_c)$ consists of $B(2) \leftarrow B(1), \neg F(1), O(1,2)$ and $B(3) \leftarrow B(2), \neg F(2), O(2,3)$. ■

Dependency and Ground Graphs. The *dependency graph* of a program p, denoted $graph(p)$, is the labelled directed graph having as nodes all the predicate symbols in p and having an edge $a \xrightarrow{r,i} b$ iff there is a rule r such that a occurs in i-th literal in r's body and b occurs in r's head. Formally, the *p-dependency graph* is the labelled directed graph $\langle N, E \rangle$ where $N = \Sigma$ and $E = \{pred(body(r,i)) \xrightarrow{r,i} pred(head(r)) \mid r \in p \wedge 1 \leq i \leq |body(r)|\}$. Figure 4.6 depicts the dependency graph from Example 4.7.

The *ground graph* of a program p is the graph obtained from its relaxed grounding. Hence, there is an edge $a \xrightarrow{r,gr,i} b$ from the ground atom a to the ground atom b iff there is a rule r and a ground rule $gr \in ground(p,r)$ such that $body(gr,i) \in \{a, \neg a\}$ and $head(gr) = b$. Formally, the ground graph of p, denoted $gg(p)$, is the labelled directed graph $\langle N, E \rangle$ where $N = ground(p) \cup \{l \in body^+(r') \mid \exists r \in p.\ r' \in ground(p,r)\} \cup \{a(\overline{c}) \mid \exists r \in p, r' \in ground(p,r), l \in body^-(r').\ a = pred(l) \wedge \overline{c} = args(l)\}$ and $E = \{a \xrightarrow{r,gr,i} b \mid r \in p \wedge gr \in ground(p,r) \wedge 1 \leq i \leq |body(r)| \wedge body(gr,i) \in \{a, \neg a\} \wedge head(gr) = b\}$.

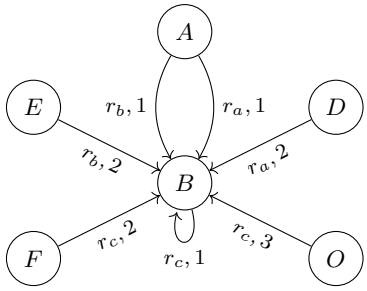


FIGURE 4.6: Dependency graph for the program in Example 4.7.

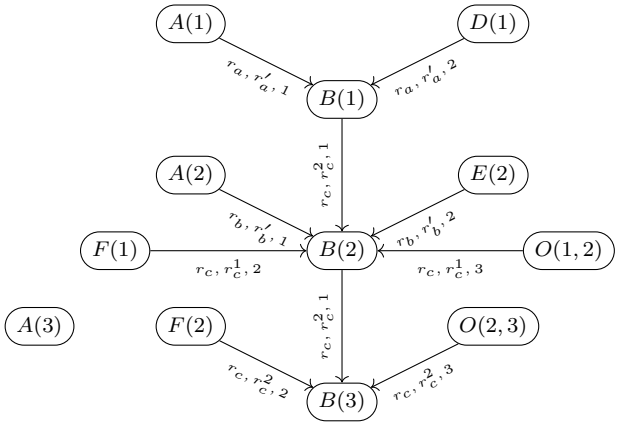


FIGURE 4.7: Ground graph for the program in Example 4.7. The ground rules r'_a, r'_b, r^1_c, and r^2_c are as follows: $r'_a = B(1) \leftarrow A(1), D(1)$, $r'_b = B(2) \leftarrow A(2), E(2)$, $r^1_c = B(2) \leftarrow B(1), \neg F(1), O(1,2)$, and $r^2_c = B(3) \leftarrow B(2), \neg F(2), O(2,3)$.

Figure 4.7 depicts the ground graph from Example 4.7. Note that there are no incoming or outgoing edges from the node $A(3)$ because it is not involved in any derivation.

Paths and Cycles. We now introduce some terminology and notation for paths and cycles in graphs. Let $G = \langle N, E \rangle$ be a directed graph. A *directed path* in G is a sequence of edges $n_1 \rightarrow n'_1 \cdot n_2 \rightarrow n'_2 \cdot \ldots \cdot n_{m-1} \rightarrow n'_{m-1} \cdot n_m \rightarrow n'_m$ such that for all $1 \leq i \leq m$, $n_i \rightarrow n'_i \in E$ and if $i \neq m$, $n'_i = n_{i+1}$. Observe that we denote a path $n_1 \rightarrow n'_1 \cdot n_2 \rightarrow n'_2 \cdot \ldots \cdot n_{m-1} \rightarrow n'_{m-1} \cdot n_m \rightarrow n'_m$ simply as $n_1 \rightarrow n_2 \rightarrow \ldots n_{m-1} \rightarrow n_m \rightarrow n'_m$. Furthermore, the *empty path* is just the empty sequence ϵ. A *directed cycle* in G is a directed path $n_1 \rightarrow n_2 \ldots n_{m-1} \rightarrow n_m$ such that $n_1 = n_m$. We say that a directed cycle C is *simple* iff each edge occurs at most once in C.

Given a directed path $P = n_1 \rightarrow n_2 \ldots n_{m-1} \rightarrow n_m$, we denote by $start(P)$ the node n_1 and by $end(P)$ the node n_m. Two directed paths $n_1 \rightarrow n_2 \ldots n_{m-1} \rightarrow n_m$ and $n'_1 \rightarrow n'_2 \ldots n'_{k-1} \rightarrow n'_k$ are *head-connected* (respectively *tail-connected*) iff $n_1 = n'_1$ (respectively $n_m = n'_k$). Furthermore, a directed path P_1 is *connected* with a directed path P_2 iff $end(P_1) = end(P_2)$, $start(P_1) = start(P_2)$, or $end(P_1) = start(P_2)$. Finally, given a node n, we denote by $reach(n)$ the set of all nodes n' such that there is a direct path from n to n'.

A *reversed path* is a sequence of reversed edges $n_1 \leftarrow n'_1 \cdot n_2 \leftarrow n'_2 \cdot \ldots \cdot n_{m-1} \leftarrow n'_{m-1} \cdot n_m \leftarrow n'_m$ such that $n'_m \rightarrow n_m \cdot n'_{m-1} \rightarrow n_{m-1} \cdot \ldots \cdot n'_2 \rightarrow n_2 \cdot n'_1 \rightarrow n_1$ is a directed path. Given a directed path P, we denote by P^-1 the corresponding reversed path. Similarly, given a reversed path P, we denote by P^-1 the corresponding directed path.

An *undirected path* in G is a sequence of undirected edges $n_1 - n'_1 \cdot n_2 - n'_2 \cdot \ldots \cdot n_{m-1} - n'_{m-1} \cdot n_m - n'_m$ such that for all $1 \leq i \leq m$, $n_i \rightarrow n'_i \in E \vee n'_i \rightarrow n_i \in E$ and if $i \neq m$, $n'_i = n_{i+1}$. Again, we denote a path $n_1 - n'_1 \cdot n_2 - n'_2 \cdot \ldots \cdot n_{m-1} - n'_{m-1} \cdot n_m - n'_m$ simply as $n_1 - n_2 - \ldots n_{m-1} - n_m - n'_m$. An *undirected cycle* in G is an undirected path $n_1 - n_2 - \ldots n_{m-1} - n_m - n'_m$ such that $n_1 = n'_m$. Observe that any undirected cycle C can be seen as a sequence of directed and reversed paths $D_1 \cdot \ldots \cdot D_m$. Observe also that a directed path (respectively cycle) is also an undirected path (respectively cycle). We say that two (directed or undirected) cycles are equivalent iff they are the same cycle.

Propagation Maps. We use propagation maps to track how information flows inside rules. Given a rule r and a literal $l \in body(r)$, the (r,l)-*vertical map* is the mapping μ from $\{1, \ldots, |l|\}$ to $\{1, \ldots, |head(r)|\}$ such that $\mu(i) = j$ iff $args(l)(i) = args(head(r))(j)$ and $args(l)(i) \in Var$. Given a rule

r and literals l and l' in r's body, the (r, l, l')-*horizontal map* is the mapping μ from $\{1, \ldots, |l|\}$ to $\{1, \ldots, |l'|\}$ such that $\mu(i) = j$ iff $args(l)(i) = args(l')(j)$ and $args(l)(i) \in Var$.

We say that a path links to a literal l if information flows along the rules to l. This can be formalized by posing constraints on the mapping obtained by combining horizontal and vertical maps along the path. Formally, given a literal l and a mapping $\nu : \mathbb{N} \to \mathbb{N}$, a directed path $pr_1 \xrightarrow{r_1, i_1} \ldots \xrightarrow{r_{n-1}, i_{n-1}} pr_n$ *ν-downward links to l* iff there is a $0 \leq j < n-1$ such that the function $\mu := \mu' \circ \mu_j \circ \ldots \circ \mu_1$ satisfies $\mu(k) = \nu(k)$ for all k for which $\nu(k)$ is defined, where for $1 \leq h \leq j$, μ_h is the $(r_h, body(r_h, i_h))$-vertical map, and μ' is the horizontal map connecting $body(r_{j+1}, i_{j+1})$ with l. Similarly, a directed path $pr_1 \xrightarrow{r_1, i_1} \ldots \xrightarrow{r_{n-1}, i_{n-1}} pr_n$ *ν-upward links to l* iff there is a $1 \leq j \leq n-1$ such that the function $\mu := \mu'^{-1} \circ \mu_{j+1}^{-1} \circ \ldots \circ \mu_{n-1}^{-1}$ satisfies $\mu(k) = \nu(k)$ for all k for which $\nu(k)$ is defined, where μ_h is the $(r_h, body(r_h, i_h))$-vertical map, for $j < h \leq n-1$, and μ' is the (r_j, l)-vertical map. Given a predicate symbol a, a path P *ν-upward links to a* (respectively *ν-downward links to a*) iff there is an atom $a(\overline{x})$ such that P ν-upward links (respectively ν-downward links) to $a(\overline{x})$.

Example 4.8. The horizontal map connecting $A(x)$ and $D(x)$ in r_a, i.e., the $(r_a, A(x), D(x))$-horizontal map, is $\{1 \mapsto 1\}$. The horizontal map connecting $A(x)$ and $E(x)$ in r_b is $\{1 \mapsto 1\}$ as well. Hence, the path $A \xrightarrow{r_a, 1} B$ downward links to D and the path $A \xrightarrow{r_b, 1} B$ downward links to E for the mapping $\{1 \mapsto 1\}$. Furthermore, the path $B \xrightarrow{r_c, 1} B$ downward links to O for $\{1 \mapsto 1\}$ since the $(r_c, B(x), O(x, y))$-horizontal map is $\{1 \mapsto 1\}$. Finally, the path $B \xrightarrow{r_c, 1} B$ upward links to O for $\{2 \mapsto 1\}$ since the $(r_c, O(x, y))$-vertical map is $\{2 \mapsto 1\}$. ■

4.5.2 Annotations

Annotations represent properties of the relations induced by the program p, and they are syntactically derived by analyzing p's ground atoms and rules. We now illustrate how to derive three kinds of annotations: disjointness, ordering, and uniqueness annotations. In this section, whenever we use the notation $reach(\cdot)$ (defined in Section 4.5.1), we always refer to p's dependency graph.

Disjointness annotations. Let $a, a' \in \Sigma$ be two predicate symbols such that $|a| = |a'|$. A *disjointness annotation* $DIS(a, a')$ represents that the relations induced by a and a' (given p's relaxed grounding) are disjoint. Informally, $DIS(a, a')$ can be derived from p iff (1) no rules in p contain a or a' in their heads, and there is no $\overline{v} \in \mathbf{dom}^{|a|}$ where both $a(\overline{v})$ and $a'(\overline{v})$ appear as (possibly probabilistic) ground atoms in p, or (2) a and a' can be derived only from other disjoint predicate symbols. Hence, the relations induced by a and a' are disjoint. Formally, the annotation $DIS(a, a')$ can be derived from p iff one of the following conditions hold:

1. There are no rules $r \in p$ such that $body(r) \neq \emptyset$ and $pred(head(r)) = a$ or $pred(head(r)) = a'$, and the sets $\{\overline{c} \mid a(\overline{c}) \in p \vee \exists v.\, v{::}a(\overline{c}) \in p\}$ and $\{\overline{c} \mid a'(\overline{c}) \vee \exists v.\, v{::}a'(\overline{c}) \in p\}$ are disjoint.
2. There are no rules $r \in p$ such that $body(r) \neq \emptyset$ and $pred(head(r)) = a$, the annotation $DIS(a, b)$ can be derived from p, $b \notin reach(a')$, $b \neq a'$, and for all rules $r \in p$ such that $head(r) = a'(\overline{x})$, $b(\overline{x}) \in body(r)$.
3. There are no rules $r \in p$ such that $body(r) \neq \emptyset$ and $pred(head(r)) = a'$, the annotation $DIS(b, a')$ can be derived from p, $b \notin reach(a)$, $b \neq a$, and for all rules $r \in p$ such that $head(r) = a(\overline{x})$, $b(\overline{x}) \in body(r)$.
4. The annotation $DIS(b, b')$ can be derived from p, $\{b, b'\} \cap (reach(a) \cup reach(a')) = \emptyset$, $b \neq a$, $b' \neq a'$, and for all rules $r \in p$, if $head(r) = a(\overline{x})$, $b(\overline{x}) \in body(r)$ and if $head(r) = a'(\overline{x})$, $b'(\overline{x}) \in body(r)$.

Ordering annotations. Let $n \in \mathbb{N}$ and $A \subseteq \Sigma$ be a set of predicate symbols such that $|a| = 2n$ for all $a \in A$. An *ordering annotation* $ORD(A)$ represents that the transitive closure of the union of the relations induced by predicates in A given p's relaxed grounding is a strict partial order over $\mathbf{dom}^n$. The annotation $ORD(A)$ can be derived from the program p iff (1) there is no rule $r \in p$ that contains any of the predicates in A in its head and the transitive closure of $\bigcup_{a \in A} \{\langle \langle v_1, \ldots, v_n \rangle, \langle v_{n+1}, \ldots, v_{2n} \rangle \rangle \mid \exists k.\ k{::}a(v_1, \ldots, v_{2n}) \in p\}$ is a strict partial order over $\mathbf{dom}^n$, or (2) the predicates in A are derived from other predicates for which we can derive an ordering annotation. Hence, the closure of the relation $\bigcup_{a \in A} \{\langle \langle v_1, \ldots, v_n \rangle, \langle v_{n+1}, \ldots, v_{2n} \rangle \rangle \mid a(v_1, \ldots, v_{2n}) \in ground(p)\}$ induced by the relaxed grounding is a strict partial order. Formally, the annotation $ORD(A)$ can be derived from the program p iff one of the following conditions hold:

1. For all $pr \in A$, there is no rule r in p such that $pred(head(r)) = pr$ and $body(r) \neq \emptyset$, and the transitive closure of the relation $R = \bigcup_{pr \in A} \{\langle \overline{c}, \overline{v} \rangle \mid \exists r \in p.\, ((head(r) = pr(\overline{c}, \overline{v}) \vee \exists v'.\, head(r) = v'{::}pr(\overline{c}, \overline{v})) \wedge body(r) = \emptyset) \wedge |\overline{c}| = |\overline{v}| = {}^{|pr|}/_2\}$ is strict partial order.
2. The annotation $ORD(A')$ can be derived from p and there are two distinct predicate symbols $pr \in \Sigma$ and $pr' \in A'$ such that (a) $|pr| = |pr'|$, (b) $A = (A' \setminus \{pr'\}) \cup \{pr\}$, (c) $pr' \notin$

$\bigcup_{a\in A} reach(a)$, and (d) for all rules r in p such that $pred(head(r)) = pr$, there are sequences of variables $\overline{x}$ and $\overline{y}$ such that $head(r) = pr(\overline{x},\overline{y})$, $pr'(\overline{x},\overline{y}) \in body(r)$, and $|\overline{x}| = |\overline{y}|$.

Uniqueness annotations. Let $a \in \Sigma$ be a predicate symbol and $K \subseteq \{1,\dots,|a|\}$. A *uniqueness annotation* $UNQ(a,K)$ represents that the attributes in K are a primary key for the relation induced by a given the relaxed grounding. We say that $UNQ(a,K)$ can be derived from a program p iff no rule contains a in its head and for all $\overline{v},\overline{v}' \in \mathbf{dom}^{|a|}$, if (1) $\overline{v}(i) = \overline{v}'(i)$ for all $i \in K$, and (2) there are k and k' such $k{::}a(\overline{v}) \in p$ and $k'{::}a(\overline{v}') \in p$, then $\overline{v} = \overline{v}'$. This ensures that whenever $a(\overline{v}), a(\overline{v}') \in ground(p)$ and $\overline{v}(i) = \overline{v}'(i)$ for all $i \in K$, then $\overline{v} = \overline{v}'$. Alternatively, uniqueness annotations can be derived from other uniqueness annotations under appropriate conditions. Formally, the annotation $UNQ(pr,K)$ can be derived from p iff one of the following conditions hold:

1. There are no rules $r \in p$ such that $body(r) \neq \emptyset$ and $pred(head(r)) = pr$, and for all $\overline{t},\overline{v}$ in $\{\overline{c} \mid pr(\overline{c}) \in p \vee \exists v.\ v{::}pr(\overline{c}) \in p\}$, if $\overline{t}(i) = \overline{v}(i)$ for all $i \in K$, then $\overline{t} = \overline{v}$.
2. Both the following conditions hold:
   (a) For each rule $r \in p$ such that $head(r) = pr(\overline{x})$ and $body(r) \neq \emptyset$, there is a mapping μ' associating to each predicate pr' not in $reach(pr)$ a set K such that (a) the annotation $UNQ(pr',\mu(pr'))$ can be derived from p, and (b) $V \subseteq \bigcup_{l\in bound(head(r),K,body^+(r),\mu')} vars(l)$, where $V = \{\overline{x}(i) \mid i \notin K \wedge \overline{x}(i) \in Var\}$, $u(\overline{y},K) = \{\overline{y}(i) \mid i \in K\}$, and $bound(h,K,L,\mu') = \bigcup_{\substack{b(\overline{y})\in L\wedge b\notin reach(pr)\wedge b\neq pr\wedge \\ u(\overline{y},\mu'(b))\subseteq u(args(h),K)}} \{b(\overline{y})\} \cup \bigcup_{\substack{b(\overline{y})\in L\wedge b\notin reach(pr)\wedge b\neq pr\wedge \\ \exists l'\in bound(h,K,L,\mu').u(\overline{y},\mu'(b))\subseteq vars(l')}} \{b(\overline{y})\}$.
   (b) For all rules $r_1, r_2 \in p$, if $r_1 \neq r_2$, $pred(head(r_1)) = pred(head(r_2)) = pr$, and $K \neq \{1,\dots,|pr|\}$, then there is a value $i \in K$ such that $args(head(r_1))(i) \in \mathbf{dom}$, $args(head(r_2))(i) \in \mathbf{dom}$, and $args(head(r_1))(i) \neq args(head(r_1))(i)$.

Templates. A Σ-*template* $\mathcal{T}$ is a set of annotations.

Example 4.9. We can derive $DIS(D,E)$ from the program in Example 4.7 since no rule generates facts for D and E and the relations defined by the ground atoms are $\{1\}$ and $\{2\}$. We can also derive $ORD(\{O\})$ since the relation defined by O's ground atoms is $\{(1,2),(2,3)\}$, whose transitive closure is a strict partial order. Finally, we can derive $UNQ(O,\{1\})$, $UNQ(O,\{2\})$, and $UNQ(O,\{1,2\})$ since both arguments of O uniquely identify the tuples in the relation induced by O. ■

4.5.3 Acyclic ProbLog programs

A sufficient condition for tractable inference is that p's ground graph is a forest of poly-trees. This requires that p's ground graph neither contains directed nor undirected cycles, or, equivalently, the undirected version of p's ground graph is acyclic. To illustrate, the ground graph in Figure 4.7 is a poly-tree. The key insight here is that a cycle among ground atoms is caused by a (directed or undirected) cycle among p's predicate symbols. In a nutshell, acyclicity requires that all possible cycles in $graph(p)$ are guarded. This ensures that cycles in $graph(p)$ do not lead to cycles in the ground graph. Additionally, acyclicity requires that programs are negation-guarded. This ensures that the relaxed grounding and the ground graph are well-defined.

Unsafe structures. An unsafe structure models a part of the dependency graph that may introduce cycles in the ground graph. We define directed and undirected unsafe structures which may respectively introduce directed and undirected cycles in the ground graph.

A *directed unsafe structure* in $graph(p)$ is a directed cycle C in $graph(p)$. We say that a directed unsafe structure C *covers* a directed cycle C' iff C is equivalent to C'.

An *undirected unsafe structure* in $graph(p)$ is quadruple $\langle D_1,D_2,D_3,U\rangle$ such that (1) D_1, D_2, and D_3 are directed paths whereas U is an undirected path, (2) D_1 and D_2 start from the same node, (3) D_2 and D_3 end in the same node, and (4) $D_1 \cdot U \cdot D_3 \cdot D_2$ is an undirected cycle in $graph(p)$. We say that an unsafe structure $\langle D_1,D_2,D_3,U\rangle$ *covers* an undirected cycle U' in $graph(p)$ iff $D_1 \cdot U \cdot D_3 \cdot D_2$ is equivalent to U'.

Example 4.10. The cycle introduced by the rule r_c is captured by the directed unsafe structure $B \xrightarrow{r_c,1} B$, while the cycle introduced by r_a and r_b is captured by the structure $\langle A \xrightarrow{r_a,1} B, A \xrightarrow{r_b,1} B, \epsilon,\epsilon\rangle$, where ϵ denotes the empty path. ■

Connected Rules. A connected rule r ensures that a grounding of r is fully determined either by the assignment to the head's variables or to the variables of any literal in r's body. Formally, a strongly connected rule r guarantees that for any two groundings gr', gr'' of r, if $head(gr') = head(gr'')$, then $gr' = gr''$. In contrast, a weakly connected rule r guarantees that for any two groundings gr', gr'' of r, if $body(gr',i) = body(gr'',i)$ for some i, then $gr' = gr''$. This is done by exploiting uniqueness annotations and the rule's structure.

Before formalizing connected rules, we introduce join trees. A join tree represents how multiple predicate symbols in a rule share variables. In the following, let r be a rule and $\mathcal{T}$ be a template. A

join tree for a rule r is a rooted labelled tree $\langle N, E, root, \lambda\rangle$, where $N \subseteq body(r)$, E is a set of edges (i.e., unordered pairs over N^2), $root \in N$ is the tree's root, and λ is the labelling function. Moreover, we require that for all $n, n' \in N$, if $n \neq n'$ and $(n, n') \in E$, then $\lambda(n, n') = vars(n) \cap vars(n')$ and $\lambda(n, n') \neq \emptyset$. A join tree $\langle N, E, root, \lambda\rangle$ *covers* a literal l iff $l \in N$. Given a join tree $J = \langle N, E, root, \lambda\rangle$ and a node $n \in N$, the *support of* n, denoted $support(n)$, is the set $vars(head(r)) \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\} \cup \{vars(n') \mid n' \in anc(J, n)\}$, where $anc(J, n)$ is the set of n's ancestors in J, i.e., the set of all nodes (different from n) on the path from $root$ to n. A join tree $J = \langle N, E, root, \lambda\rangle$ is $\mathcal{T}$*-strongly connected* iff for all positive literals $l \in N$, there is a set $K \subseteq \{i \mid \overline{x} = args(l) \wedge \overline{x}(i) \in support(l)\}$ such that $UNQ(pred(l), K) \in \mathcal{T}$ and for all negative literals $l \in N$, $vars(l) \subseteq support(l)$. In contrast, a join tree $\langle N, E, root, \lambda\rangle$ is $\mathcal{T}$*-weakly connected* iff for all $(a(\overline{x}), a'(\overline{x}')) \in E$, there are $K \subseteq \{i \mid \overline{x}(i) \in L\}$ and $K' \subseteq \{i \mid \overline{x}'(i) \in L\}$ such that $UNQ(a, K), UNQ(a', K') \in \mathcal{T}$, where $L = \lambda(a(\overline{x}), a'(\overline{x}'))$.

We now formalize strongly and weakly connected rules. A rule r is $\mathcal{T}$*-strongly connected* iff there exist $\mathcal{T}$-strongly connected join trees $J_1, \ldots, J_n$ that cover all literals in r's body. This guarantees that for any two groundings gr', gr'' of r, if $head(gr') = head(gr'')$, then $gr' = gr''$.

Given a rule r, a set of literals L, and a template $\mathcal{T}$, a literal $l \in body(r)$ is $(r, \mathcal{T}, L)$*-strictly guarded* iff (1) $vars(l) \subseteq \bigcup_{l' \in L \cap body^+(r)} vars(l') \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\}$, and (2) there is a positive literal $a(\overline{x}) \in L$ and an annotation $UNQ(a, K) \in \mathcal{T}$ such that $\{\overline{x}(i) \mid i \in K\} \subseteq vars(l)$. A rule r is *weakly connected for* $\mathcal{T}$ iff there exists a $\mathcal{T}$-weakly connected join tree $J = \langle N, E, root, \lambda\rangle$ such that $N \subseteq body^+(r)$, and all literals in $body(r) \setminus N$ are $(r, \mathcal{T}, N)$-strictly guarded. This guarantees that for any two groundings gr', gr'' of r, if $body(gr', i) = body(gr'', i)$ for some i, then $gr' = gr''$.

Example 4.11. Let $\mathcal{T}$ be the template from Example 4.9. The rule $r_c := B(y) \leftarrow B(x), \neg F(x), O(x, y)$ is $\mathcal{T}$-strongly connected. Indeed, the join tree having $O(x, y)$ as root and $B(x)$ and $\neg F(x)$ as leaves is such that (1) there is a uniqueness annotation $UNQ(O, \{2\})$ in $\mathcal{T}$ such that the second variable in $O(x, y)$ is included in those of r_c's head, (2) the variables in $B(x)$ and $\neg F(x)$ are a subset of those of their ancestors, and (3) the tree covers all literals in r_c's body. The rule is also $\mathcal{T}$-weakly connected: the join tree consisting only of $O(x, y)$ is $\mathcal{T}$-weakly connected and the literals $B(x)$ and $\neg F(x)$ are strictly guarded. Note that the rules r_a and r_b are trivially both strongly and weakly connected. ■

Guarded undirected structures. Guarded undirected structures ensure that undirected cycles in the dependency graph do not correspond to undirected cycles in the ground graph by exploiting disjointness and uniqueness annotations. Formally, an undirected unsafe structure $\langle D_1, D_2, D_3, U\rangle$ is *guarded by a template* $\mathcal{T}$ iff either $\langle D_1, D_2\rangle$ is $\mathcal{T}$-head-guarded or $\langle D_2, D_3\rangle$ is $\mathcal{T}$-tail-guarded.

A pair of non-empty paths $\langle P_1, P_2\rangle$ sharing the same initial node a is $\mathcal{T}$*-head guarded* iff (1) if $P_1 = P_2$, all rules in P_1 are weakly connected for $\mathcal{T}$, and (2) if $P_1 \neq P_2$, there is an annotation $DIS(pr, pr') \in \mathcal{T}$, a set $K \subseteq \{1, \ldots, |a|\}$, and a bijection $\nu : K \to \{1, \ldots, |pr|\}$ such that P_1 ν-downward links to pr and P_2 ν-downward links to pr'. Given two ground paths P_1' and P_2' corresponding to P_1 and P_2, the first condition ensures that $P_1' = P_2'$ whereas the second ensures that P_1' or P_2' are not in the ground graph.

Similarly, a pair of non-empty paths $\langle P_1, P_2\rangle$ sharing the same final node a is $\mathcal{T}$*-tail guarded* iff (1) if $P_1 = P_2$, all rules in P_1 are strongly connected for $\mathcal{T}$, and (2) if $P_1 \neq P_2$, there is an annotation $DIS(pr, pr') \in \mathcal{T}$, a set $K \subseteq \{1, \ldots, |a|\}$, and a bijection $\nu : K \to \{1, \ldots, |pr|\}$, such that P_1 ν-upward links to pr and P_2 ν-upward links to pr'.

Example 4.12. The only non-trivially guarded undirected cycle in the graph from Figure 4.6 is the one represented by the undirected unsafe structure $\langle A \xrightarrow{r_a,1} B, A \xrightarrow{r_b,1} B, \epsilon, \epsilon\rangle$. The structure is guarded since the paths $A \xrightarrow{r_a,1} B$ and $A \xrightarrow{r_b,1} B$ are head guarded by $DIS(D, E)$. Indeed, for the same ground atom $A(v)$, for some $v \in \{1, 2, 3\}$, only one of r_a and r_b can be applied since D and E are disjoint. ■

Guarded directed structures. Guarded directed structures exploit ordering annotations to ensure that directed cycles in the dependency graph do not correspond to directed cycles among ground atoms. A directed unsafe structure $pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_n,i_n} pr_1$ is *guarded by a template* $\mathcal{T}$ iff there is an annotation $ORD(O) \in \mathcal{T}$, integers $1 \leq y_1 < y_2 < \ldots < y_e = n$, atoms $o_1(\overline{x}_1), \ldots, o_e(\overline{x}_e)$ (where $o_1, \ldots, o_e \in O$), a non-empty set $K \subseteq \{1, \ldots, |pr_1|\}$, and a bijection $\nu : K \to \{1, \ldots, {}^{|o|}/_2\}$ such that for each $0 \leq k < e$, (1) $pr_{y_k} \xrightarrow{r_{y_k}, i_{y_k}} \ldots \xrightarrow{r_{y_{k+1}-1}, i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν-downward connects to $o_{k+1}(\overline{x}_{k+1})$, and (2) $pr_{y_{k+1}-1} \xrightarrow{r_{y_{k+1}-1}, i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν'-upward connects to $o_{k+1}(\overline{x}_{k+1})$, where $\nu'(i) = \nu(x) + {}^{|o_1|}/_2$ for all $1 \leq i \leq {}^{|o_1|}/_2$, and $y_0 = 1$.

Example 4.13. The directed unsafe structure $B \xrightarrow{r_c,1} B$ is guarded by $ORD(\{O\})$ in the template from Example 4.9. Indeed, the strict partial order induced by O breaks the cycle among ground atoms belonging to B. In particular, the path $B \xrightarrow{r_c,1} B$ both downward links and upward links to $O(x, y)$; see Example 4.8. ■

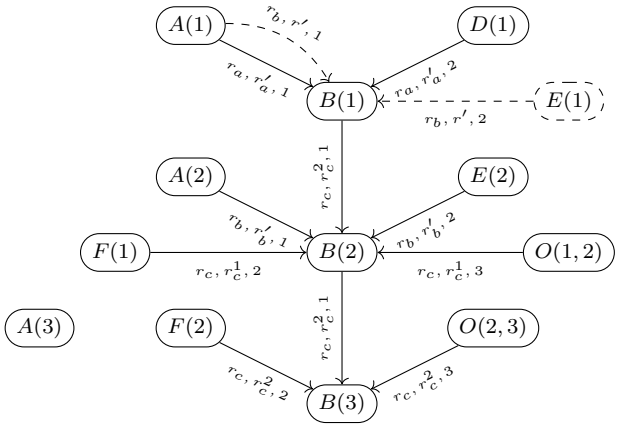
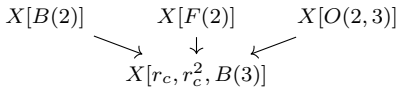


FIGURE 4.8: Ground graph for the program in Example 4.7 extended with the atom $E(1)$. The additional edges and nodes are represented using dashed lines. The ground rules r'_a, r'_b, r^1_c, and r^2_c are as in Figure 4.7, and $r' = B(1) \leftarrow A(1), E(1)$.



Selected CPTs.

$X[B(2)]$	$X[F(2)]$	$X[O(2,3)]$	$X[r_c, r^2_c, B(3)]$ $\top$	$\bot$
$\top$	$\top$	$\top$	0	1
$\top$	$\top$	$\bot$	0	1
$\top$	$\bot$	$\top$	1	0
$\top$	$\bot$	$\bot$	0	1
$\bot$	$\top$	$\top$	0	1
$\bot$	$\top$	$\bot$	0	1
$\bot$	$\bot$	$\top$	0	1
$\bot$	$\bot$	$\bot$	0	1

FIGURE 4.9: Portion of the resulting BN for the atoms $B(2)$, $F(2)$, and $O(2,3)$, the rule $r_c = B(y) \leftarrow B(x), \neg F(x), O(x,y)$, and the ground rule $r^2_c = B(3) \leftarrow B(2), \neg F(2), O(2,3)$, together with the CPT encoding r^2_c's semantics.

Acyclic Programs. Let p be a negation-guarded program and $\mathcal{T}$ be the template containing all annotations that can be derived from p. We say that p is *acyclic* iff (a) for all undirected cycles U in *graph*(p) that are not directed cycles, there is a $\mathcal{T}$-guarded undirected unsafe structure that covers U, and (b) for all directed cycles C in *graph*(p), there is a $\mathcal{T}$-guarded directed unsafe structure that covers C. This ensures the absence of cycles in the ground graph.

Proposition 4.1. *Let p be a* ProbLog *program. If p is acyclic, then the ground graph of p is a forest of poly-trees.*

Example 4.14. The program p from Example 4.7 is acyclic. This is reflected in the ground graph in Figure 4.7. The program $q = p \cup \{E(1)\}$, however, is not acyclic: we cannot derive $DIS(D,E)$ from q and the undirected unsafe structure $\langle A \xrightarrow{r_a,1} B, A \xrightarrow{r_b,1} B, \epsilon, \epsilon \rangle$ is not guarded. As expected, q's ground graph contains an undirected cycle between $A(1)$ and $B(1)$, as shown in Figure 4.8. ■

Expressiveness. Acyclicity trades some of ProbLog's expressiveness for a tractable inference procedure. Acyclic programs, nevertheless, can encode many relevant probabilistic models.

Proposition 4.2. *Any forest of poly-tree BNs can be represented as an acyclic* ProbLog *program.*

To clarify this proposition's scope, observe that poly-tree BNs are one of the few classes of BNs with tractable inference procedures. From Proposition 4.2, it follows that a large class of probabilistic models with tractable inference can be represented as acyclic programs. This supports our thesis that our syntactic constraints are not overly restrictive. In Section 4.9, we relax acyclicity to support a limited form of annotated disjunctions and rules sharing a part of their bodies, which are needed to encode the example from Section 4.4 and for Proposition 4.2. The proofs of all our results are in Appendix B.

4.5.4 Inference Engine

Our inference algorithm for acyclic ProbLog programs consists of three steps: (1) we compute the relaxed grounding of p (see Section 4.5.1), (2) we compile the relaxed grounding into a Bayesian Network (BN), and (3) we perform the inference using standard algorithms for poly-tree Bayesian Networks [105].

Encoding as BNs. We compile the relaxed grounding $ground(p)$ into the BN $bn(p)$. The boolean random variables in $bn(p)$ are as follows: (a) for each atom a in $ground(p)$ and ground literal a or $\neg a$ occurring in any $gr \in \bigcup_{r \in p} ground(p, r)$, there is a random variable $X[a]$, (b) for each rule $r \in p$ and each ground atom a such that there is a ground rule $gr \in ground(p, r)$ satisfying $a = head(gr)$, there is a random variable $X[r, a]$, and (c) for each rule $r \in p$, each ground atom a, and each ground rule $gr \in ground(p, r)$ such that $a = head(gr)$, there is a random variable $X[r, gr, a]$.

The edges in $bn(p)$ are as follows: (a) for each ground atom a, rule r, and ground rule gr, there is an edge from $X[r, gr, a]$ to $X[r, a]$ and an edge from $X[r, a]$ to $X[a]$, and (b) for each ground atoms a and b, rule r, and ground rule gr, there is an edge from $X[b]$ to $X[r, gr, a]$ if b occurs in gr's body as a positive or negative literal.

Finally, the Conditional Probability Tables (CPTs) of the variables in $bn(p)$ are as follows. The CPT of variables of the form $X[a]$ and $X[r, a]$ is just the OR of the values of their parents, i.e., the value is $\top$ with probability 1 iff at least one of the parents has value $\top$. For variables of the form $X[r, gr, a]$ such that $body(r) \neq \emptyset$, the variable's CPT encode the semantics of the rule r, i.e., the value of $X[r, gr, a]$ is $\top$ with probability 1 iff all positive literals have value $\top$ and all negative literals have value $\bot$. In contrast, for variables of the form $X[r, gr, a]$ such that $body(r) = \emptyset$, the variable has value $\top$ with probability v and $\bot$ with probability $1 - v$, where r is of the form $v::a$ (if $r = a$, then $v = 1$).

To ensure that the size of the CPT of variables of the form $X[r, a]$ is independent of the size of the relaxed grounding, instead of directly connecting variables of the form $X[r, gr, a]$ with $X[r, a]$, we construct a binary tree of auxiliary variables where the leaves are all variables of the form $X[r, gr, a]$ and the root is the variable $X[r, a]$. Figure 4.9 depicts a portion of the BN for the program in Example 4.7.

Complexity. We now introduce the main result of this section.

Theorem 4.1. *The data complexity of inference for acyclic* ProbLog *programs is* Ptime.

This follows from (1) the relaxed grounding and the encoding can be computed in Ptime in terms of data complexity, (2) the encoding ensures that, for acyclic programs, the resulting Bayesian Network is a forest of poly-trees, and (3) inference algorithms for poly-tree BNs [105] run in polynomial time in the BN's size. In Section 4.9, we extend our encoding to handle additional features such as (a limited class of) annotated disjunctions, whereas in Appendix B we prove its correctness and complexity.

4.6 Angerona

Angerona is a DBIC mechanism that provably secures databases against probabilistic inferences. Angerona is parametrized by an AtkLog model representing the attacker's capabilities and it leverages ProbLog's inference capabilities.

4.6.1 Checking Query Security

Algorithm 1 presents Angerona. It takes as input a system state $s = \langle db, U, P \rangle$, a history h, the current query q issued by the user u, a system configuration C, and an AtkLog model ATK formalizing the users' beliefs. Angerona checks whether disclosing the result of the current query q may violate any secrets in $secrets(P, u)$. If this is the case, the algorithm concludes that q's execution would be insecure and returns $\bot$. Otherwise, it returns $\top$ and authorizes q's execution. Note that once we fix a configuration C and an AtkLog model ATK, Angerona is a C-PDP as defined in Section 4.3.3.

To check whether a query q may violate a secret $\langle u, \psi, l \rangle \in secrets(P, u)$, Angerona first checks whether the secret has been already violated. If this is not the case, Angerona checks whether disclosing q violates any secret. This requires checking that u's belief about the secret ψ stays below the threshold independently of the result of the query q; hence, we must ensure that u's belief is below the threshold both in case the query q holds in the actual database and in case q does not hold (this ensures that the security decision itself does not leak information). Angerona, therefore, first checks whether there exists at least one possible database state where q is satisfied given h, using the procedure *pox*. If this is the case, the algorithm extends the current history h with the new event recording that the query q is authorized and its result is $\top$ and it checks whether u's belief about ψ is still below the corresponding threshold once q's result is disclosed, using the *secure*

Algorithm 1: ANGERONA Enforcement Algorithm.

Input: A system state $s = \langle db, U, P\rangle$, a history h, an action $\langle u, q\rangle$, a system configuration C, and a C-ATKLOG model ATK.
Output: The security decision in $\{\top, \bot\}$.

```
begin
    for ⟨u, ψ, l⟩ ∈ secrets(P, u) do
        if secure(C, ATK, h, ⟨u, ψ, l⟩)
            if pox(C, ATK, h, ⟨u, q⟩)
                h' := h · ⟨⟨u, q⟩, ⊤, ⊤⟩
                if ¬secure(C, ATK, h', ⟨u, ψ, l⟩)
                    return ⊥
            if pox(C, ATK, h, ⟨u, ¬q⟩)
                h' := h · ⟨⟨u, q⟩, ⊤, ⊥⟩
                if ¬secure(C, ATK, h', ⟨u, ψ, l⟩)
                    return ⊥
    return ⊤

function secure(⟨D, Γ⟩, ATK, h, ⟨u, ψ, l⟩)
    p := ATK(u)
    for φ ∈ observations(h, u) do
        p := p ∪ PL(φ) ∪ {evidence(head(φ), true)}
    p := p ∪ PL(ψ)
    return [[p]]_D(head(ψ)) < l

function pox(⟨D, Γ⟩, ATK, h, ⟨u, ψ⟩)
    p := ATK(u)
    for φ ∈ observations(h, u) do
        p := p ∪ PL(φ) ∪ {evidence(head(φ), true)}
    p := p ∪ PL(ψ)
    return [[p]]_D(head(ψ)) > 0
```

procedure. Afterwards, ANGERONA checks whether there exists at least a possible database state where q is not satisfied given h, it extends the current history h with another event representing that the query q does not hold, and it checks again whether disclosing that q does not hold in the current database state violates the secret. Note that checking whether there is a database state where q is (or is not) satisfied is essential to ensure that the conditioning that happens in the *secure* procedure is well-defined, i.e., the set of states we condition on has non-zero probability.

ANGERONA uses the *secure* subroutine to determine whether a secret's confidentiality is violated. This subroutine takes as input a system configuration, an ATKLOG model ATK, a history h, and a secret $\langle u, \psi, l\rangle$. It first computes the set $observations(h, u)$ containing all the authorized queries in the u-projection of h, i.e., $observations(h, u) = \{\phi \mid \exists i.\, h|_u(i) = \langle\langle u, \phi\rangle, \top, \top\rangle\} \cup \{\neg\phi \mid \exists i.\, h|_u(i) = \langle\langle u, \phi\rangle, \top, \bot\rangle\}$. Afterwards, it generates a PROBLOG program p by extending $ATK(u)$ with additional rules. In more detail, it translates each relational calculus sentence $\phi \in observations(h, u)$ to an equivalent set of PROBLOG rules $PL(\phi)$. The translation $PL(\phi)$ is standard [10]. For example, given a query $\phi = (A(1) \wedge B(2)) \vee \neg C(3)$, the translation $PL(\phi)$ consists of the rules $\{(h_1 \leftarrow A(1)), (h_2 \leftarrow B(2)), (h_3 \leftarrow \neg C(3)), (h_4 \leftarrow h_1, h_2), (h_5 \leftarrow h_3), (h_5 \leftarrow h_4)\}$, where $h_1, \ldots, h_5$ are fresh predicate symbols. We denote by $head(\phi)$ the unique predicate symbol associated with the sentence ϕ by the translation $PL(\phi)$. In our example, $head(\phi)$ is the fresh predicate symbol h_5. The algorithm then conditions the initial probability distribution $ATK(u)$ based on the sentences in $observations(h, u)$. This is done using *evidence statements*, which are special PROBLOG statements of the form $evidence(a, v)$, where a is a ground atom and v is either `true` or `false` (see Section 4.4.1). For each sentence $\phi \in observations(h, u)$, the program p contains a statement $evidence(head(\phi), \texttt{true})$. Finally, the algorithm translates ψ to a set of logic programming rules and checks whether ψ's probability is below the threshold l.

The *pox* subroutine takes as input a system configuration, an ATKLOG model ATK, a history h, and a query $\langle u, \psi\rangle$. It determines whether there is a database db' that satisfies ψ and complies with the history $h|_u$. Internally, the routine again constructs a PROBLOG program starting from ATK, $observations(h, u)$, and ψ. Afterwards, it uses the program to check whether the probability of ψ given $h|_u$ is greater than 0.

Given a run $\langle s, h\rangle$ and a user u, the *secure* and *pox* subroutines condition u's initial beliefs based on the sentences in $observations(h, u)$, instead of using the set $[\![r]\!]_{\sim_u}$ as in the ATKLOG semantics. The key insight is that, as we prove in Appendix B, the set of possible database states defined by the sentences in $observations(h, u)$ is equivalent to $[\![r]\!]_{\sim_u}$, which contains all database states derivable from the runs $r' \sim_u r$. This allows us to use PROBLOG to implement ATKLOG's semantics without explicitly computing $[\![r]\!]_{\sim_u}$.

Example 4.15. Let ATK be the attacker model in Example 4.4, u be the user *Mallory*, the database

state be $s_{\{\texttt{A,B,C}\}}$, where `Alice`, `Bob`, and `Carl` have cancer, and the policy P be the one from Example 4.2. Furthermore, let $q_1, \ldots, q_4$ be the queries issued by *Mallory* in Example 4.3. AngeRona permits the execution of the first two queries since they do not violate the policy. In contrast, it denies the execution of the last two queries as they leak sensitive information. ■

Confidentiality. As we prove in Appendix B, AngeRona provides the desired security guarantees for any AtkLog-attacker. Namely, it authorizes only those queries whose disclosure does not increase an attacker's beliefs in the secrets above the corresponding thresholds. AngeRona also provides precise completeness guarantees: it authorizes all secrecy-preserving queries. Informally, a query $\langle u, q\rangle$ is *secrecy-preserving* given a run r and a secret $\langle u, \psi, l\rangle$ iff disclosing the result of $\langle u, q\rangle$ in any run $r' \sim_u r$ does not violate the secret.

Theorem 4.2. *Let a system configuration C and a C-AtkLog model ATK be given, and let* AngeRona *be the C-PDP f.* AngeRona *provides data confidentiality with respect to C and $\lambda u \in \mathcal{U}.\ [\![ATK(u)]\!]_D$, and it authorizes all secrecy-preserving queries.*

Complexity. AngeRona's complexity is dominated by the complexity of inference. We focus our analysis only on data complexity, i.e., the complexity when only the ground atoms in the ProbLog programs are part of the input while everything else is fixed. A *literal query* is a query consisting either of a ground atom $a(\bar{c})$ or its negation $\neg a(\bar{c})$. We call an AtkLog model *acyclic* if all belief programs in it are acyclic. Furthermore, a *literal secret* is a secret $\langle U, \phi, l\rangle$ such that ϕ is a literal query. We prove in Appendix B that for acyclic AtkLog models, literal queries, and literal secrets, the ProbLog programs produced by the *secure* and *pox* subroutines are acyclic. We can therefore use our dedicated inference engine from Section 4.5 to reason about them. Hence, AngeRona can be used to protect databases in Ptime in terms of data complexity.

Theorem 4.3. *For all acyclic* AtkLog *attackers, for all literal queries q, for all runs r whose histories contain only literal queries and contain only secrets expressed using literal queries,* AngeRona*'s data complexity is* Ptime.

Discussion. Our tractability guarantees apply only to acyclic AtkLog models, literal queries, and literal secrets. Nevertheless, AngeRona can still handle relevant problems of interest. As stated in Section 4.5, acyclic models are as expressive as poly-tree Bayesian Networks, one of the few classes of Bayesian Networks with tractable inference. Hence, for many probabilistic models that cannot be represented as acyclic AtkLog models, exact probabilistic inference is intractable.

Literal queries are expressive enough to state simple facts about the database content. For instance, they can be used to formulate queries such as "does *Alice* have cancer?". More complex (non-literal) queries can be simulated using (possibly large) sequences of literal queries. Similarly, policies with non-literal secrets can be implemented as sets of literal secrets, and the Boole–Fréchet inequalities [89] can be used to derive the desired thresholds. In both cases, however, our completeness guarantees hold only for the resulting literal queries, not for the original ones.

Finally, whenever our tractability constraints are violated, AngeRona can still be used by directly using ProbLog's inference capabilities. In this case, one would retain the security and completeness guarantees (Theorem 4.2) but lose the tractability guarantees (Theorem 4.3).

4.6.2 Implementation and Empirical Evaluation

To evaluate the feasibility of our approach in practice, we implemented a prototype of AngeRona, available at [85]. The prototype implements our dedicated inference algorithm for acyclic ProbLog programs (Section 4.5), which computes the relaxed grounding of the input program p, constructs the Bayesian Networks BN, and performs the inference over BN using belief propagation [105]. For inference over BN, we rely on the GRRM library [154]. Observe that evidence statements in ProbLog are encoded by fixing the values of the corresponding random variables in the BN. Note also that computing the relaxed grounding of p takes polynomial time in terms of data complexity, where the exponent is determined by p's rules. A key optimization is to pre-compute the relaxed grounding and construct BN off-line. This avoids grounding p and constructing the (same) Bayesian Network for each query. In our experiments we measure this time separately.

We use our prototype to study AngeRona's efficiency and scalability. We run our experiments on a PC with an Intel i7 processor and 32GB of RAM. For our experiments, we consider the database schema from Section 4.3. For the belief programs, we use the ProbLog program given in Section 4.4, which can be encoded as an acyclic program whenever each person has at most one son. We evaluate AngeRona's efficiency and scalability in terms of the number of ground atoms in the belief programs. We generate synthetic belief programs containing 1,000 to 100,000 patients and for each of these instances, we generate 100 random queries of the form $R(\bar{t})$, where R is a predicate symbol and $\bar{t}$ is a tuple. For each instance and sequence of queries, we check the security of each query with our

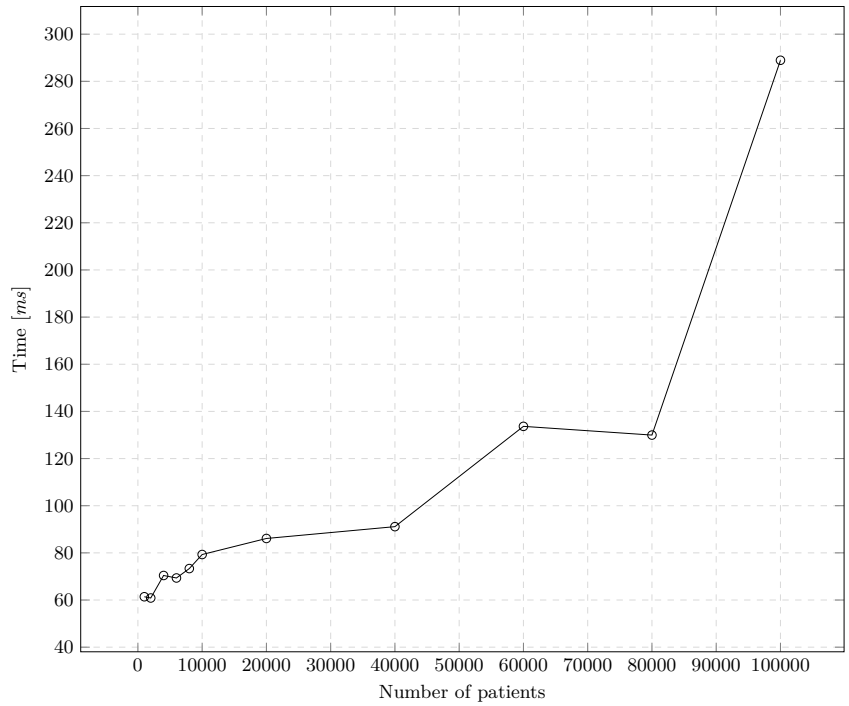


FIGURE 4.10: ANGERONA execution time in seconds.

prototype, against a policy containing 100 randomly generated secrets specified as literal queries. Note that in our experiments we repeated the above process 5 times, i.e., we generated 5 synthetic belief programs per data point and, for each of the programs, we generated random queries and policies.

Figure 4.10 reports the average execution times for our case study. Once the BN is generated, ANGERONA takes under 300 milliseconds on average, even for our larger examples, to check a query's security. During the initialization phase of our dedicated inference engine, we ground the original PROBLOG program and translate it into a BN. Most of the time is spent in the grounding process, whose data complexity is polynomial, where the polynomial's degree is determined by the number of free variables in the belief program. Our prototype uses a naive bottom-up grounding technique, and for our larger examples the initialization times are less than 2.5 minutes. We remark, however, that the initialization is performed just once per belief program. Furthermore, it can be done offline and its performance can be greatly improved by naive parallelization.

4.7 Related Work

Database Inference Control. Existing DBIC approaches protect databases (either at design time [120, 121] or at runtime [44, 45, 91]) only against restricted classes of probabilistic dependencies, e.g., those arising from functional and multi-valued dependencies. ATKLOG, instead, supports arbitrary probabilistic dependencies, and even our acyclic fragment can express probabilistic dependencies that are not supported by [44,45,91,120,121]. Wiese [166] proposes a DBIC framework, based on possibilistic logic, that formalizes secrets as sentences and expresses policies by associating bounds to secrets. Possibility theory differs from probability theory, which results in subtle differences. For instance, there is no widely accepted definition of conditioning for possibility distributions, cf. [38]. Thus, the probabilistic model from Section 4.1 cannot be encoded in Wiese's framework [166].

Statistical databases store information associated to different individuals and support queries that return statistical information about an entire population [47]. DBIC solutions for statistical databases [11,47,62,65,67] prevent leakages of information about single individuals while allowing the execution of statistical queries. These approaches rely on various techniques, such as perturbating the original data, synthetically generating data, or restricting the data on which the queries are executed. Instead, we protect specific secrets in the presence of probabilistic data dependencies and we return the original query result, without modifications, if it is secure.

Controlled Query Evaluation. *Controlled Query Evaluation* (CQE) is an inference control framework initially proposed by Biskup and Bonatti [28, 29, 36]. The framework has been specialized to several settings, such as deductive databases [36, 146], incomplete databases [33–35], databases in the

presence of updates [31,32], possibilistic databases [166], and ontologies [82]. In a CQE system, each query q is executed on the database and its result is inspected by a *censor*, which decides whether the query's result violates the security policy. In case the result is considered sensitive, the censor may reject the result [28,29] or modify it [36] (or a combination of both strategies [30]). Observe that the first strategy corresponds to the Non-Truman model, whereas the last one corresponds to the Truman model.

Works on CQE formalize security policies using secrecies and secrets. A *secret* is a sentence ϕ defined on the database, whereas a *secrecy* is a set $\{\phi, \neg\phi\}$ where ϕ is a secret. A policy then is either a set of secrets or a set of secrecies. Note that secrecies can be reduced to secrets. Our security policies from Section 4.3 can be seen as an extension of secrets to the probabilistic setting.

In a nutshell, the two security conditions studied in CQE are defined as follows:

- *Secrecies*: a censor is secure iff for all sequences of queries Q, for all database states db, for all secrecies $\{\phi, \neg\phi\}$ in S, there exists a state db' such that the results of all queries in Q on db and db' are the same, ϕ holds in one state, and $\neg\phi$ holds in the other one.
- *Secrets*: a censor is secure iff for all sequences of queries Q, for all database states db, for all secrets ϕ in S, there exists a state db' such that the results of all queries in Q on db and db' are the same, and in one of the two states $\neg\phi$ holds.

Intuitively, the security condition for secrecies guarantees that an attacker is not able to infer ϕ's truth value, whereas the one for secrets only guarantees that an attacker is not able to infer that ϕ holds in current database states. Observe that our data confidentiality property from Section 4.3 generalizes the latter condition to the probabilistic setting.

Differential Privacy. Differential Privacy [68,69] is widely used for privacy-preserving data analysis. Systems such as Airavat [132], ProPer [70], and PINQ [118] provide users with automated ways to perform differentially private computations. A differentially private computation guarantees that the presence (or absence) of an individual's data in the input data set affects the probability of the computation's result only in limited way, i.e., by at most a factor e^{ϵ} where ϵ is a parameter controlling the privacy-utility trade-off. While differential privacy does not make any assumption about the attacker's beliefs, we assume that the attacker's belief is known and we guarantee that for all secrets in the policy, no user can increase his beliefs, as specified in the attacker model, over the corresponding thresholds by interacting with the system.

Database Access Control. Rastogi et al. [129] propose an access control language for uncertain data and a perturbation algorithm to securely answer queries. Their security notion differs from ours, as it is based on differential privacy. To prevent leaks, their algorithm adds noise to the original query answer. Instead, Angerona returns the original result to the user if this does not violate the security policy.

Query Auditing. Query auditing [71,103,123] is the task of determining whether the answer to a user's query may lead to a privacy breach. Evfimievski et al.'s security notion [71] guarantees that for a *secure query*, a user cannot increase his confidence in the secret by observing the query's output. Instead, our security notion guarantees that users cannot use queries to increase their confidence in a secret above a given threshold. Our approach works for arbitrary relational calculus queries. In contrast, existing online query auditing approaches [103,123] are limited to restricted classes of queries, such as `COUNT` and `MAX` queries.

Information-flow Control. Quantified Information Flow [16,51,106,115] aims at quantifying the amount of information leaked by a program. Instead of measuring the amount of leaked information, we focus on restricting the information that an attacker may obtain about a set of given secrets.

Non-interference has been extended to consider probabilities [15,136,164] for concurrent programs. Our security notion, instead, allows those leaks that do not increase an attacker's beliefs in a secret above the threshold, and it can be seen as a probabilistic extension of *opacity* [142], which allows any leak except leaking whether the secret holds.

Mardziel et al. [116] present a general DBIC architecture, where users' beliefs are expressed as probabilistic programs, security requirements as threshold on these beliefs, and the beliefs are updated in response to the system's behaviour. Our work directly builds on top of this architecture. However, instead of using an imperative probabilistic language, we formalize beliefs using probabilistic logic programming, which provides a natural and expressive language for formalizing dependencies arising in the database setting, e.g., functional and multi-valued dependencies, as well as common probabilistic models, like Bayesian Networks.

Mardziel et al. [116] also propose a DBIC mechanism based on abstract interpretation. They do not provide any precise complexity bound for their mechanism. Their algorithm's complexity class, however, appears to be intractable, since they use a probabilistic extension of the polyhedra abstract domain, whose asymptotic complexity is exponential in the number of program variables [148]. In contrast, Angerona exploits our inference engine for acyclic programs to secure databases against

a practically relevant class of probabilistic inferences, and it provides precise tractability and completeness guarantees.

We now compare (unrestricted) AtkLog with the imperative probabilistic language used in [116]. AtkLog allows one to concisely encode probabilistic properties specifying relations between tuples in the database. For instance, a property like "the probability of $A(x)$ is $1/2^n$, where n is the number of tuples (x, y) in B" can be encoded as $1/2::A(x) \leftarrow B(x, y)$. Encoding this property as an imperative program is more complex; it requires a **for** statement to iterate over all variables representing tuples in B and an **if** statement to filter the tuples. In contrast to [116], AtkLog provides limited support for numerical constraints (as we support only finite domains). Mardziel et al. [116] formalize queries as imperative probabilistic programs. They can, therefore, also model probabilistic queries or the use of randomization to limit disclosure. While all these features are supported by AtkLog, our goal is to protect databases from attackers that use standard query languages like SQL. Hence, we formalize queries using relational calculus and ignore probabilistic queries. Similarly to [116], our approach can be extended to handle some uncertainty on the attackers' capabilities. In particular, we can associate to each user a finite number of possible beliefs, instead of a single one. However, like [116], we cannot handle infinitely many alternative beliefs.

Probabilistic Programming. Probabilistic programming is an active area of research [81]. Here, we position ProbLog with respect to expressiveness and inference. Similarly to [77, 116], ProbLog can express only discrete probability distributions, and it is less expressive than languages supporting continuous distributions [76, 80, 144]. Current exact inference algorithms for probabilistic programs are based on program analysis techniques, such as symbolic execution [76, 144] or abstract interpretation [116]. In this respect, we present syntactic criteria that *ensure* tractable inference for ProbLog. Sampson et al. [139] symbolically execute probabilistic programs and translate them to BNs to verify probabilistic assertions. In contrast, we translate ProbLog programs to BNs to perform exact inference and our translation is tailored to work together with our acyclicity constraints to allow tractable inference.

4.8 Conclusions

Effectively securing databases that store data with probabilistic dependencies requires an expressive language to capture the dependencies and a tractable enforcement mechanism. To address these requirements, we developed AtkLog, a formal language providing an expressive and concise way to represent attackers' beliefs while interacting with the system. We leveraged this to design Angerona, a provably secure DBIC mechanism that prevents the leakage of sensitive information in the presence of probabilistic dependencies. Angerona is based on a dedicated inference engine for a fragment of ProbLog where exact inference is tractable. We see these results as providing a foundation for building practical protection mechanisms, which include probabilistic dependencies, as part of real-world database systems.

4.9 Technical Details, Extensions, and Additional Examples

In Section 4.9.1, we introduce an extension of acyclic programs. We present a simple encoding from relational calculus queries to logic programs in Section 4.9.2. Afterwards, we show that the program associated with our medical example from Section 4.1 is a relaxed acyclic program (Section 4.9.3). Finally, in Section 4.9.4 we present an additional example from the genomic data domain.

4.9.1 Relaxed acyclic programs

Here we present an extension of acyclic programs, called *relaxed acyclic programs*. Relaxed acyclic programs support a restricted class of annotated disjunctions. They also support rules with overlapping bodies. Relaxed acyclic programs are needed to encode the example from Section 4.4 and for Proposition 4.2. We also generalize our BN encoding to relaxed acyclic programs. Observe that (1) acyclic programs are also relaxed acyclic programs, and (2) all our results extend to relaxed acyclic programs.

The key components of this new fragment are two syntactic transformations over ProbLog programs that preserve certain key aspects of the program's structure. Intuitively, a ProbLog program p is a *relaxed acyclic* program iff the program obtained from p by applying the transformations is an acyclic ProbLog program. Finally, we develop a procedure for compiling any relaxed acyclic ProbLog program into a poly-tree Bayesian Network. Since any acyclic program is a relaxed acyclic program as well, this procedure can be applied also to acyclic programs.

4.9.1.1 Rule Domination

Let r_1 and r_2 be two rules. Rule r_1 is *dominated* by rule r_2, written $r_1 \sqsubseteq r_2$, iff: (a) $head(r_1) = head(r_2)$, (b) $cstr(r_1) = cstr(r_2)$, (c) for all $1 \leq i \leq |body(r_1)|$, then $pred(body(r_1, i)) = pred(body(r_2, i))$ and $args(body(r_1, i)) = args(body(r_2, i))$. We extend the domination relation also to atoms and probabilistic atoms as follows: $a_1(\bar{c}_1) \sqsubseteq a_2(\bar{c}_1)$ iff $a_1 = a_2$ and $\bar{c}_1 = \bar{c}_2$, and $v_1{::}a_1(\bar{c}_1) \sqsubseteq v_2{::}a_2(\bar{c}_1)$ iff $v_1 = v_2$, $a_1 = a_2$, and $\bar{c}_1 = \bar{c}_2$.

Given a program p and a rule $r \in p$, $[r]_{\sqsubseteq p}$ denotes the set $\{r' \in p \mid r' \sqsubseteq r\}$. We say that a rule r is *maximal* in a program p, written $maximal(r, p)$, iff there is no rule $r' \in p$ such that $r \sqsubseteq r'$. The *kernel of* $[r]_{\sqsubseteq p}$, denoted $k([r]_{\sqsubseteq p})$, is the rule r' defined as follows: $head(r') = head(r)$, $|body(r')| = |body(r)|$, $cstr(r') = cstr(r)$, and for all $1 \leq i \leq |body(r')|$, then $body(r', i) = a_i(\bar{c}_i)$ if for all rules $r'' \in [r]_{\sqsubseteq p}$ (1) $|body(r'')| \geq i$ and (2) $body(r'', i)$ is a positive literal and $body(r', i) = \neg a_i(\bar{c}_i)$ otherwise, where $a_i = pred(body(r, i))$ and $\bar{c}_i = args(body(r, i))$.

The *maximal projection* of p, denoted $p{\Downarrow_{\sqsubseteq}}$, is $\{k([r]_{\sqsubseteq p}) \mid r \in p \wedge maximal(r, p)\}$. The syntactic transformation α, which removes all non-maximal rules, takes as input a ProbLog program p and returns as output its maximal projection $p{\Downarrow_{\sqsubseteq}}$.

4.9.1.2 Safe annotated disjunctions

We now present *safe annotated disjunctions* predicates, a special kind of probabilistic structures that can be formalized using annotated disjunctions. While our fragment does not support the unrestricted use of annotated disjunctions, as they may introduce cycles in the ground graph, safe annotated disjunctions can be still encoded as poly-trees (using non-boolean random variables).

Notation. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, pr be a predicate symbol in Σ of arity $|pr|$, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, and K be a set of distinct integer values such that for all $i \in K$, $1 \leq i \leq |pr|$. Given a tuple $\bar{t}$, we denote by $\bar{t}{\downarrow_{\{i_1,\ldots,i_n\}}}$ the tuple $\langle \bar{t}(i_1), \ldots, \bar{t}(i_n) \rangle$, where $i_1 \leq i_2 \leq \ldots \leq i_n$. The syntactic transformation $\Downarrow_{pr,K}$ is defined as follows:

- $(pr(\overline{x}))\Downarrow_{pr,K}$ is $pr(\overline{x}{\downarrow_{\{1,\ldots,|\overline{x}|\}\setminus K}})$,
- $(pr'(\overline{x}))\Downarrow_{pr,K}$ is $pr'(\overline{x})$, where $pr \neq pr'$,
- $(v{::}a(\overline{x}))\Downarrow_{pr,K}$ is $v{::}(a(\overline{x})\Downarrow_{pr,K})$,
- $(\neg a(\overline{x}))\Downarrow_{pr,K}$ is $\neg(a(\overline{x})\Downarrow_{pr,K})$, and
- $(h \leftarrow l_1, \ldots, l_n, c_1, \ldots, c_n)\Downarrow_{pr,K}$ is $h\Downarrow_{pr,K} \leftarrow l_1\Downarrow_{pr,K}, \ldots, l_n\Downarrow_{pr,K}, c_1, \ldots, c_n$, and
- $p\Downarrow_{pr,K}$, where p is a program, is $\{r\Downarrow_{pr,K} \mid r \in p\}$.

Given $(pr_1, K_1), \ldots, (pr_n, K_n)$, the transformation $\Downarrow_{(pr_1,K_1),\ldots,(pr_n,K_n)}$ is $\Downarrow_{pr_1,K_1} \circ \ldots \circ \Downarrow_{pr_n,K_n}$. Finally, given a partial function μ assigning to predicate symbols pr sets $K \subseteq \{1, \ldots, |pr|\}$, the transformation $\Downarrow_\mu$ is $\Downarrow_{(pr_1,\mu(pr_1)),\ldots,(pr_n,\mu(pr_n))}$, where $pr_1, \ldots, pr_n$ are all predicate symbols for which μ is defined.

Partitionings and CPT-schemas. Let p be a program.

An *horizontal partitioning* π_H is a function taking as input a program p and a predicate symbol pr and returning as output an element of $2^{2^{2^{rules(p)}}}$ such that the following conditions hold:

1. For all $RR \in \pi_H(p, pr)$ and for all $R \in RR$, $RR \neq \emptyset$ and $R \neq \emptyset$.
2. For all $RR \in \pi_H(p, pr)$, for all distinct R_1 and R_2 in RR, $R_1 \cap R_2 = \emptyset$.
3. For all distinct RR_1 and RR_2 in $\pi_H(p, pr)$, $\bigcup_{R_1 \in RR_1} R_1 \cap \bigcup_{R_2 \in RR_2} R_2 = \emptyset$.
4. $\bigcup_{RR \in \pi_H(p,pr)} \bigcup_{R \in RR} R = \{r \in rules(p) \mid pred(head(r)) = pr\}$.

A *vertical partitioning* π_V is a function that takes as input rules in p and returns as output a triple $\langle row, sel, sw \rangle$, where row, sel, and sw are sequences of literals, such that for each rule $r \in rules(p)$, $\pi_V(r) = \langle row, sel, sw \rangle$ and $row{\cdot}sel{\cdot}sw = body(r)$. A *CPT-schema* is a triple $\langle \pi_H, \pi_V, \mu \rangle$ such that π_H is an horizontal partitioning, π_V is a vertical partitioning, and μ is a total function associating to each predicate symbol pr in p a set in $2^{\mathbb{N}}$ such that $\mu(pr) \subseteq \{1, \ldots, |pr|\}$.

K-fixed domain predicates. Let p be a program, pr be a predicate symbol, and $K \subseteq \{1, \ldots, |pr|\}$ be a non-empty set. We say that pr is *K-fixed domain with respect to p* iff for all rules $r \in p$, the following conditions hold: (1) if $pred(head(r)) = pr$, then $args(head(r))\downarrow_K$ is a tuple in $\mathbf{dom}^{|K|}$, and (2) if $pred(body(r, j)) = pr$, for some $1 \leq j \leq |body(r)|$, then $args(body(r, j))\downarrow_K$ is a tuple in $\mathbf{dom}^{|K|}$. We denote by $pdom(pr, p, K)$ the set of $|K|$-tuples representing all constant values associated with the positions in K. Namely, $pdom(pr, p, K) = \{args(head(r))\downarrow_K \mid r \in p \wedge pred(head(r)) = pr\} \cup \{args(body(r, j))\downarrow_K \mid r \in p \wedge 1 \leq j \leq |body(r)| \wedge pred(body(r, j)) = pr\}$.

Switch predicates. Let p be a program. We say that a predicate symbol pr is a *switch predicate for p* iff (1) there is no rule in $rules(p)$ having pr in the head, (2) for any $v{::}pr(\bar{c}_1) \in p$, $v \neq 0$, and (3) for any two ground atoms $v_1{::}pr(\bar{c}_1) \in p$ and $v_2{::}pr(\bar{c}_2) \in p$, then $v_1 = v_2$.

Safe schemas. Let p be a program, $\langle \pi_H, \pi_V, \mu \rangle$ be a CPT-schema for p, and $\mathcal{T}$ be the template with all annotations that can be derived from p. We say that $\langle \pi_H, \pi_V, \mu \rangle$ is a *safe schema* iff for all pr in Σ such that $\mu(pr) \neq \emptyset$ the following conditions hold:

1. For all rules $r \in p$ such that $pred(head(r)) = pr$, r is strongly connected for $\mathcal{T}$ and $body(r) \neq \emptyset$. Note that this implies that there are no ground atoms $pr(\overline{c})$ in p.
2. The predicate symbol pr is $\mu(pr)$-fixed domain with respect to p.
3. For all $RR \in \pi_H(p, pr)$, the following conditions hold:
   (a) For all $R \in RR$, the following conditions hold:
      i. For any two distinct $r_1, r_2 \in R$, the following conditions hold:
         A. $row_1 = row_2$ and $sel_1 = sel_2$, where $\pi_V(r_1) = \langle row_1, sel_1, sw_1\rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, sw_2\rangle$.
         B. $\overline{x}_1{\downarrow_K} \neq \overline{x}_2{\downarrow_K}$ and $\overline{x}_1{\downarrow_{\{1,\ldots,|pr|\}\setminus K}} = \overline{x}_2{\downarrow_{\{1,\ldots,|pr|\}\setminus K}}$ hold, where $pr(\overline{x}_1) = head(r_1)$, $pr(\overline{x}_2) = head(r_2)$, and $K = \mu(pr)$.
      ii. One of the following conditions hold:
         A. There is an ordering $r_1, \ldots, r_n$ of the rules in R and switch predicates $sw_1, \ldots, sw_{n-1}$ such that (1) for all $1 \leq i < n$, $psw_i = \neg sw_1(\overline{x}_1), \ldots, \neg sw_{i-1}(\overline{x}_{i-1}), sw_i(\overline{x}_i)$, and $psw_n = \neg sw_1(\overline{x}_1), \ldots, \neg sw_{n-1}(\overline{x}_{n-1})$, where $\pi_V(r_i) = \langle row_i, sel_i, psw_i\rangle$ and $\overline{x}_j = args(head(r_j))$ for $1 \leq j \leq n$, and (2) $\sum_{1\leq i\leq n-1} p(i) = 1$, where $p(i) = v_i \cdot (1 - \sum_{1\leq j<i} p(j))$ and v_i is the probability associated to the predicate symbol sw_i in the program p.
         B. There is an ordering $r_1, \ldots, r_n$ of the rules in R and switch predicates $sw_1, \ldots, sw_n$ such that (1) for all $1 \leq i \leq n$, $psw_i = \neg sw_1(\overline{x}_1), \ldots, \neg sw_{i-1}(\overline{x}_{i-1}), sw_i(\overline{x}_i)$, where $\pi_V(r_i) = \langle row_i, sel_i, psw_i\rangle$ and $\overline{x}_j = args(head(r_j))$ for $1 \leq j \leq n$, and (2) $\sum_{1\leq i\leq n} p(i) = 1$, where $p(i) = v_i \cdot (1 - \sum_{1\leq j<i} p(j))$ and v_i is the probability associated to the predicate symbol sw_i in the program p.
      iii. For all $r \in R$, $pred(head(r)) = pr$, $args(head(r)){\downarrow_{\mu(pr)}} \in pdom(pr, p, \mu(pr))$, and $vars(head(r)) \subseteq \bigcup_{l\in row\cap body^+(r)} vars(l)$, where $\pi_V(r) = \langle row, sel, sw\rangle$.
   (b) For any two distinct $R_1, R_2 \in RR$, for all $r_1 \in R_1$ and $r_2 \in R_2$, where $\pi_V(r_1) = \langle row_1, sel_1, sw_1\rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, sw_2\rangle$, the following conditions hold:
      i. $args(head(r_1)){\downarrow_{\{1,\ldots,|pr|\}\setminus\mu(pr)}} = args(head(r_2)){\downarrow_{\{1,\ldots,|pr|\}\setminus\mu(pr)}}$.
      ii. For all $1 \leq i \leq min(|row_1|, |row_2|)$, the following conditions hold:
         A. $pred(row_1(i)) = pred(row_2(i))$.
         B. $args(row_1(i)){\downarrow_{K_{1,i}}} = args(row_2(i)){\downarrow_{K_{2,i}}}$, where $K_{1,i} = \{1, \ldots, |pred(row_1(i))|\} \setminus \mu(pred(row_1(i)))$ and $K_{2,i} = \{1, \ldots, |pred(row_2(i))|\} \setminus \mu(pred(row_2(i)))$.
      iii. There exists a $1 \leq i \leq min(|row_1|, |row_2|)$ such that one of the following conditions hold:
         A. $\mu(pred(row_1(i))) \neq \emptyset$, $row_1(i)$ is a positive literal, $row_2(i)$ is a positive literal, and $args(row_1(i)){\downarrow_{K_{1,i}}} \neq args(row_2(i)){\downarrow_{K_{2,i}}}$, where $K_{1,i} = \mu(pred(row_1(i)))$ and $K_{2,i} = \mu(pred(row_2(i)))$.
         B. $\mu(pred(row_1(i))) = \emptyset$, $row_1(i)$ is a positive literal, and $row_2(i)$ is a negative one or vice versa.
   (c) The set $\{k([r]_{\sqsubseteq p}) \mid r \in \bigcup_{R\in RR} R \wedge maximal(r, \bigcup_{R\in RR} R)\}$ has cardinality 1.
4. For any two distinct RR_1 and RR_2 in $\pi_H(p, pr)$, the following conditions hold:
   (a) For all $R_1 \in RR_1$, $R_2 \in RR_2$, $r_1 \in R_1$, and $r_2 \in R_2$, $args(head(r_1)){\downarrow_{\{1,\ldots,|pr|\}\setminus\mu(pr)}} = args(head(r_2)){\downarrow_{\{1,\ldots,|pr|\}\setminus\mu(pr)}}$.
   (b) There is a disjointness annotation $DIS(a, b) \in \mathcal{T}$ and indexes i, j such that for all $R_1 \in RR_1$ and $R_2 \in RR_2$, all $r_1 \in R_1$ and all $r_2 \in R_2$, $a(args(head(r_1)){\downarrow_{\{1,\ldots,|pr|\}\setminus\mu(pr)}}) = body^+(r_1, i)$ and $b(args(head(r_2)){\downarrow_{\{1,\ldots,|pr|\}\setminus\mu(pr)}}) = body^+(r_2, j)$.
   (c) The set $\{k([r]_{\sqsubseteq p}) \mid r \in \bigcup_{R_1\in RR_1} R_1 \cup \bigcup_{R_2\in RR_2} R_2 \wedge maximal(r, \bigcup_{R_1\in RR_1} R_1 \cup \bigcup_{R_2\in RR_2} R_2)\}$ has cardinality 2.

Syntactic transformation. Let p be a program and $\langle \pi_H, \pi_V, \mu\rangle$ be a CPT-schema for p. The syntactic transformation $\beta_{\langle \pi_H, \pi_V, \mu\rangle}$ takes as input a ProbLog program p and it returns as output the program $p' := p{\Downarrow_{(pr_1,\mu(pr_1)),\ldots,(pr_n,\mu(pr_n))}}$, where $pr_1, \ldots, pr_n$ are the predicate symbols in μ's domain. Note that p' is defined on a different database schema than p. The program p' is defined over the reduced database schema obtained by applying β to the atoms in p.

4.9.1.3 Relaxed Acyclic ProbLog programs

Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program. An *acyclicity witness for a program p* is safe CPT-schema $W = \langle \pi_H, \pi_V, \mu\rangle$ such that $\alpha(\beta_W(p))$ is an acyclic ProbLog program. We say that p is a *relaxed acyclic* $(\Sigma, \mathbf{dom})$-ProbLog *program* iff there is an acyclicity witness $W = \langle \pi_H, \pi_V, \mu\rangle$ for p. We remark that an acyclic program is also a relaxed acyclic program (for any assignment $\langle \pi_H, \pi_V, \mu\rangle$ such that $\mu(pr) = \emptyset$ for any pr). Given a relaxed acyclic $(\Sigma, \mathbf{dom})$-ProbLog program p and a

witness W, we associate to each rule r in $\alpha(\beta_W(p))$, the set $[r]_{p,W}$ of rules in p defined as follows:

$$[r]_{p,W} := \bigcup_{r' \in [r]_{\sqsubseteq \beta_W(p)}} \{r'' \in p \mid \beta_W(r'') = r'\}$$

The set $[r]_{p,W}$ contains all rules in p that are represented by the rule r in $\alpha(\beta_W(p))$.

4.9.1.4 Notation

Let p be a relaxed acyclic ProbLog program and $W = \langle \pi_H, \pi_V, \mu \rangle$ be a witness for p. For each atom $a(\overline{x})$, we denote by $a(\overline{c})\downarrow_p$ the atom obtained by applying the transformations α and β_W, whereas we denote by $a(\overline{x})\uparrow_p$ the constants not used in the transformed program, namely $a(\overline{x})\uparrow_p$ is $\overline{x}\downarrow_{\mu(a)}$. Furthermore, given a rule $r \in p$ (respectively a ground rule $s \in \mathit{ground}(p, r)$), we denote by $r\downarrow_p$ (respectively $s\downarrow_p$) the rule obtained by applying the transformations α and β_W. If $\mu(a) = \emptyset$, then $a(\overline{x})\uparrow_p = \top$ and $\neg a(\overline{x})\uparrow_p = \bot$.

4.9.1.5 Compilation to Bayesian Networks

Given a relaxed acyclic ProbLog program p, its encoding as a Bayesian Network $BN = \langle N, E, D, CPT, \preceq \rangle$, denoted $bn(p)$, is defined by Algorithm 2. We remark that this encoding extends the one we presented in Section 4.5 in two main aspects:

1. it supports safe annotated disjunctions by using non-boolean random variables (instead of just using boolean random variables), and
2. it supports rules with overlapping bodies by encoding the semantics of multiple rules in the conditional probability table of a single random variable (see the CPT auxiliary function in Algorithm 4).

For simplicity, in the following we assume given the total ordering $\preceq$ over random variables in N. Furthermore, we do not explicitly refer to each variable's domain, as it can be immediately derived from CPT. Hence, we refer to a Bayesian Network as a triple $\langle N, E, CPT \rangle$ instead of a 5-tuple $\langle N, E, D, CPT, \preceq \rangle$. In Algorithm 2, given a ground rule $s = h \leftarrow l_1, \ldots, l_n, e_1, \ldots, e_m$, we denote by $\mathit{pos}(s)$ the rule $h \leftarrow \mathit{atom}(l_1), \ldots, \mathit{atom}(l_n), e_1, \ldots, e_m$, where each negative literal is replaced with the corresponding positive literal.

4.9.2 From relational calculus to logic programs

In the following, let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema. Without loss of generality, we focus only on relational calculus formulae ϕ where no distinct pair of quantifiers binds the same variable.

Normal Form. We say that a formula $\psi \wedge \neg\gamma$ is *guarded* iff $\mathit{free}(\gamma) \subseteq \mathit{free}(\psi)$. We say that a relational calculus formula ϕ is in *Normal Form* (NF) iff (1) ϕ uses only existential quantifiers, (2) negation is used only in sub-formulae of the form $\psi \wedge \neg\gamma$ and it is always guarded, (3) for any sub-formula $\psi \vee \gamma$, $\mathit{free}(\psi) = \mathit{free}(\gamma)$, (4) no distinct pair of quantifiers binds the same variable, and (5) there are no equality and inequality constraints.

Most of the time, domain-independent relational calculus formulae can be easily written in NF by just re-arranging sub-formulae. We remark that any domain-independent relational calculus formula can be written in NF by (1) extending the database schema with two relations *eq* and *neq* encoding $=$ and $\neq$ among constants in **dom** (this is always possible because **dom** is finite), (2) renaming the quantified variables in a unique way, (3) replacing all universally quantified sub-formula $\forall x.\phi_e \rightarrow \psi$ with the equivalent existentially quantified version $\neg\exists x.\, \phi_e \wedge \neg\psi$, (4) replacing each negated sub-formula $\neg\psi$ with the equivalent sub-formula $(\bigwedge_{x \in \mathit{free}(\psi)} \mathit{adom}(x)) \wedge \neg\psi$, where $\mathit{adom}(x)$ is $\bigvee_{R \in D} \bigvee_{1 \leq i \leq |R|} \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{|R|}.\, R(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{|R|})$, and (5) replacing sub-formulae of the form $\psi \vee \gamma$ with the equivalent formula $(\bigwedge_{x \in \mathit{free}(\gamma) \setminus \mathit{free}(\psi)} \mathit{adom}(x) \wedge \psi) \vee (\bigwedge_{x \in \mathit{free}(\psi) \setminus \mathit{free}(\gamma)} \mathit{adom}(x) \wedge \gamma)$. Note that the resulting NF formula is equivalent to the original one (because the rewriting does not modify the formula's semantics). Therefore, in the following we consider only NF relational calculus formulae.

From Relational Calculus to Logic Programming. Let ϕ be a NF relational calculus sentence. We denote by $\mathit{sub}(\phi)$ the sequence $\phi_1, \ldots, \phi_n$ of all ϕ's sub-formulae ordered from the smallest to the largest, i.e., $\phi_n = \phi$, and by $\mathit{DIsub}(\phi)$ the sequence obtained by removing from the sequence $\mathit{sub}(\phi)$ all formulae of the form $\neg\psi$. Since ϕ is in NF, all negated sub-formulae in ϕ appear only in NF in the sequence $\mathit{DIsub}(\phi)$.

The function $PL(\phi)$ encodes any NF relational calculus sentence as a set of equivalent logic programming rules. We associate a unique predicate symbol H_i and a set of rules $r(H_i)$ to each ψ_i in $\mathit{DIsub}(\phi)$ as follows:

- If $\psi_i := x_i \neq x_j$, then $r(H_i)$ is $\{H_i(\overline{x}_i) \leftarrow \mathit{neq}(x_i, x_j)\}$.

Algorithm 2: Constructing the Bayesian Network. The sub-routines CPT is shown in Algorithm 3, whereas the sub-routines *tree* and $CPT_{\oplus}$ are shown in Algorithm 4.

Input: A relaxed acyclic $(\Sigma, \mathbf{dom})$-ProbLog program p_0 and a witness $W = \langle \pi_H, \pi_V, \mu' \rangle$ for p_0.
Output: A Bayesian Network $\langle N, E, CPT \rangle$.

▷Transform the original program.
$p = \alpha(\beta_W(p_0))$
▷Initialize the domain of each predicate symbol.
$D = \emptyset$
for $pr \in \Sigma$ **do**
  if $|\mu'(pr)| > 0$ **then**
    $D(pr) = pdom(pr, p_0, \mu(pr)) \cup \{\bot\}$
  else
    $D(pr) = \{\top, \bot\}$
▷Initialize the BN.
$N = \emptyset$
$E = \emptyset$
$CPT = \emptyset$
▷Creates the nodes.
for $r \in p$ **do**
  for $r' \in ground(p, r)$ **do**
    $N = N \cup \{X[head(r')]\}$
    for $a(\overline{c}) \in body^{+}(r')$ **do**
      $N = N \cup \{X[a(\overline{c})]\}$
    for $\neg a(\overline{c}) \in body^{-}(r')$ **do**
      $N = N \cup \{X[a(\overline{c})]\}$
for $r \in p$ **do**
  $\mu = \emptyset$
  for $s \in ground(p, r)$ **do**
    if $head(s) \notin dom(\mu)$ **then**
      $\mu = \mu[head(s) \mapsto \emptyset]$
    $\mu(head(s)) = \mu(head(s)) \cup \{pos(s)\}$
  for $a(\overline{c}) \in dom(\mu)$ **do**
    $N = N \cup \{X[r, a(\overline{c})]\}$
    $E = E \cup \{X[r, a(\overline{c})] \to X[a(\overline{c})]\}$
    $K = \emptyset$
    for $I \in \mu(a(\overline{c}))$ **do**
      $K = K \cup \{X[r, I, a(\overline{c})]\}$
      if $I = \emptyset$ **then**
        ▷Here, $[r]_{p_0,W} = \{r\}$ and $D(a) = \{\top, \bot\}$.
        if $\exists v.\ head(r) = v{::}a(\overline{c})$ **then**
          $prob = v$
        else
          $prob = 1$
        $CPT(X[r, \emptyset, a(\overline{c})]) = (\top \mapsto prob, \bot \mapsto (1 - prob))$
      else
        for $b(\overline{v}) \in I$ **do**
          $E = E \cup \{X[b(\overline{v})] \to X[r, I, a(\overline{c})]\}$
        for $\neg b(\overline{v}) \in I$ **do**
          $E = E \cup \{X[b(\overline{v})] \to X[r, I, a(\overline{c})]\}$
        $CPT(X[r, I, a(\overline{c})]) = CPT(I, r, D, W, p_0)$
    $\langle N', E', CPT' \rangle = tree(K, X[r, a(\overline{c})], D(a))$
    $N = N \cup N'$
    $E = E \cup E'$
    $CPT = CPT \cup CPT'$
▷Set the CPT for the variables associated to the atoms.
for $X[a(\overline{c})] \in N$ **do**
  $CPT(X[a(\overline{c})]) = CPT_{\oplus}(|\{X[r, a(\overline{c})] \in N\}| + 1, D(a))$
return $\langle N, E, CPT \rangle$

Algorithm 3: Auxiliary functions used in Algorithm 2.

function $CPT((a_1(\overline{c}_1), \ldots, a_n(\overline{c}_n)), r, D, W, p_0)$
  $A = D(a_1) \times \ldots \times D(a_n) \times D(pred(head(r)))$
  $CPT = \emptyset$
  for $\overline{a} \in A$ **do**
    $\overline{a}' = removeDuplicates((a_1(\overline{c}_1), \ldots, a_n(\overline{c}_n)), \overline{a})$
    if $satisfiable(r, \overline{a}, D, W, p_0) \wedge \forall 1 \leq i, j \leq n.\ ((i \neq j \wedge a_i(\overline{c}_i) = a_j(\overline{c}_j)) \rightarrow \overline{a}(i) = \overline{a}(j))$ **then**
      $CPT = CPT \cup \{\overline{a}' \mapsto 1\}$
    else
      $CPT = CPT \cup \{\overline{a}' \mapsto 0\}$
  return CPT

function $satisfiable(r, (v_1, \ldots, v_n), D, W, p_0)$
  if $v_n \neq \bot$ **then**
    ▷At least one satisfiable assignment.
    $res = \bot$
    for $r' \in [r]_{p_0, W}$ **do**
      if $filter(head(r'), D, \mu) = v_n$ **then**
        $sat = \top$
        for $1 \leq i \leq |body(r')|$ **do**
          if $v_i \notin filter(body(r', i), D, \mu)$ **then**
            $sat = \bot$
        $res = res \vee sat$
    return res
  if $v_n = \bot$ **then**
    ▷All assignments must be unsatisfiable.
    $res = \bot$
    for $r' \in [r]_{p_0, W}$ **do**
      $sat = \top$
      for $1 \leq i \leq |body(r')|$ **do**
        if $v_i \notin filter(body(r', i), D, \mu)$ **then**
          $sat = \bot$
      $res = res \vee sat$
    return $\neg res$

function $filter(l, D, \langle \pi_H, \pi_V, \mu \rangle)$
  if $\exists a \in \Sigma, \overline{x} \in (Var \cup \mathbf{dom})^{|a|}.\ l = a(\overline{x})$ **then**
    if $\mu(pred(l)) \neq \emptyset$ **then**
      return $\{args(l)\downarrow_{\mu(pred(l))}\}$
    else
      return $\{\top\}$
  if $\exists a \in \Sigma, \overline{x} \in (Var \cup \mathbf{dom})^{|a|}.\ l = \neg a(\overline{x})$ **then**
    if $\mu(pred(l)) \neq \emptyset$ **then**
      return $(D(pred(l)) \setminus \{args(l)\downarrow_{\mu(pred(l))}\}) \cup \{\bot\}$
    else
      return $\{\bot\}$

Algorithm 4: Auxiliary functions used in Algorithm 2.

function $CPT_{\oplus}(n, D)$
- ▷If $D \neq \{\top, \bot\}$, then $n = 2$ (since the program is relaxed-acyclic).
- **if** $D \neq \{\top, \bot\} \wedge n = 2$ **then**
  - $CPT = \emptyset$
  - **for** $(v_1, v_2) \in D^2$ **do**
    - **if** $v_1 = v_2$ **then**
      - $CPT = CPT \cup \{(v_1, v_2) \mapsto 1\}$
    - **else**
      - $CPT = CPT \cup \{(v_1, v_2) \mapsto 0\}$
  - **return** CPT
- **if** $D = \{\top, \bot\}$ **then**
  - $E = \{\top, \bot\}^n$
  - $CPT = \emptyset$
  - **for** $(v_1, \ldots, v_n) \in E$ **do**
    - $K = \{v_i \mid 1 \leq i \leq n-1 \wedge v_i = \top\}$
    - **if** $(v_n = \top \wedge K \neq \emptyset) \vee (v_n = \bot \wedge K = \emptyset)$ **then**
      - $CPT = CPT \cup \{\overline{v} \mapsto 1\}$
    - **else**
      - $CPT = CPT \cup \{\overline{v} \mapsto 0\}$
  - **return** CPT

function $tree(\mathbb{X}, root, D)$
- $p = nil$
- $N = \mathbb{X}$
- $E = \emptyset$
- $CPT = \emptyset$
- $\mathbb{X}' = \emptyset$
- **if** $\mathbb{X} = \{x\}$ **then**
  - $N = N \cup \{root\}$
  - $E = E \cup \{x \to root\}$
  - $CPT(root) = CPT_{\oplus}(2, D)$
- **else**
  - **for** $v \in \mathbb{X}$ **do**
    - **if** $p = nil$ **then**
      - $p = v$
    - **else**
      - $\mathbb{X}' = \mathbb{X}' \cup \{X[p, v]\}$
      - $N = N \cup \{X[p, v]\}$
      - $E = E \cup \{p \to X[p, v], v \to X[p, v]\}$
      - $CPT(X[p, v]) = CPT_{\oplus}(3, D)$
      - $p = nil$
  - **if** $p \neq nil$ **then**
    - $\mathbb{X}' = \mathbb{X}' \cup \{p\}$
  - $\langle N', E', CPT' \rangle = tree(\mathbb{X}', root, D)$
- **return** $\langle N \cup N', E \cup E', CPT \cup CPT' \rangle$

- If $\psi_i := x_i = x_j$, then $r(H_i)$ is $\{H_i(\overline{x}_i) \leftarrow eq(x_i, x_j)\}$.
- If $\psi_i := R(\overline{x}_i)$, for some $R \in \Sigma$, then $r(H_i)$ is $\{H_i(\overline{x}_i) \leftarrow R(\overline{x}_i)\}$, where $\overline{x}_i$ are ψ_i's free variables.
- If $\psi_i := \psi_j \wedge \neg\psi_k$, then $r(H_i)$ is $\{H_i(\overline{x}_i) \leftarrow H_j(\overline{x}_j), \neg H_k(\overline{x}_k)\}$, where $\overline{x}_i$ are ψ_i's free variables, $\overline{x}_j$ are ψ_j's free variables, and $\overline{x}_k$ are ψ_k's free variables (note that $\overline{x}_i = \overline{x}_j$ and $\overline{x}_k \subseteq \overline{x}_j$).
- If $\psi_i = \psi_j \wedge \psi_k$, then $r(H_i)$ is $\{H_i(\overline{x}_i) \leftarrow H_j(\overline{x}_j), H_k(\overline{x}_k)\}$, where $\overline{x}_i$ are ψ_i's free variables, $\overline{x}_j$ are ψ_j's free variables, and $\overline{x}_k$ are ψ_k's free variables.
- If $\psi_i = \psi_j \vee \psi_k$, then $r(H_i)$ contains the rules $H_i(\overline{x}_i) \leftarrow H_j(\overline{x}_i)$ and $H_i(\overline{x}_i) \leftarrow H_k(\overline{x}_i)$ (note that $free(\psi_i) = free(\psi_j) = free(\psi_k)$).
- If $\psi_i = \exists x.\, \psi_j$, then $r(H_i)$ is $\{H_i(\overline{x}_i) \leftarrow H_j(\overline{x}_j)\}$, where $\overline{x}_i$ are ψ_i's free variables and $\overline{x}_j$ are ψ_j's free variables (i.e., those in $\overline{x}_i$ and x).

Additionally, we add ground atoms encoding the equality and inequality relations *eq* and *neq*. Furthermore, we denote by $head(\phi)$ the predicate symbol associated to ψ_m.

4.9.3 Acyclicity of the medical data example

We now show, step by step, that the ProbLog program we presented in Section 4.4, which captures the motivating example from Section 4.1, is a relaxed acyclic program.

Original program. For simplicity, we report here the rules defining the original ProbLog program.

$$
\begin{aligned}
&1/20::cancer(x) \leftarrow patient(x)\\
&5/19::cancer(x) \leftarrow smokes(x)\\
&3/14::cancer(y) \leftarrow father(x,y), cancer(x), mother(z,y), \neg cancer(z)\\
&3/14::cancer(y) \leftarrow father(x,y), \neg cancer(x), mother(z,y), cancer(z)\\
&3/7::cancer(y) \leftarrow father(x,y), cancer(x), mother(z,y), cancer(z)
\end{aligned}
$$

We do not fix the set of ground atoms. Instead, below we present a set of requirements for acyclicity.

Removing syntactic sugar. The first step is removing the syntactic sugar, which, in this case, consists of the probabilistic rules. We do so exactly as specified in Section 4.4. We therefore obtain the following program:

$$
\begin{aligned}
&1/20::sw_1(\_)\\
&5/19::sw_2(\_)\\
&3/14::sw_3(\_)\\
&3/7::sw_4(\_)\\
&cancer(x) \leftarrow patient(x), sw_1(x)\\
&cancer(x) \leftarrow smokes(x), sw_2(x)\\
&cancer(y) \leftarrow father(x,y), mother(z,y), cancer(x), \neg cancer(z), sw_3(y)\\
&cancer(y) \leftarrow father(x,y), mother(z,y), \neg cancer(x), cancer(z), sw_3(y)\\
&cancer(y) \leftarrow father(x,y), mother(z,y), cancer(x), cancer(z), sw_4(y)
\end{aligned}
$$

Some rewriting. The last three rules in the above program use the same predicate symbols in their first four literals. However, they differ on the last literal, since the first two rules use $sw_3(y)$ while the last one uses $sw_4(y)$. We thus rewrite the program in an equivalent way:

$$
\begin{aligned}
&1/20::sw_1(\_)\\
&5/19::sw_2(\_)\\
&3/14::sw_3(\_)\\
&3/7::sw_4(\_)\\
&cancer(x) \leftarrow patient(x), sw_1(x)\\
&cancer(x) \leftarrow smokes(x), sw_2(x)\\
&cancer(y) \leftarrow father(x,y), mother(z,y), cancer(x), \neg cancer(z), sw_3(y)\\
&cancer(y) \leftarrow father(x,y), mother(z,y), \neg cancer(x), cancer(z), sw_3(y)\\
&cancer(y) \leftarrow father(x,y), mother(z,y), cancer(x), cancer(z), sw_3(y), sw_4(y)\\
&cancer(y) \leftarrow father(x,y), mother(z,y), cancer(x), cancer(z), \neg sw_3(y), sw_4(y)
\end{aligned}
$$

In the above program, we replaced the rule $cancer(y) \leftarrow father(x,y), mother(z,y), cancer(x), cancer(z), sw_4(y)$ with the two rules $cancer(y) \leftarrow father(x,y), mother(z,y), cancer(x), cancer(z), sw_3(y), sw_4(y)$

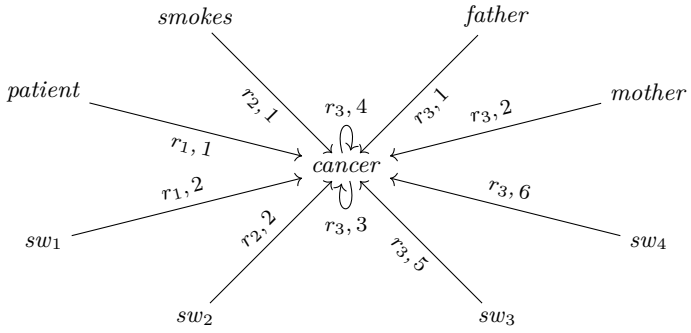


FIGURE 4.11: Dependency graph for the program capturing the motivating example in Section 4.1.

and $cancer(y) \leftarrow father(x, y), mother(z, y), cancer(x), cancer(z), \neg sw_3(y), sw_4(y)$. This does not modify the program's semantics.

Syntactic transformations. By applying the syntactic transformation for relaxed acyclic programs (see Section 4.9.1), we obtain the following program:

$$
\begin{aligned}
&{}^{1}/_{20}::sw_1(\_)\\
&{}^{5}/_{19}::sw_2(\_)\\
&{}^{3}/_{14}::sw_3(\_)\\
&{}^{3}/_{7}::sw_4(\_)\\
&cancer(x) \leftarrow patient(x), sw_1(x)\\
&cancer(x) \leftarrow smokes(x), sw_2(x)\\
&cancer(y) \leftarrow father(x, y), mother(z, y), \neg cancer(x), \neg cancer(z), \neg sw_3(y), sw_4(y)
\end{aligned}
$$

In the above transformation we considered a CPT-schema $\langle \pi_V, \pi_H, \mu \rangle$ such that $\mu(pr) = \emptyset$ for any predicate symbol. Note that such a CPT-schema is always safe.

The dependency graph of this program is depicted in Figure 4.11, where r_1 denotes $cancer(x) \leftarrow patient(x), sw_1(x)$, r_2 denotes the rule $cancer(x) \leftarrow smokes(x), sw_2(x)$, and r_3 denotes the rule $cancer(y) \leftarrow father(x, y), mother(z, y), \neg cancer(x), \neg cancer(z), \neg sw_3(y), sw_4(y)$.

Restrictions and annotations. To ensure that the parent-child relationship forms a poly-tree, we require that (1) each patient has at most one son, and (2) the relations induced by *father* and *mother* form a (strict) partial order. Furthermore, we require that the *father* and *mother* relations are disjoint. Whenever the ground atoms in the program satisfy these conditions, we can derive the annotations $UNQ(father, \{1\})$, $UNQ(father, \{2\})$, $UNQ(mother, \{1\})$, $UNQ(mother, \{2\})$, $ORD(\{father, mother\})$, and $DIS(father, mother)$ from the program.

Relaxed Acyclicity. We now show that the program is relaxed acyclic. First, we observe that all rules in the program are both weakly and strongly connected. Hence, all undirected structures of the form $\langle P, P, \epsilon, \epsilon \rangle$ and $\langle \epsilon, P, P, \epsilon \rangle$, where P is a directed path in the dependency graph, are guarded. Therefore, any undirected cycle involving predicates other than *cancer* is immediately guarded (see Proposition B.8 in Appendix B).

Second, consider an undirected cycle, which is not a directed cycle, that involves only the predicate symbol *cancer*. For simplicity, we consider only cycles consisting of 2 distinct paths, denoted P_1 and P_2, of length 1. There are two possible cases:

1. $P_1 = cancer \xrightarrow{r_3,3} cancer$ and $P_2 = cancer \xrightarrow{r_3,4} cancer$. The cycle is guarded by the structure $\langle cancer \xrightarrow{r_3,3} cancer, cancer \xrightarrow{r_3,4} cancer, \epsilon, \epsilon \rangle$ (since we can derive the annotation $DIS(father, mother)$).
2. $P_1 = cancer \xrightarrow{r_3,4} cancer$ and $P_2 = cancer \xrightarrow{r_3,3} cancer$. The cycle is guarded by the structure $\langle cancer \xrightarrow{r_3,4} cancer, cancer \xrightarrow{r_3,4} cancer, \epsilon, \epsilon \rangle$ (since we can derive the annotation $DIS(father, mother)$).

Hence, all undirected cycles consisting of two paths of length 1 are guarded. We can extend this result to all undirected cycles (without restrictions) using Proposition B.8 from Appendix B.

Third, consider the two directed cycles $C_1 = cancer \xrightarrow{r_3,3} cancer$ and $C_2 = cancer \xrightarrow{r_3,4} cancer$. Both are guarded, since we can derive $ORD(\{father, mother\})$. Furthermore, any directed cycle can be obtained only by combining C_1 and C_2. From this and Proposition B.9 from Appendix B, it follows that all directed cycles are guarded.

Mother	Father	Child CC	Child CT	Child TT
CC	CC	1	0	0
CC	CT	1/2	1/2	0
CC	TT	0	1	0
CT	CC	1/2	1/2	0
CT	CT	1/4	1/2	1/4
CT	TT	0	1/2	1/2
TT	CC	0	1	0
TT	CT	0	1/2	1/2
TT	TT	0	0	1

FIGURE 4.12: Probability distribution for a child's genome given his parents' genome, for a fixed position.

Since all directed and undirected cycles are guarded, the program is relaxed acyclic.

4.9.4 Genomic Data Example

We first present an overview of our example. Afterwards, we encode it in ProbLog, and we show how it can be represented as a relaxed acyclic program. Finally, we present some experimental results obtained with Angerona.

4.9.4.1 Overview

Genomic data is extremely sensitive. From an individual's genome, one can derive his predisposition to different diseases and his relationship to other persons. At the same time, genomic data is becoming increasingly available in hospitals, research centres, and online [1, 2, 5]. It is thus essential to understand how to secure genomic databases.

We now briefly summarize relevant aspects of genomics, see [42, 109] for an in-depth introduction to the topic. A *chromosome* is a string of DNA consisting of a sequence of complementary nucleotide pairs, where each pair consists either of Adenine (A) and Thymine (T), or Cytosine (C) and Guanine (G). Each cell's genetic material is stored in 23 pairs of chromosomes, where in each pair one of the chromosomes derives from the corresponding maternal chromosome and the other derives from the paternal one. Figure 4.12 illustrates the probabilistic model defined by Mendel's inheritance laws.

Humans have 99.9% of their DNA in common. Hence, what characterizes and differentiates us are the variations in our DNA. The most common variation is called Single Nucleotide Polymorphism (SNP), which occurs when a nucleotide (at a specific position) varies among the individuals in a population. SNPs are related to the susceptibility to various diseases, and therefore information about SNPs, precise or probabilistic, is sensitive. For instance, the SNP *rs11200638* in the HTRA1 promoter is associated with a 10-fold increased risk of age-related macular degeneration [104].

A simple example of a database for storing and querying patients' genomic data is as follows. The database contains: a table *patient* storing the patients' information; a table *doctor* storing the doctors' information; a table *patientof*, with two attributes *doctor* and *patient*, associating the doctors with the patients they treat; a table *genome*, with four attributes *patient*, *position*, *value1*, and *value2*, storing the genomic data in the form of SNPs; and tables *father* and *mother*, with two attributes *parent* and *child*, representing the parent-child relationships. This database is shared between different doctors and each doctor can access only the information associated with the patients he is treating. Furthermore, each doctor has access to the tables *father*, *mother*, and *patient*.

Controlling probabilistic inference is critical in this scenario. Restricting direct access to the genome of the patients not treated by a doctor, for instance by granting to each doctor d access to a view $genome_d$ disclosing only the genome of d's patients, is insufficient to prevent a malicious doctor from inferring sensitive information about patients that he is not treating. For instance, suppose that a malicious doctor d is treating both *Alice* and *Bob* but not *Carl*, which is *Alice* and *Bob*'s son. Suppose too that the nucleotide pair associated with the age-related macular disease is TT for *Alice* and CT for *Bob*. The doctor can use this information to infer that *Carl* has nucleotide TT with probability 50% for the position associated with age-related macular disease and is, therefore, 10 times more likely to suffer from the disease. Securing the database requires restricting direct and indirect access and reasoning about the probabilistic dependencies among the data.

4.9.4.2 Encoding

Original program. We encode the probability distribution associated with the Mendel's inheritance laws in ProbLog. We assume given the tables *patient*, *father*, and *mother* (denoted respectively P, F, and M). Furthermore, we assume that there are two auxiliary tables *withParents* and *noParents*

(denoted WP and NP respectively), which represent whether a patient's parents are in the database or not. Finally, we assume that the content of the above tables is the same in all possible states. The content of these tables can be encoded using ground non-probabilistic facts.

In contrast, the content of the *genome* table (denoted by G) follows the probability distribution shown in Figure 4.12. For simplicity, we assume that patients either have both parents in the database or neither of them, and in all possible database states, all patients have an entry in the *genome* table. For the patients without parents in the database, the three possible values are uniformly distributed. This is represented using the rule:

$$1/3::G(x,p,\texttt{C},\texttt{C});\ 1/3::G(x,p,\texttt{C},\texttt{T});\ 1/3::G(x,p,\texttt{T},\texttt{T}) \leftarrow P(x), NP(x)$$

In contrast, for patients with both parents in the database, the possible values are distributed according to Mendel's laws:

$$\begin{aligned}
G(x,p,\texttt{C},\texttt{C}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{C},\texttt{C}), M(m,x), G(m,p,\texttt{C},\texttt{C})\\
1/2::G(x,p,\texttt{C},\texttt{C});\ 1/2::G(x,p,\texttt{C},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{C},\texttt{C}), M(m,x), G(m,p,\texttt{C},\texttt{T})\\
G(x,p,\texttt{C},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{C},\texttt{C}), M(m,x), G(m,p,\texttt{T},\texttt{T})\\
1/2::G(x,p,\texttt{C},\texttt{C});\ 1/2::G(x,p,\texttt{C},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{C},\texttt{T}), M(m,x), G(m,p,\texttt{C},\texttt{C})\\
1/4::G(x,p,\texttt{C},\texttt{C});\ 1/2::G(x,p,\texttt{C},\texttt{T});\ 1/4::G(x,p,\texttt{T},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{C},\texttt{T}), M(m,x), G(m,p,\texttt{C},\texttt{T})\\
1/2::G(x,p,\texttt{C},\texttt{T});\ 1/2::G(x,p,\texttt{T},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{C},\texttt{T}), M(m,x), G(m,p,\texttt{T},\texttt{T})\\
G(x,p,\texttt{C},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{T},\texttt{T}), M(m,x), G(m,p,\texttt{C},\texttt{C})\\
1/2::G(x,p,\texttt{C},\texttt{T});\ 1/2::G(x,p,\texttt{T},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{T},\texttt{T}), M(m,x), G(m,p,\texttt{C},\texttt{T})\\
G(x,p,\texttt{T},\texttt{T}) &\leftarrow WP(x), F(f,x), G(f,p,\texttt{T},\texttt{T}), M(m,x), G(m,p,\texttt{T},\texttt{T})
\end{aligned}$$

In the rules, we use annotated disjunctions to encode the probability distribution, once the parents' genome is fixed.

Remove syntactic sugar. We now rewrite the program by removing all syntactic sugar, i.e., probabilistic rules and annotated disjunctions. We refer the reader to Section 4.4 for a discussion about how to encode these features in plain ProbLog. Note that we also perform some minor simplifications on the de-sugared program.

$$\begin{aligned}
&1/3::sw_1(_)\\
&1/2::sw_2(_)\\
&1/2::sw_3(_)\\
&1/2::sw_4(_)\\
&G(x,\texttt{C},\texttt{C}) \leftarrow P(x), NP(x), sw_1(x,\texttt{C},\texttt{C})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow P(x), NP(x), \neg sw_1(x,\texttt{C},\texttt{C}), sw_2(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{T},\texttt{T}) \leftarrow P(x), NP(x), \neg sw_1(x,\texttt{C},\texttt{C}), \neg sw_2(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{C},\texttt{C}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{C}), M(m,x), G(m,\texttt{C},\texttt{C})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{C}), M(m,x), G(m,\texttt{C},\texttt{T}), sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{C},\texttt{C}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{C}), M(m,x), G(m,\texttt{C},\texttt{T}), \neg sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{C}), M(m,x), G(m,\texttt{T},\texttt{T})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{C}), sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{C},\texttt{C}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{C}), \neg sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{T}), sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{C},\texttt{C}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{T}), \neg sw_3(x,\texttt{C},\texttt{T}), sw_4(x,\texttt{C},\texttt{C})\\
&G(x,\texttt{T},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{T}), \neg sw_3(x,\texttt{C},\texttt{T}), \neg sw_4(x,\texttt{C},\texttt{C})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{T}), M(m,x), G(m,\texttt{T},\texttt{T}), sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{T},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{C},\texttt{T}), M(m,x), G(m,\texttt{T},\texttt{T}), \neg sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{T},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{C})\\
&G(x,\texttt{C},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{T},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{T}), sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{T},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{T},\texttt{T}), M(m,x), G(m,\texttt{C},\texttt{T}), \neg sw_3(x,\texttt{C},\texttt{T})\\
&G(x,\texttt{T},\texttt{T}) \leftarrow WP(x), F(f,x), G(f,\texttt{T},\texttt{T}), M(m,x), G(m,\texttt{T},\texttt{T})
\end{aligned}$$

Syntactic transformations. By applying the syntactic transformation for relaxed acyclic programs (see Section 4.9.1), we obtain the following program (observe that the rules defining the predicate G

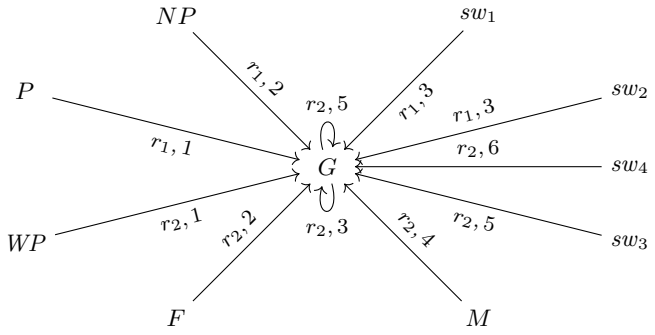


FIGURE 4.13: Dependency graph for the genomic data example.

are safe annotated disjunctions):

$$
\begin{aligned}
&1/3::sw_1(\_)\\
&1/2::sw_2(\_)\\
&1/2::sw_3(\_)\\
&1/2::sw_4(\_)\\
&\quad G(x) \leftarrow P(x), NP(x), \neg sw_1(x, \mathtt{C}, \mathtt{C}), \neg sw_2(x, \mathtt{C}, \mathtt{T})\\
&\quad G(x) \leftarrow WP(x), F(f, x), G(f), M(m, x), G(m), \neg sw_3(x, \mathtt{C}, \mathtt{T}), \neg sw_4(x, \mathtt{C}, \mathtt{C})
\end{aligned}
$$

The dependency graph of this program is shown in Figure 4.13, where r_1 denotes the rule $G(x) \leftarrow P(x), NP(x), \neg sw_1(x, \mathtt{C}, \mathtt{C}), \neg sw_2(x, \mathtt{C}, \mathtt{T})$ and r_2 denotes the rule $G(x) \leftarrow WP(x), F(f, x), G(f), M(m, x), G(m), \neg sw_3(x, \mathtt{C}, \mathtt{T}), \neg sw_4(x, \mathtt{C}, \mathtt{C})$.

Restrictions and annotations. We make the following assumptions on the set of ground atoms:

1. To ensure that the parent-child relationship forms a poly-tree, we require that (1) each patient has at most one son, and (2) the relations induced by F and M form a (strict) partial order.
2. We require that the F and M relations are disjoint.
3. We require that the WP and NP relations are disjoint. Furthermore, we require that these relations capture their intuitive semantics (e.g., a patient belongs to NP iff he has no parents in the database).

Whenever the ground atoms in the program satisfy these conditions, we can derive the annotations $UNQ(F, \{1\})$, $UNQ(F, \{2\})$, $UNQ(M, \{1\})$, $UNQ(M, \{2\})$, $ORD(\{F, M\})$, $DIS(F, M)$, and $DIS(WP, NP)$ from the program. Note that we can also derive other annotations, such as $UNQ(G, \{1\})$.

Relaxed Acyclicity. We can finally show that the program is relaxed acyclic. Observe that all rules in the program are both weakly and strongly connected. Hence, all undirected structures of the form $\langle P, P, \epsilon, \epsilon\rangle$ (and $\langle \epsilon, P, P, \epsilon\rangle$), where P is a directed path in the dependency graph, are guarded. Therefore, any undirected cycle involving predicates other than G is immediately guarded. Similar to Section 4.9.3, for undirected cycles that involve only G, the acyclicity directly follows from the annotations $DIS(F, M)$ and $DIS(WP, NP)$. Additionally, consider the two directed cycles $C_1 = G \xrightarrow{r_2,3} G$ and $C_2 = G \xrightarrow{r_2,5} G$. Both are guarded, since we can derive $ORD(\{F, M\})$. Furthermore, any directed cycle can be obtained only by combining C_1 and C_2. From this and Proposition B.9 from Appendix B, it follows that all directed cycles are guarded. Since all directed and undirected cycles are guarded, the program is relaxed acyclic.

4.9.4.3 Experiments

We run our experiments on a PC with an Intel i7 processor and 32GB of RAM. We consider the database schema and the probabilistic model we presented in the previous section. For our experiments, we generate synthetic belief programs containing 100 to 10,000 patients and for each of these instances, we generate 100 random literal queries extracting information from the *genome* table. For each instance and sequence of queries, we check the security of each query with our prototype, against a policy containing 100 randomly generated secrets specified as literal queries (note that the secrets are specified against the *genome* table). Note that in our experiments we repeated the above process 5 times, i.e., we generated 5 synthetic belief programs per data point and, for each of the programs, we generated random queries and policies. The average execution time for our experiments is shown in Figure 4.14. After the generation of the BN, ANGERONA takes under 100 milliseconds on average, even for our larger examples, to check a query's security, whereas the grounding time is

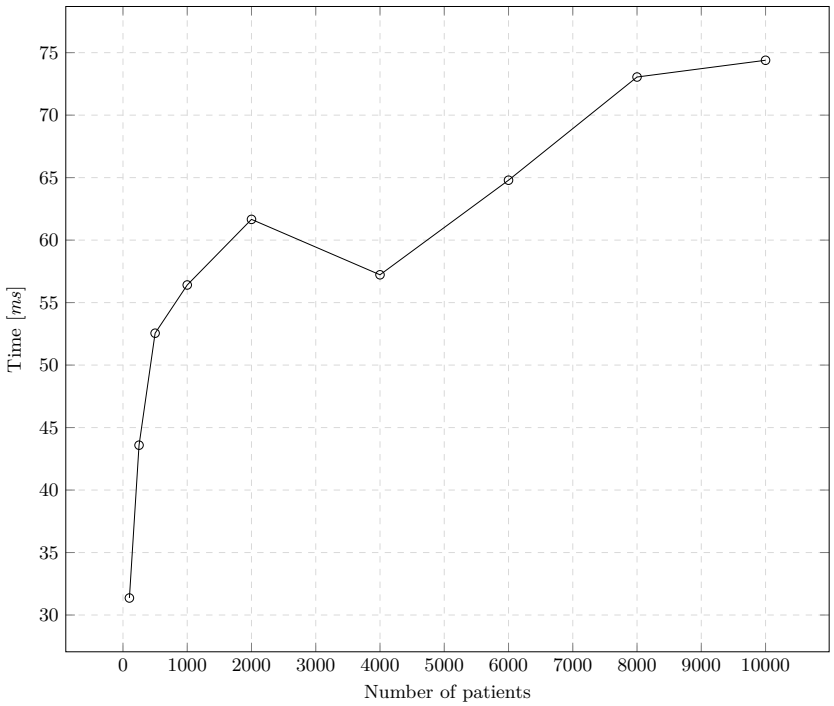


FIGURE 4.14: ANGERONA execution time in seconds.

less than 1.5 minutes on our largest examples. We remark that the grounding process is done off-line and just once.

Part III

Beyond `SELECT`-only attackers

Chapter 5

A Formal Model of Databases

Surely we want solid foundations. What kind of castle can we build on sand? What is the point of devoting effort to balconies and minarets, if the foundation may be so weak as to allow the structure to collapse of its own weight? We want our foundations set on bedrock, designed to last for generations. Who would want an architect who cannot certify the soundness of the foundations of his buildings?

Henry E. Kyburg – Why do we need foundations for modelling uncertainties?

Reasoning about the security of database access control mechanisms requires a formal model of database systems. Such a formal model specifies how the database behaves in response to user commands, and, ultimately, it will serve as a basis for the security proofs.

Existing DBAC works, however, either consider very simplistic formal models [131, 165], which account only for `SELECT` queries, or do not provide any formal model at all [8, 41, 145]. To address this, we present a formal operational semantics of database systems. In contrast to existing models, our semantics supports advanced database features like triggers and views. As we show in Chapter 6, the advanced features supported by our model are security-critical, and attackers can exploit them to violate a database's confidentiality and integrity. Note that this chapter is largely based on [87].

Structure. We informally present our system model together with the features supported by our formal model in Section 5.1. In Section 5.2, we formalize our database model as well as the other supported features, such as triggers and views. Finally, Section 5.3 presents our operational semantics of database systems.

5.1 Overview

Here, we overview our system model, and we present the supported features and commands.

5.1.1 System Model

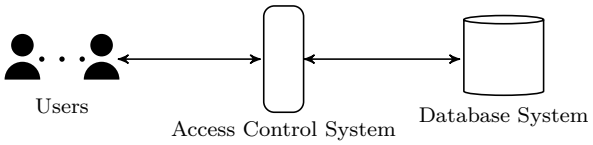


FIGURE 5.1: System model.

In our system model, shown in Figure 5.1, users interact with two components: a database system and an access control system. The access control system contains both a policy enforcement point and a policy decision point. We assume that all the communication between the users and the components is over secure channels.

Database System. The database system (or database for short) manages the data. The database's state is represented by a mapping from relation schemas to sets of tuples. We assume that all database operations are atomic.

Users. Users interact with the database where each command is checked by the access control system. Each user has a unique account through which he can issue commands. We overview the main features supported by our formal model in Section 5.1.2.

SqlStmt := *SelectStmt* | *SqlBasicStmt* | *CreateTrigger* | *CreateView* | *AddUser*
SqlBasicStmt := *InsertStmt* | *DeleteStmt* | *GrantStmt* | *RevokeStmt*
SelectStmt := SELECT φ | SELECT $\{\overline{x} \mid \varphi\}$
InsertStmt := INSERT INTO **tableId** VALUES (*valueList*)
valueList := **const** | *valueList* , **const**
DeleteStmt := DELETE FROM **tableId** WHERE *restrictedExpr*
restrictedExpr := **varId** = **const** | *restrictedExpr* AND **varId** = **const**
GrantStmt := GRANT *privilege* TO **userId** (WITH GRANT OPTION)
RevokeStmt := REVOKE *privilege* FROM **userId** WITH CASCADE
AddUser := ADD USER **userId**
privilege := SELECT ON (**tableId** | **viewId**) | CREATE VIEW |
(INSERT | DELETE | CREATE TRIGGER) ON **tableId**
CreateTrigger := CREATE TRIGGER **triggerId** WITH EXECUTE AS (OWNER | ACTIVATOR)
AFTER (INSERT | DELETE) ON **tableId** WHEN *SelectStmt* DO *SqlBasicStmt*
CreateView := CREATE VIEW **viewId** WITH EXECUTE AS (OWNER | ACTIVATOR) AS *SelectStmt*

FIGURE 5.2: Syntax of the supported commands.

The *system administrator* is a distinguished user responsible for defining the database schema and the initial security policy. In addition to issuing queries and commands, he can create user accounts and assign them to users. The administrator interacts with the access control system through a special account *admin*.

Access Control System. The access control system protects the confidentiality and integrity of the data in the database. It is configured with a security policy S, it intercepts all commands issued by the users, and it prevents the execution of commands that are not authorized by S. When a user u issues a command c, the access control system decides whether u is authorized to execute c. If c complies with the policy, then the access control system forwards the command to the database system, which executes c and returns its result to u. Otherwise, it raises a *security exception* and rejects c. Note that this corresponds to the Non-Truman model [131]. The access control system also logs all issued commands. When evaluating a command, the access control system can access the database's current state and the log.

5.1.2 Supported features

In this chapter, we consider the following SQL features: `SELECT`, `INSERT`, `DELETE`, `GRANT`, `REVOKE`, `CREATE TRIGGER`, `CREATE VIEW`, and `ADD USER` commands. Figure 5.2 presents the syntax of the SQL fragment corresponding to the commands supported by our operational semantics. The fragment's syntax is loosely inspired by TRANSACT-SQL, the SQL's dialect supported by Microsoft SQL Server. Observe that the supported fragment contains the most common SQL commands for data manipulation and access control as well as the core commands for creating triggers and views. The ideas and the techniques presented in this chapter (and in the subsequent ones) are general and can be extended to the entire SQL standard.

Users can retrieve information from the database using `SELECT` commands. Rather than using SQL, however, we formalize queries using the relational calculus (RC), which has a simple and well-defined semantics (see Section 2.2).

We support changes to the database security policy through `GRANT` and `REVOKE` commands. In particular, we support `GRANT` commands to add permissions to the security policy. We also support delegation through `GRANT` commands with `GRANT OPTION`. Finally, privileges can be revoked using `REVOKE` commands with the `CASCADE OPTION`, i.e., when a user revokes a privilege, he also revokes all the privileges that depend on it [137, 162]. Namely, when a user u revokes a privilege p (which has been granted before with `GRANT OPTION`) from a user u', the system automatically revokes all the privileges that u' has further delegated to other users. This check is recursively applied for each revoked privilege with `GRANT OPTION`, leading to the revocation of all dependent privileges.

Users can modify the database's content using `INSERT` and `DELETE` commands. Specifically, we support `INSERT` and `DELETE` commands that explicitly identify the tuple to be inserted or deleted, i.e., commands of the form `INSERT INTO` $table(x_1, \ldots, x_n)$ `VALUES` $(v_1, \ldots, v_n)$ and `DELETE FROM` *table* `WHERE` $x_1 = v_1 \wedge \ldots \wedge x_n = v_n$, where $x_1, \ldots, x_n$ are *table*'s attributes and $v_1, \ldots, v_n$ are the tuple's values. More complex `INSERT` and `DELETE` commands, as well as `UPDATE`s, can be simulated by combining `SELECT`, `INSERT`, and `DELETE` commands.

Our model also supports triggers, which are procedures automatically executed by the database system in response to user commands. In particular, we support only `AFTER` triggers on `INSERT` and `DELETE` events, i.e., triggers that are executed in response to `INSERT` and `DELETE` commands. Our operational semantics supports triggers that are executed either under the privileges of the trigger's owner or under the privileges of the user that executed the command that activated the trigger. The

triggers' `WHEN` conditions are arbitrary boolean queries and their actions are `GRANT`, `REVOKE`, `INSERT`, or `DELETE` commands. Note that database systems usually impose severe restrictions on the `WHEN` clause, such as it must not contain sub-queries. However, most database systems can express arbitrary conditions on triggers by combining control flow statements with `SELECT` commands inside the trigger's body. Thus, we support the class of triggers whose body is of the form `BEGIN IF` *expr* `THEN` *act* `END`, where *act* is either a `GRANT`, `REVOKE`, `INSERT`, or `DELETE` command.

Finally, we support views with both the owner's privileges and the activator's privileges. Additionally, we support `CREATE` commands for creating new triggers and views as well as `ADD USER` commands to create new user accounts. We also support two kinds of integrity constraints: functional dependencies and inclusion dependencies [10]. They model the most widely used families of SQL integrity constraints, namely the `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints.

5.2 Formal Database Model

We now formalize databases including features like views, security policies, and triggers. Our formalization of databases and queries follows Chapter 2, and our security policies formalize SQL policies. In the following, let $\mathcal{U}$, $\mathcal{V}$, and $\mathcal{T}$ be mutually disjoint, countably infinite sets, respectively representing identifiers of users, views, and triggers. Furthermore, we assume that $\mathcal{R} \cap \mathcal{U} = \emptyset$, $\mathcal{R} \cap \mathcal{V} = \emptyset$, and $\mathcal{R} \cap \mathcal{T} = \emptyset$, where $\mathcal{R}$ is the set of relation identifiers from Section 2.2.1.

5.2.1 Databases and Queries

We use the database model presented in Section 2.2. Furthermore, we use the Relational Calculus as query language (see Section 2.2.2). We also support integrity constraints. In this respect, we consider only functional and inclusion dependencies. Recall that functional dependencies are sentences of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}')$, whereas inclusion dependencies are sentence of the form $\forall \overline{x}, \overline{y}. (R(\overline{x}, \overline{y}) \rightarrow \exists \overline{z}. S(\overline{x}, \overline{z}))$.

5.2.2 Views

Let D be a schema. A *view* V *over* D is a tuple $\langle id, o, q, m \rangle$, where $id \in \mathcal{V}$ is the view identifier, $o \in \mathcal{U}$ is the view's owner, q is the non-boolean query over D defining the view, and $m \in \{A, O\}$ is the security mode, where A stands for *activator's privileges* and O stands for *owner's privileges*. Note that the query q may refer to other views. We assume, however, that views have no cyclic dependencies between them. We denote by $\mathcal{VIEW}_D$ the set of all views over D.

The *materialization of a view* $\langle V, o, q, m \rangle$ *in a state* s, denoted by $s(V)$, is $[q]^s$. We extend the relational calculus in the standard way to work with views [10]. Namely, we treat the view materialization in the current database state as the relation associated with the corresponding predicate symbol.

Given a view v, we denote v's owner by $owner(v)$. Moreover, given a formula ϕ, we denote by $tables(\phi)$ the set of tables occurring in the formula ϕ', where ϕ' is obtained from ϕ by (recursively) replacing all views with their definitions. Let D be a database schema and V be a set of views over D. *A query* q *is defined over* D *and* V, denoted by $defined(q, D, V)$, iff q refers only to relation schemas in D and views in V. Moreover, *a view* v *is defined over* D *and* V, denoted by $defined(v, D, V)$, iff the query specifying v is defined over D and V.

5.2.3 Security Policies

We now formalize the SQL access control model [137, 162]. We first formalize five privileges. Let D be a database schema. A `SELECT` *privilege over* D is a tuple $\langle \texttt{SELECT}, R \rangle$, where R is a relation schema in D or a view over D. A `CREATE VIEW` *privilege over* D is a tuple $\langle \texttt{CREATE VIEW} \rangle$. An `INSERT` *privilege over* D is a tuple $\langle \texttt{INSERT}, R \rangle$, a `DELETE` *privilege over* D is a tuple $\langle \texttt{DELETE}, R \rangle$, and a `CREATE TRIGGER` *privilege over* D is a tuple $\langle \texttt{CREATE TRIGGER}, R \rangle$, where R is a relation schema in D. We denote by $\mathcal{PRIV}_D$ the set of privileges over D. Given a database schema D and a set of views V over D, we say that *a privilege* p *is defined over* D *and* V, denoted by $defined(p, D, V)$, iff p refers only to tables and views in D and V.

Following SQL, we use `GRANT` commands to assign privileges to users. Let $U \subseteq \mathcal{U}$ be a set of users and D be a database schema. We now define (U, D)-grants and (U, D)-revokes. There are two types of (U, D)-grants. A (U, D)-*simple grant* is a tuple $\langle \oplus, u, p, u' \rangle$, where $u \in U$ is the user receiving the privilege $p \in \mathcal{PRIV}_D$ and $u' \in U$ is the user granting this privilege. A (U, D)-*grant with grant option* is a tuple $\langle \oplus^*, u, p, u' \rangle$, where u, p, and u' are as before. A (U, D)-*revoke* is a tuple $\langle \ominus, u, p, u' \rangle$, where $u \in U$ is the user from which the privilege $p \in \mathcal{PRIV}_D$ will be revoked and $u' \in U$ is the user revoking this privilege. We denote by $\Omega^{sec}_{U,D}$ the set of all (U, D)-grants and (U, D)-revokes. A grant

$\langle \oplus, u, p, u' \rangle$ models the command `GRANT` p `TO` u issued by u', a grant with grant option $\langle \oplus^*, u, p, u' \rangle$ models the command `GRANT` p `TO` u `WITH GRANT OPTION` issued by u', and a revoke $\langle \ominus, u, p, u' \rangle$ models the command `REVOKE` p `FROM` u `CASCADE` issued by u'.

Finally, we define a (U, D)-*security policy* S as a finite set of (U, D)-grants. We denote by $\mathcal{S}_{U,D}$ the set of all (U, D)-policies.

In the following, we formalize the semantics of `GRANT` and `REVOKE` commands, i.e., how their execution modifies the security policy.

Semantics of `GRANT` commands. Executing a `GRANT` command $\langle \oplus, u, p, u' \rangle$ on a security policy S simply amounts to adding the `GRANT` to the policy, i.e., $S \cup \{\langle \oplus, u, p, u' \rangle\}$. The same holds for `GRANT`s with grant option.

Semantics of `REVOKE` commands. We now define the function *revoke* that models the semantics of SQL's `REVOKE` statements with cascade. In the following, let S be a security policy and p be a privilege. A *chain* is a sequence of grants $g_1 \cdot g_2 \cdot \ldots \cdot g_n$ such that (1) there is a user identifier u and an $op \in \{\oplus, \oplus^*\}$ such that $g_1 = \langle op, u, p, start \rangle$, (2) if $p \neq \langle \texttt{SELECT}, V \rangle$, where V is a view with owner's privileges, then $start = admin$, and $start \in \{admin, owner(V)\}$ otherwise, and (3) for each $1 \leq i \leq n-1$, there are user identifiers u, u', u'' and an $op \in \{\oplus, \oplus^*\}$ such that $g_i = \langle \oplus^*, u', p, u \rangle$ and $g_{i+1} = \langle op, u'', p, u' \rangle$. The function $chain : \mathcal{S}_{\mathcal{U},D} \to 2^{\Omega^{sec}_{\mathcal{U},D}{}^*}$, which takes as input a policy S and constructs all possible chains, is the smallest function satisfying the following recurrence relation:

$$\begin{aligned} chain(S) = &\{\langle op, u, p, u' \rangle \in S \mid u' = admin\} \cup \\ &\{\langle op, u, \langle \texttt{SELECT}, V \rangle, u' \rangle \in S \mid V = \langle v, o, q, O \rangle \wedge u' = o\} \cup \\ &\bigcup_{G \in chain(S)} \{G \cdot g \mid g \in S \wedge g = \langle op, u, p, u' \rangle \wedge G(|G|) = \langle \oplus^*, u', p, u'' \rangle \wedge \\ &\qquad \forall i \in \{1, \ldots, |G|\}.\, G(i) \neq g\}. \end{aligned}$$

The function *filter* takes as input a set of chains C and a grant g and outputs the set of all chains in C not containing g:

$$filter(C, g) := \{g_1 \cdot \ldots \cdot g_n \in C \mid \forall i \in \{1, \ldots, n\}.\, g_i \neq g\}.$$

The function *policy* constructs a policy starting from a set of chains C:

$$policy(C) := \bigcup_{g_1 \cdot \ldots \cdot g_n \in C} \bigcup_{1 \leq i \leq n} \{g_i\}.$$

Finally, the function *revoke*, which models the semantics of the `REVOKE` command, is as follows:

$$revoke(S, u, p, u') := policy(filter(chain(policy(filter(chain(S), \langle \oplus, u, p, u' \rangle))), \langle \oplus^*, u, p, u' \rangle)).$$

Given a policy S, $revoke(S, u, p, u')$ denotes the policy obtained by applying $\langle \ominus, u, p, u' \rangle$ to S.

Example 5.1. Consider a database schema D with three relation schemas N, P, and T, all with arity 1. Moreover, consider the following policy: the user u has read and write access to the tables P and N. Finally, assume that the only other user is the administrator *admin*. This setting can be formalized as follows. The set U is $\{u, admin\}$ and the policy S contains the following grants: $\langle \oplus, u, \langle \texttt{SELECT}, P \rangle, admin \rangle$, $\langle \oplus, u, \langle \texttt{INSERT}, P \rangle, admin \rangle$, $\langle \oplus, u, \langle \texttt{DELETE}, P \rangle, admin \rangle$, $\langle \oplus, u, \langle \texttt{SELECT}, N \rangle, admin \rangle$, $\langle \oplus, u, \langle \texttt{INSERT}, N \rangle, admin \rangle$, and $\langle \oplus, u, \langle \texttt{DELETE}, N \rangle, admin \rangle$. ■

5.2.4 Triggers

Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema. A *trigger over* D is a tuple $\langle id, u, e, R, \phi, a, m \rangle$, where $id \in \mathcal{T}$ is the trigger identifier, $u \in \mathcal{U}$ is the trigger's owner, $e \in \{INS, DEL\}$ is the trigger event (where *INS* stands for `INSERT` and *DEL* stands for `DELETE`), R is a relation schema in Σ, the trigger condition ϕ is a relational calculus formula such that $free(\phi) \subseteq \{x_1, \ldots, x_{|R|}\}$, and the trigger action a is one of: (1) $\langle \texttt{INSERT}, R', \bar{t} \rangle$, where $R' \in D$ and $\bar{t}$ is a $|R'|$-tuple of values in **dom** and variables in $\{x_1, \ldots, x_{|R|}\}$, (2) $\langle \texttt{DELETE}, R', \bar{t} \rangle$, where R' and $\bar{t}$ are as before, or (3) $\langle op, u, p \rangle$, where $op \in \{\oplus, \oplus^*, \ominus\}$, $u \in \mathcal{U}$, and p is a privilege over D. Finally, $m \in \{A, O\}$ is the security mode, where A stands for *activator's privileges* and O stands for *owner's privileges*. We denote by $\mathcal{TRIGGER}_D$ the set of all triggers over D. Furthermore, given a trigger t, we denote t's owner by $owner(t)$ and we denote its security mode as $mode(t)$. To illustrate, the trigger $\langle t, admin, INS, P, T(x_1), \langle \texttt{INSERT}, N, x_1 \rangle, O \rangle$ models a trigger (with identifier t), created by *admin*, that is executed (under *admin*'s privileges) whenever a user inserts a tuple in the relation P, and if $T(v_1)$ holds in the current database state, it inserts $N(v_1)$ into the table N. Here, x_1 is bound, at run-time, to the value v_1 inserted in the table P by the trigger's invoker.

Let U be a set of users, D be a database schema, and V be a set of views over D. A trigger t is a *U-trigger*, denoted by $\mathit{usersIn}(t, U)$, iff $\mathit{owner}(t) \in U$ and t's statement refers just to users in U. Furthermore, we say that *a trigger $t = \langle id, u, e, R, \phi, a, m\rangle$ is defined over a database schema D and a set of views V*, denoted by $\mathit{defined}(t, D, V)$, iff (1) the table R belongs to D, (2) t's `WHEN` condition ϕ is defined over D and V, and (3) t's action a refers only to tables and views in $D \cup V$.

We assume that (1) any command a is executed atomically together with all the triggers activated by a, and (2) triggers do not recursively activate other triggers. Hence, all executions terminate. Formally, a set of triggers T is *safe*, written $\mathit{safe}(T)$, iff for all triggers $t_1, t_2 \in T$: (1) if t_1 is activated by an `INSERT` on a table R, then t_2's action is not of the form $\langle \texttt{INSERT}, R, \overline{t}\rangle$, and (2) if t_1 is activated by a `DELETE` on a table R, then t_2's action is not of the form $\langle \texttt{DELETE}, R, \overline{t}\rangle$. Observe that safety ensures termination. We enforce this condition syntactically at the trigger's creation time. This simple termination condition is sufficient for the purposes of this thesis, i.e., reasoning about database security. Note, however, that our results can be easily extended to more complex and permissive termination conditions.

5.3 Operational Semantics

We formalize our system model as a labelled transition system (LTS). First, we define a system configuration, which describes the database schema and the integrity constraints, and the user actions. Afterwards, we define the system's state, which represents a snapshot of the system that contains the database's state, the identifiers of the users interacting with the system, the access control policy, and the current triggers and views in the system. Finally, we formalize the system's behavior as a small step operational semantics, including all features necessary to reason about security, even in the presence of attacks exploiting advanced database features.

System configurations. A *system configuration* is a tuple $\langle D, \Gamma\rangle$ such that D is a database schema and Γ is a finite set of integrity constraints over D. We assume that the constraints in Γ are functional or inclusion dependencies. Observe that functional and inclusion dependencies model *primary key* and *foreign key* constraints, some of the most common integrity constraints used in practice in database systems.

User actions. Let $M = \langle D, \Gamma\rangle$ be a system configuration and $u \in \mathcal{U}$ be a user. A *(D, u)-action* is one of the following tuples:

- $\langle u, \texttt{ADD\_USER}, u'\rangle$, where $u = \mathit{admin}$ and $u' \in \mathcal{U} \setminus \{\mathit{admin}\}$,
- $\langle u, \texttt{SELECT}, q\rangle$, where q is a boolean query[1] over D,
- $\langle u, \texttt{INSERT}, R, \overline{t}\rangle$, where $R \in D$ and $\overline{t} \in \mathbf{dom}^{|R|}$,
- $\langle u, \texttt{DELETE}, R, \overline{t}\rangle$, where R and $\overline{t}$ are as above,
- $\langle \mathit{op}, u', p, u\rangle$, where $\langle \mathit{op}, u', p, u\rangle \in \Omega^{\mathit{sec}}_{D,\mathcal{U}}$, or
- $\langle u, \texttt{CREATE}, o\rangle$, where $o \in \mathcal{TRIGGER}_D \cup \mathcal{VIEW}_D$.

We denote by $\mathcal{A}_{D,u}$ the set of all (D, u)-actions and by $\mathcal{A}_{D,U}$, for some $U \subseteq \mathcal{U}$, the set $\bigcup_{u\in U} \mathcal{A}_{D,u}$.

System states. Given a system configuration $M = \langle D, \Gamma\rangle$, an *$M$-system state* is a tuple $\langle db, U, \mathit{sec}, T, V\rangle$ such that $db \in \Omega^{\Gamma}_{D}$ is a database state, $U \subset \mathcal{U}$ is a finite set of users such that $\mathit{admin} \in U$, $\mathit{sec} \in \mathcal{S}_{U,D}$ is a security policy, T is a finite set of safe triggers over D, and V is a finite set of views over D. We denote by Π_M the set of all M-system states.

Contexts. An *M-context*, where M is a system configuration, describes the system's history, the scheduled triggers that must be executed, and how to modify the system's state in case a roll-back occurs. We denote by $\mathcal{C}_M$ the set of all M-contexts. We assume that $\mathcal{C}_M$ contains a distinguished element ϵ representing the empty context, which is the context in which the system starts.

We now formalize contexts and all their components. In the following, let $M = \langle D, \Gamma\rangle$ be a system configuration and u be a user. An *(M, u)-action effect* is a tuple $\langle \mathit{act}, \mathit{secDec}, \mathit{res}, E\rangle$, where $\mathit{act} \in \mathcal{A}_{D,u}$ is an action, $\mathit{secDec} \in \{\top, \bot\}$ is the security decision for that action (where $\top$ stands for *permit* and $\bot$ stands for *deny*), $\mathit{res} \in \{\top, \bot\}$ is the action's result, and $E \subseteq \Gamma$ is the set of integrity constraints violated by the action. We denote by $\Omega^{\mathit{actEff}}_{M,u}$ the set of all (M, u)-action effects and by $\Omega^{\mathit{actEff}}_{M,U}$, for some $U \subseteq \mathcal{U}$, the set $\bigcup_{u\in U} \Omega^{\mathit{actEff}}_{M,u}$. An *$(M, u)$-trigger effect* is a triple $\langle t, \mathit{when}, \mathit{stmt}\rangle$ where $t \in \mathcal{TRIGGER}_D$ is a trigger, $\mathit{when} \in \Omega^{\mathit{actEff}}_{M,u}$ is the action effect associated with the trigger's `WHEN` condition, and $\mathit{stmt} \in \Omega^{\mathit{actEff}}_{M,u} \cup \{\epsilon\}$ is the action effect associated with the statement in the trigger's body. We denote by $\Omega^{\mathit{triEff}}_{M,u}$ the set of all (M, u)-trigger effects and by $\Omega^{\mathit{triEff}}_{M,U}$, for some $U \subseteq \mathcal{U}$, the set $\bigcup_{u\in U} \Omega^{\mathit{triEff}}_{M,u}$.

[1] Without loss of generality, we focus only on boolean queries [10]. We can support non-boolean queries as follows. Given a database state s and a query $q := \{\overline{x} \mid \phi\}$, if the access control mechanism authorizes the boolean query $\bigwedge_{\overline{t}\in[q]^s} \phi[\overline{x} \mapsto \overline{t}] \wedge (\forall \overline{x}.\, \phi \rightarrow \bigvee_{\overline{t}\in[q]^s} \overline{x} = \overline{t})$, then we return q's result, and otherwise we reject q as unauthorized.

An *M-trigger transaction* is a 4-tuple $\langle s, \bar{t}, u, tr\rangle$, where $s \in \Pi_M \cup \{\epsilon\}$ is an M-system state representing the "roll-back state", i.e., the state that should be restored in case a roll-back happens, $\bar{t} \in \{\epsilon\} \cup \bigcup_{n\in\mathbb{N}^+} \mathbf{dom}^n$ is the tuple involved in the event that has fired the transaction, $u \in \mathcal{U} \cup \{\epsilon\}$ is the user that has activated the triggers in the transactions, and $tr \in \mathcal{TRIGGER}_D^*$ is a sequence of triggers. Note that we denote by $\langle \epsilon, \epsilon, \epsilon, \epsilon\rangle$ the empty M-trigger transaction.

An *M-history* h is a sequence of action effects and trigger effects, i.e., $h \in (\Omega_{M,\mathcal{U}}^{actEff} \cup \Omega_{M,\mathcal{U}}^{triEff})^*$, and we denote by $\mathcal{H}_M$ the set of all M-histories.

We are now ready to formally define contexts. Let $M = \langle D, \Gamma\rangle$ be a system configuration. An *M-context* is a tuple $\langle h, actEff, tr\rangle$, where $h \in \mathcal{H}_M$ models the system's history, $actEff \in \Omega_{M,\mathcal{U}}^{actEff} \cup \Omega_{M,\mathcal{U}}^{triEff} \cup \{\epsilon\}$ describes the last action's effect, i.e., whether the action has been accepted by the access control mechanism and the action's result, and tr is an M-trigger transaction. Furthermore, the empty context ϵ is the element $\langle \epsilon, \epsilon, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle$.

Runtime states. An *M-runtime state* is a tuple $\langle db, U, sec, T, V, ctx\rangle$ such that $\langle db, U, sec, T, V\rangle$ is an M-system state and $ctx \in \mathcal{C}_M$ is an M-context. We denote by Ω_M the set of all M-runtime states. Observe that we often write $\langle db, U, sec, T, V, h, actEff, tr\rangle$ instead of $\langle db, U, sec, T, V, \langle h, actEff, tr\rangle\rangle$. For simplicity, we sometimes write $\langle s, ctx\rangle$, where s is a system state $\langle db, U, sec, T, V\rangle$ and ctx is a context to represent the runtime state $\langle db, U, sec, T, V, c\rangle$. Finally, whenever this is clear from the context, we often refer to *system states* and *runtime states* simply as *states*.

A runtime state $\langle db, U, sec, T, V, c\rangle$ is *initial* iff (a) sec contains only grants issued by *admin*, (b) T (respectively V) contains only triggers (respectively views) owned by *admin*, and (c) $c = \epsilon$. We denote by $\mathcal{I}_M$ the set of all initial states. Given a runtime state $s = \langle db, U, sec, T, V, c\rangle$, we denote by $sysState(s)$ the M-system state $\langle db, U, sec, T, V\rangle$ obtained from s by dropping the context c and by $ctx(s)$ its context c. Furthermore, given an M-state $s = \langle db, sec, U, T, V, c\rangle$, we use a dot notation to refer to its components. For instance, we use $s.db$ to refer to the database's state in s and $s.sec$ to refer to the policy in s.

Policy Decision Points. Given a system configuration M, an *M-Policy Decision Point* (M-PDP) is a total function $f : \Omega_M \times \mathcal{A}_{D,\mathcal{U}} \to \{\top, \bot\}$ that maps each runtime state s and action a to a security decision represented by a boolean value, where $\top$ stands for permit and $\bot$ stands for deny. An *extended configuration* is a tuple $\langle M, f\rangle$, where M is a system configuration and f is an M-PDP.

Operational Semantics. We now define the LTS representing the system model.

Definition 5.1. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ and f is an M-PDP. The *P-LTS* is the labelled transition system $\langle S, A, \to_f, I\rangle$ where $S = \Omega_M$ is the set of states, $A = \mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$ is the set of actions, $\to_f \subseteq S \times A \times S$ is the transition relation, and $I = \mathcal{I}_M$ is the set of initial states. □

Let $P = \langle M, f\rangle$ be an extended configuration. A *run* r of a P-LTS L is a finite alternating sequence of runtime states and actions, which starts with an initial state s, ends in some state s', and respects the transition relation $\to_f$. We denote by $traces(L)$ the set of all L's runs. Given a run r, $|r|$ denotes the number of states in r, $last(r)$ denotes r's last state, and r^i, where $1 \leq i \leq |r|$, denotes the run obtained by truncating r at the i-th state. With a slight abuse of notation, we denote by r^0 the empty run, i.e., the one without states.

The relation $\to_f$ formalizes the system's small step operational semantics, and it is defined in Figures 5.3–5.6. Observe that in the rules we assume fixed a database schema D. Note also that the rules rely on several auxiliary functions that we define in Section 5.3.1.

Figure 5.3 presents the rules for the `ADD USER`, `SELECT`, `GRANT`, and `REVOKE` commands. For each of these commands, there are two rules: one handling the command's successful execution, which is applied in case the command is authorized by the PDP f, and the other handling commands whose execution has been deemed insecure by the PDP f. For instance, the rule `SELECT` Success models the system's behavior when the user u issues a `SELECT` query q that is authorized by the PDP f, whereas the `SELECT` Deny rule models the system's behavior whenever a `SELECT` query is unauthorized. Both rules check the security decision produced by the PDP f and whether the query is well-defined given the database schema D and the current set of views V. Additionally, the `SELECT` Success rule also computes q's result on the current database state db. The only component of the M-state s that changes is the context, which is updated by extending the history with the action event associated with the `SELECT` query. Observe that, for the rule `SELECT` Success, the action event also contains q's result on the current database state. The rules for the other commands are similar to those for the `SELECT` command, the main difference being how they modify the database state.

Figure 5.4 formalizes the semantics for `INSERT` and `DELETE` commands. We illustrate the rules using `INSERT` commands as an example. The rules for `DELETE` commands are similar. The rules `INSERT` Success 1 and `INSERT` Success 2 formalize the successful execution of `INSERT` commands. They update the database state by adding the tuple $\bar{t}$ to the relation R. They also check that (1) the PDP f authorizes the `INSERT` command, and (2) the database state obtained by executing the

ADD USER Success

$$\frac{aE' = \langle\langle admin, \texttt{ADD\_USER}, u\rangle, \top, \top, \emptyset\rangle \qquad admin \in U \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle, \langle admin, \texttt{ADD\_USER}, u\rangle) = \top}{\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle admin, \texttt{ADD\_USER}, u\rangle}_f \langle db, U \cup \{u\}, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

ADD USER Deny

$$\frac{aE' = \langle\langle admin, \texttt{ADD\_USER}, u\rangle, \bot, \bot, \emptyset\rangle \qquad admin \in U \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle, \langle admin, \texttt{ADD\_USER}, u\rangle) = \bot}{\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle admin, \texttt{ADD\_USER}, u\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

SELECT Success

$$\frac{u \in U \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle, \langle u, \texttt{SELECT}, q\rangle) = \top \qquad [q]^{db} = v \qquad aE' = \langle\langle u, \texttt{SELECT}, q\rangle, \top, v, \emptyset\rangle \qquad defined(q, D, V)}{\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{SELECT}, q\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

SELECT Deny

$$\frac{u \in U \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle, \langle u, \texttt{SELECT}, q\rangle) = \bot \qquad aE' = \langle\langle u, \texttt{SELECT}, q\rangle, \bot, \bot, \emptyset\rangle \qquad defined(q, D, V)}{\langle db, U, sec, T, V, h, aE, \langle rS, \vec{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{SELECT}, q\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

GRANT Success

$$\frac{u, u' \in U \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u'', \epsilon\rangle\rangle, \langle op, u, p, u'\rangle) = \top \qquad aE' = \langle\langle op, u, p, u'\rangle, \top, \top, \emptyset\rangle \qquad op \in \{\oplus, \oplus^*\} \qquad defined(p, D, V) \qquad sec' = sec \cup \{\langle op, u, p, u'\rangle\}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u'', \epsilon\rangle\rangle \xrightarrow{\langle op, u, p, u'\rangle}_f \langle db, U, sec', T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

REVOKE Success

$$\frac{u, u' \in U \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u'', \epsilon\rangle\rangle, \langle \ominus, u, p, u'\rangle) = \top \qquad aE' = \langle\langle \ominus, u, p, u'\rangle, \top, \top, \emptyset\rangle \qquad defined(p, D, V) \qquad sec' = revoke(sec, u, p, u')}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u'', \epsilon\rangle\rangle \xrightarrow{\langle \ominus, u, p, u'\rangle}_f \langle db, U, sec', T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

GRANT-REVOKE Deny

$$\frac{u, u' \in U \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u'', \epsilon\rangle\rangle, \langle op, u, p, u'\rangle) = \bot \qquad aE' = \langle\langle op, u, p, u'\rangle, \bot, \bot, \emptyset\rangle \qquad op \in \{\oplus, \oplus^*, \ominus\} \qquad defined(p, D, V)}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u'', \epsilon\rangle\rangle \xrightarrow{\langle op, u, p, u'\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

Figure 5.3: Rules for the ADD USER, SELECT, GRANT, and REVOKE commands.

INSERT Success 1

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, act) = \top \\ act = \langle u, \texttt{INSERT}, R, \bar{t}\rangle \qquad db[R \oplus \bar{t}] \in \Omega_D^\Gamma \qquad aE' = \langle act, \top, \top, \emptyset\rangle \qquad \mathit{filter}(T, INS, R) = \epsilon \vee \bar{t} \in db(R) \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{INSERT}, R, \bar{t}\rangle}_f \langle db[R \oplus \bar{t}], U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

INSERT Success 2

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad act = \langle u, \texttt{INSERT}, R, \bar{t}\rangle \\ f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, act) = \top \qquad db[R \oplus \bar{t}] \in \Omega_D^\Gamma \\ aE' = \langle act, \top, \top, \emptyset\rangle \qquad tr = \mathit{filter}(T, INS, R) \qquad tr \neq \epsilon \qquad \bar{t} \notin db(R) \qquad rS' = \langle db, U, sec, T, V\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{INSERT}, R, \bar{t}\rangle}_f \langle db[R \oplus \bar{t}], U, sec, T, V, h{\cdot}aE, aE', \langle rS', \bar{t}, u, tr\rangle\rangle}$$

INSERT Exception

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top \\ E' = \{\phi \in \Gamma \mid [\phi]^{db[R \oplus \bar{t}]} = \bot\} \qquad E' \neq \emptyset \qquad aE' = \langle\langle u, \texttt{INSERT}, R, \bar{t}\rangle, \top, \bot, E'\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{INSERT}, R, \bar{t}\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

INSERT Deny

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad aE' = \langle\langle u, \texttt{INSERT}, R, \bar{t}\rangle, \bot, \bot, \emptyset\rangle \\ f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \bot \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{INSERT}, R, \bar{t}\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

DELETE Success 1

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, act) = \top \\ act = \langle u, \texttt{DELETE}, R, \bar{t}\rangle \qquad db[R \ominus \bar{t}] \in \Omega_D^\Gamma \qquad aE' = \langle act, \top, \top, \emptyset\rangle \qquad \mathit{filter}(T, DEL, R) = \epsilon \vee \bar{t} \notin db(R) \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{DELETE}, R, \bar{t}\rangle}_f \langle db[R \ominus \bar{t}], U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

DELETE Success 2

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, act) = \top \\ act = \langle u, \texttt{DELETE}, R, \bar{t}\rangle \qquad db[R \ominus \bar{t}] \in \Omega_D^\Gamma \\ aE' = \langle act, \top, \top, \emptyset\rangle \qquad tr = \mathit{filter}(T, DEL, R) \qquad tr \neq \epsilon \qquad \bar{t} \in db(R) \qquad rS' = \langle db, U, sec, T, V\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{DELETE}, R, \bar{t}\rangle}_f \langle db[R \ominus \bar{t}], U, sec, T, V, h{\cdot}aE, aE', \langle rS', \bar{t}, u, tr\rangle\rangle}$$

DELETE Exception

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, \langle u, \texttt{DELETE}, R, \bar{t}\rangle) = \top \\ E' = \{\phi \in \Gamma \mid [\phi]^{db[R \ominus \bar{t}]} = \bot\} \qquad E' \neq \emptyset \qquad aE' = \langle\langle u, \texttt{DELETE}, R, \bar{t}\rangle, \top, \bot, E'\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{DELETE}, R, \bar{t}\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

DELETE Deny

$$\frac{\begin{array}{c} u \in U \qquad R \in D \qquad aE' = \langle\langle u, \texttt{DELETE}, R, \bar{t}\rangle, \bot, \bot, \emptyset\rangle \\ f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle, \langle u, \texttt{DELETE}, R, \bar{t}\rangle) = \bot \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}', u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{DELETE}, R, \bar{t}\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

Figure 5.4: Rules for the INSERT and DELETE commands.

TRIGGER INSERT-DELETE SUCCESS

$$\frac{\begin{array}{c} invk, ow \in U \qquad t = \langle id, ow, ev, R', \phi, stmt, m\rangle \qquad u = getActualUser(m, ow, invk) \\ \phi' = \phi[\overline{x}^{|R'|} \mapsto \overline{t}] \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, \langle u, \texttt{SELECT}, \phi'\rangle) = \top \\ [\phi']^{db} = \top \qquad aE' = \langle\langle u, \texttt{SELECT}, \phi'\rangle, \top, \top, \emptyset\rangle \\ act = action(stmt, u, \overline{t}) \qquad db' = apply(act, db) \qquad f(\langle db, U, sec, T, V, h{\cdot}aE, aE', \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, act) = \top \\ db' \in \Omega_D^{\Gamma} \qquad aE'' = \langle act, \top, \top, \emptyset\rangle \qquad tE' = \langle t, aE', aE''\rangle \qquad ID(act) = \top \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle \xrightarrow{t}_f \langle db', U, sec, T, V, h{\cdot}aE, tE', \langle rS, \overline{t}, invk, tr\rangle\rangle}$$

TRIGGER INSERT-DELETE EXCEPTION

$$\frac{\begin{array}{c} invk, ow \in U \qquad t = \langle id, ow, ev, R', \phi, stmt, m\rangle \qquad u = getActualUser(m, ow, invk) \\ rS = \langle db', U', sec', T', V'\rangle \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, \langle u, \texttt{SELECT}, \phi'\rangle) = \top \\ [\phi']^{db} = \top \qquad aE' = \langle\langle u, \texttt{SELECT}, \phi'\rangle, \top, \top, \emptyset\rangle \qquad act = action(stmt, u, \overline{t}) \\ f(\langle db, U, sec, T, V, h{\cdot}aE, aE', \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, act) = \top \qquad ID(act) = \top \qquad \phi' = \phi[\overline{x}^{|R'|} \mapsto \overline{t}] \\ E' = \{\phi \in \Gamma \mid [\phi]^{apply(act, db)}\} \qquad E' \neq \emptyset \qquad aE'' = \langle act, \top, \bot, E'\rangle \qquad tE' = \langle t, aE', aE''\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle \xrightarrow{t}_f \langle db', U', sec', T', V', h{\cdot}aE, tE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

TRIGGER GRANT SUCCESS

$$\frac{\begin{array}{c} invk, ow \in U \qquad t = \langle id, ow, ev, R', \phi, stmt, m\rangle \qquad u = getActualUser(m, ow, invk) \\ \phi' = \phi[\overline{x}^{|R'|} \mapsto \overline{t}] \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, \langle u, \texttt{SELECT}, \phi'\rangle) = \top \\ [\phi']^{db} = \top \qquad aE' = \langle\langle u, \texttt{SELECT}, \phi'\rangle, \top, \top, \emptyset\rangle \\ \langle op, u', p, u\rangle = action(stmt, u, \overline{t}) \qquad f(\langle db, U, sec, T, V, h{\cdot}aE, aE', \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, \langle op, u', p, u\rangle) = \top \\ aE'' = \langle\langle op, u', p, u\rangle, \top, \top, \emptyset\rangle \qquad tE' = \langle t, aE', aE''\rangle \qquad op \in \{\oplus, \oplus^*\} \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle \xrightarrow{t}_f \langle db, U, sec \cup \{\langle op, u', p, u\rangle\}, T, V, h{\cdot}aE, tE', \langle rS, \overline{t}, invk, tr\rangle\rangle}$$

TRIGGER REVOKE SUCCESS

$$\frac{\begin{array}{c} invk, ow \in U \qquad t = \langle id, ow, ev, R', \phi, stmt, m\rangle \qquad u = getActualUser(m, ow, invk) \\ \phi' = \phi[\overline{x}^{|R'|} \mapsto \overline{t}] \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, \langle u, \texttt{SELECT}, \phi'\rangle) = \top \\ [\phi']^{db} = \top \qquad aE' = \langle\langle u, \texttt{SELECT}, \phi'\rangle, \top, \top, \emptyset\rangle \\ \langle \ominus, u', p, u\rangle = action(stmt, u, \overline{t}) \qquad f(\langle db, U, sec, T, V, h{\cdot}aE, aE', \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, \langle \ominus, u', p, u\rangle) = \top \\ aE'' = \langle\langle \ominus, u', p, u\rangle, \top, \top, \emptyset\rangle \qquad tE' = \langle t, aE', aE''\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle \xrightarrow{t}_f \langle db, U, revoke(sec, u, p, u'), T, V, h{\cdot}aE, tE', \langle rS, \overline{t}, invk, tr\rangle\rangle}$$

TRIGGER DISABLED

$$\frac{\begin{array}{c} invk, ow \in U \qquad t = \langle id, ow, ev, R', \phi, stmt, m\rangle \qquad u = getActualUser(m, ow, invk) \\ \phi' = \phi[\overline{x}^{|R'|} \mapsto \overline{t}] \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, tr\rangle\rangle, \langle u, \texttt{SELECT}, \phi'\rangle) = \top \\ [\phi']^{db} = \bot \qquad aE' = \langle\langle u, \texttt{SELECT}, \phi'\rangle, \top, \bot, \emptyset\rangle \qquad tE' = \langle t, aE', \epsilon\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle \xrightarrow{t}_f \langle db, U, sec, T, V, h{\cdot}aE, tE', \langle rS, \overline{t}, invk, tr\rangle\rangle}$$

TRIGGER DENY CONDITION

$$\frac{\begin{array}{c} invk, ow \in U \qquad t = \langle id, ow, ev, R', \phi, stmt, m\rangle \qquad u = getActualUser(m, ow, invk) \\ rS = \langle db', U', sec', T', V'\rangle \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, tr\rangle\rangle, \langle u, \texttt{SELECT}, \phi'\rangle) = \bot \\ aE' = \langle\langle u, \texttt{SELECT}, \phi'\rangle, \bot, \bot, \emptyset\rangle \qquad tE' = \langle t, aE', \epsilon\rangle \qquad \phi' = \phi[\overline{x}^{|R'|} \mapsto \overline{t}] \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle \xrightarrow{t}_f \langle db', U', sec', T', V', h{\cdot}aE, tE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

TRIGGER DENY ACTION

$$\frac{\begin{array}{c} invk, ow \in U \qquad t = \langle id, ow, ev, R', \phi, stmt, m\rangle \\ u = getActualUser(m, ow, invk) \qquad f(\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, \langle u, \texttt{SELECT}, \phi'\rangle) = \top \\ [\phi']^{db} = \top \qquad aE' = \langle\langle u, \texttt{SELECT}, \phi'\rangle, \top, \top, \emptyset\rangle \\ act = action(stmt, u, \overline{t}) \qquad f(\langle db, U, sec, T, V, h{\cdot}aE, aE', \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle, act) = \bot \\ \phi' = \phi[\overline{x}^{|R'|} \mapsto \overline{t}] \qquad aE'' = \langle act, \bot, \bot, \emptyset\rangle \qquad tE' = \langle t, aE', aE''\rangle \qquad rS = \langle db', U', sec', T', V'\rangle \end{array}}{\langle db, U, sec, T, V, h, aE, \langle rS, \overline{t}, invk, t{\cdot}tr\rangle\rangle \xrightarrow{t}_f \langle db', U', sec', T', V', h{\cdot}aE, tE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}$$

FIGURE 5.5: Rules for triggers.

$$\frac{u \in U \quad defined(t, D, V) \quad safe(\{t\} \cup T) \quad usersIn(t, U) \quad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle, \langle u, \texttt{CREATE}, t\rangle) = \top \quad aE' = \langle\langle u, \texttt{CREATE}, t\rangle, \top, \top, \emptyset\rangle \quad t = \langle id, u, ev, R, \phi, stmt, m\rangle \quad \neg\exists t' \in T.\, t' = \langle id, ow', ev', R', \phi', stmt', m'\rangle}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{CREATE}, t\rangle}_f \langle db, U, sec, T \cup \{t\}, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}\ \textsc{Create Trigger Success}$$

$$\frac{u \in U \quad defined(t, D, V) \quad safe(\{t\} \cup T) \quad usersIn(t, U) \quad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle, \langle u, \texttt{CREATE}, t\rangle) = \top \quad aE' = \langle\langle u, \texttt{CREATE}, t\rangle, \top, \bot, \emptyset\rangle \quad t = \langle id, u, ev, R, \phi, stmt, m\rangle \quad t' = \langle id, ow', ev', R', \phi', stmt', m'\rangle \quad t' \in T \quad t' \neq t}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{CREATE}, t\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}\ \textsc{Create Trigger Deny}$$

$$\frac{u \in U \quad defined(v, D, V) \quad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle, \langle u, \texttt{CREATE}, v\rangle) = \top \quad v = \langle id, u, q, m\rangle \quad aE' = \langle\langle u, \texttt{CREATE}, v\rangle, \top, \top, \emptyset\rangle \quad \neg\exists v' \in V.\, v' = \langle id, ow', q', m'\rangle}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{CREATE}, v\rangle}_f \langle db, U, sec, T, V \cup \{v\}, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}\ \textsc{Create View Success}$$

$$\frac{u \in U \quad defined(v, D, V) \quad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle, \langle u, \texttt{CREATE}, v\rangle) = \top \quad v = \langle id, u, q, m\rangle \quad aE' = \langle\langle u, \texttt{CREATE}, v\rangle, \top, \bot, \emptyset\rangle \quad v' = \langle id, ow', q', m'\rangle \quad v' \in V \quad v \neq v'}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{CREATE}, v\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}\ \textsc{Create View Deny}$$

$$\frac{u \in U \quad defined(o, D, V) \quad aE' = \langle\langle u, \texttt{CREATE}, o\rangle, \bot, \bot, \emptyset\rangle \quad f(\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle, \langle u, \texttt{CREATE}, o\rangle) = \bot}{\langle db, U, sec, T, V, h, aE, \langle rS, \bar{t}, u', \epsilon\rangle\rangle \xrightarrow{\langle u, \texttt{CREATE}, o\rangle}_f \langle db, U, sec, T, V, h{\cdot}aE, aE', \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}\ \textsc{Create Deny}$$

Figure 5.6: Rules for the creation of triggers and views.

`INSERT` command still satisfies the integrity constraints. Observe that the `INSERT` Success 1 rule handles `INSERT` commands that do not activate triggers. This is done by ensuring that (1) there are no triggers in the system that could be executed in response to the `INSERT` command (using the *filter* function formalized in Section 5.3.1), or (2) the execution of the `INSERT` command does not modify the database state. The `INSERT` Success 2 rule, instead, handles `INSERT` commands that activate triggers. The rule updates the current context by appending the transaction associated with the triggers to be executed. Furthermore, the `INSERT` Exception rule handles authorized `INSERT` commands that violate the integrity constraints and therefore throw integrity exceptions. The rule updates the system's context by recording the set of violated integrity constraints $\{\phi \in \Gamma \mid [\phi]^{db[R \oplus \bar{t}]} = \bot\}$. Finally, the `INSERT` Deny rule handles `INSERT` commands that are deemed insecure by the PDP f and it updates the context accordingly.

The rules for triggers are presented in Figure 5.5. Note that these rules rely on the functions *getActualUser*, *apply*, and *action*, which are formalized in Section 5.3.1. The rules Trigger `INSERT-DELETE` Success, Trigger `GRANT` Success, and Trigger `REVOKE` Success formalize the semantics of triggers that successfully execute commands, without violating the security policy or causing integrity exceptions. These rules are inspired by the corresponding rules for the successful execution of `INSERT`, `DELETE`, `GRANT`, and `REVOKE` commands. Additionally, these rules modify the system's context by consuming the first element in the trigger's transaction. This ensures that triggers are executed in the order specified by the transaction. Furthermore, the Trigger Disabled rule handles the execution of triggers whose `WHEN` condition is not satisfied in the current state. In this case, the database is not modified as a result of the trigger's execution, and the system's context is updated by consuming the first element in the current transaction. Finally, the rules Trigger `INSERT-DELETE` Exception, Trigger Deny Condition, and Trigger Deny Action formalize what happens whenever triggers throw exceptions. The first rule formalizes the semantics of triggers whose actions violate integrity constraints. The handling of the command execution and of the exceptions is similar to the `INSERT` Exception and `DELETE` Exception rules from Figure 5.4. The rules Trigger Deny Condition and Trigger Deny Action, instead, formalize the system's behavior in case the condition or the action associated with the trigger violate the security policy. These three rules (1) terminate the execution of the triggers still in the transactions by setting the list of triggers to be processed to ϵ, and (2) roll-back the current database state to the one stored in the transaction (i.e., the one before executing the command that fired all the triggers). This guarantees that our semantics matches the behavior of existing database systems, where exceptions caused by triggers terminate the execution of all triggers and roll-back the database state.

Finally, Figure 5.6 illustrates the rules associated with the `CREATE VIEW` and `CREATE TRIGGER` commands. The rules `CREATE TRIGGER` Success and `CREATE VIEW` Success handle the successful execution of `CREATE` commands. The rules `CREATE TRIGGER` Deny and `CREATE VIEW` Deny formalize the unsuccessful execution of `CREATE` commands due to clashes between triggers and views identifiers. Finally, the rule `CREATE` Deny formalizes how the system handles `CREATE` commands that violate the security policy.

We developed an executable version of our operational semantics, available at [86], using the Maude term-rewriting framework [54]. The executable model acts as a reference implementation for our semantics. We remark that both our operational semantics and the corresponding executable model can be tailored to model the behavior of specific database systems. Given the complexity of databases and their features, having an executable version of our semantics provides a way to validate it against existing database systems.

5.3.1 Auxiliary functions

Here we define several auxiliary functions to extract information from runtime states.

Function used in the operational semantics

We now define the functions mentioned in the rules in Figures 5.3–5.6. The $getActualUser(m, invk, ow)$ function takes as input a security mode $m \in \{A, O\}$ and two users $invk$ and ow in $\mathcal{U}$, and it outputs one of the two users depending on m:

$$getActualUser(m, invk, ow) = \begin{cases} invk & \text{if } m = A \\ ow & \text{if } m = O \end{cases}$$

The ID function takes as input an action $act \in \mathcal{A}_{D,\mathcal{U}}$ and returns $\top$ if act is either $\langle u, \texttt{INSERT}, R, \overline{t}\rangle$ or $\langle u, \texttt{DELETE}, R, \overline{t}\rangle$, for some $u \in \mathcal{U}$, $R \in D$, and $\overline{t} \in \mathbf{dom}^{|R|}$. The function ID returns $\bot$ otherwise.

The $apply$ function takes as input an `INSERT` or `DELETE` action $act \in \mathcal{A}_{D,\mathcal{U}}$ and a database state $db \in \Omega_D$ and it returns as output the updated database state. The function is defined as follows:

$$apply(act, db) = \begin{cases} db[R \oplus \overline{t}] & \text{if } act = \langle u, \texttt{INSERT}, R, \overline{t}\rangle \\ db[R \ominus \overline{t}] & \text{if } act = \langle u, \texttt{DELETE}, R, \overline{t}\rangle \end{cases}$$

We are now ready to define the function $action$, which takes as input a trigger $\langle id, ow, ev, R', \phi, stmt, m\rangle$, a user u, and a tuple $\overline{t}' \in \mathbf{dom}^{|R'|}$, and returns the concrete action executed by the system. Formally, $action$ is defined as follows:

$$action(\langle id, ow, ev, R', \phi, stmt, m\rangle, u, \overline{t}') = \begin{cases} \langle u, \texttt{INSERT}, R, \overline{t}[\overline{x}^{|R'|} \mapsto \overline{t}']\rangle & \text{if } stmt = \langle \texttt{INSERT}, R, \overline{t}\rangle \\ \langle u, \texttt{DELETE}, R, \overline{t}[\overline{x}^{|R'|} \mapsto \overline{t}']\rangle & \text{if } stmt = \langle \texttt{DELETE}, R, \overline{t}\rangle \\ \langle op, u', p, u\rangle & \text{if } stmt = \langle op, u', p\rangle \wedge \\ & \quad op \in \{\ominus, \oplus, \oplus^*\} \end{cases}$$

where $\overline{x}^{|R'|}$ denotes the tuple of variables $\langle x_1, \ldots, x_{|R'|}\rangle$.

We assume given a total-order relation $\preceq_{\mathcal{T}}$ over $\mathcal{T}$. We use this ordering to determine the order in which triggers are executed. Given a set of triggers T and a database schema D, we denote by $filter(T, ev, R)$, where $ev \in \{INS, DEL\}$ and $R \in D$, the sequence of triggers in T (ordered according to $\preceq_{\mathcal{T}}$) whose event is ev and whose relation schema is R.

Auxiliary Functions over contexts

Given an M-context $c = \langle h, aE, tr\rangle$, we denote by $secEx$ the following function, which returns $\top$ if the effect aE is associated with a security exception.

$$secEx(\langle h, aE, tr\rangle) = \begin{cases} \top & \text{if } aE = \langle act, \bot, res, E\rangle \\ \top & \text{if } aE = \langle t, \langle act, \bot, res, E\rangle, \epsilon\rangle \\ \top & \text{if } aE = \langle t, when, \langle act, \bot, res, E\rangle\rangle \\ \bot & \text{otherwise} \end{cases}$$

Similarly, we denote by $Ex(c)$ the function extracting the (violated) integrity constraints stored in the effect aE.

$$Ex(\langle h, aE, tr\rangle) = \begin{cases} E & \text{if } aE = \langle act, aC, res, E\rangle \\ E & \text{if } aE = \langle t, when, \langle act, aC, res, E\rangle\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

Additionally, the functions $acA(c)$ and $acC(c)$ extract the access control decision associated with the trigger's action and condition respectively:

$$acA(\langle h, aE, tr\rangle) = \begin{cases} aC' & \text{if } aE = \langle t, \langle act, aC, res, E\rangle, \langle act', aC', res', E'\rangle\rangle \wedge \langle act', aC', res', E'\rangle \neq \epsilon \\ \bot & \text{otherwise} \end{cases}$$

$$acC(\langle h, aE, tr\rangle) = \begin{cases} aC & \text{if } aE = \langle t, \langle act, aC, res, E\rangle, \epsilon\rangle \\ aC & \text{if } aE = \langle t, \langle act, aC, res, E\rangle, \langle act', aC', res', E'\rangle\rangle \wedge \langle act', aC', res', E'\rangle \neq \epsilon \\ \bot & \text{otherwise} \end{cases}$$

The function $res(c)$, instead, extracts the action's result from the effect aE:

$$res(\langle h, aE, tr\rangle) = \begin{cases} res & \text{if } aE = \langle act, aC, res, E\rangle \\ aC & \text{if } aE = \langle t, \langle act, aC, res, E\rangle, \epsilon\rangle \\ aC \wedge aC' \wedge res' & \text{if } aE = \langle t, \langle act, aC, res, E\rangle, \langle act', aC', res', E'\rangle\rangle \text{ and} \\ & \quad \langle act', aC', res', E'\rangle \neq \epsilon \end{cases}$$

We denote by $invoker(c)$ the function extracting the user in the transaction, i.e., $invoker(\langle h, aE, \langle s, \bar{t}, u, trL\rangle\rangle) = u$. Similarly, we denote by $tpl(c)$ the function extracting the tuple that has fired the transaction, namely $tpl(\langle h, aE, \langle s, \bar{t}, u, trL\rangle\rangle) = \bar{t}$, by $triggers(c)$ the function extracting the list of triggers, i.e., $triggers(\langle h, aE, \langle s, \bar{t}, u, trL\rangle\rangle) = trL$, and by $trigger(c)$, or $tr(c)$ for short, the first trigger in the sequence $triggers(c)$. With a slight abuse of notation, we lift all the aforementioned functions from contexts to runtime states. For instance, $Ex(s)$, where s is a runtime state, is simply $Ex(ctx(s))$.

Chapter 6

Database Access Control in Modern Databases

If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.

Sun Tzu – The art of war

The SQL standard supports database access control, and almost all Database Management Systems (DBMSs) have accordingly developed access control mechanisms. The standard however fails to define a precise access control semantics, the attacker model, and the security properties that the mechanisms ought to satisfy. As a consequence, existing access control mechanisms are implemented in an ad hoc fashion, with neither precise security guarantees nor the means to verify them.

This deficit has dire and immediate consequences. In this chapter, we show that popular database systems are susceptible to two types of attacks. Integrity attacks allow an attacker to perform non-authorized changes to the database. Confidentiality attacks allow an attacker to learn sensitive data. These attacks exploit advanced SQL features, such as triggers, views, and integrity constraints, and they are easy to carry out.

Current research efforts in database security are neither adequate for evaluating the security of modern databases, nor do they account for their advanced features. In more detail, existing research [13,40,131,165] implicitly considers attackers who use `SELECT` commands. But the capabilities offered by databases go far beyond `SELECT`. Users, in general, can modify the database's state and security policy, as well as use features such as triggers, views, and integrity constraints. Consequently, all proposed research solutions fail to prevent attacks such as those we present in Section 6.1.

In summary, the database vendors have been left to develop access control mechanisms without guidance from either the SQL standard or existing research in database security. It is therefore not surprising that modern databases are open to abuse.

In this chapter, we develop a comprehensive formal framework for the design and analysis of database access control. Our framework, which relies on the operational semantics from Chapter 5, consists of a precise attacker model complemented with adequate security properties. We use our framework to design and verify an access control mechanism that prevents confidentiality and integrity attacks that defeat existing mechanisms. Note that this chapter is largely based on [87].

Organization. In Section 6.1, we present attacks that illustrate serious weaknesses in existing DBMSs. We introduce and formalize our attacker model in Section 6.2. We define our security properties in Sections 6.3 and 6.4. In Section 6.5, we present our provably secure DBAC mechanism, and in Section 6.6 we discuss related work. Finally, we draw conclusions in Section 6.7. For the sake of readability, some technical details are given in Section 6.8. Furthermore, the proof of all our results are given in Appendix C. A prototype of our enforcement mechanism is available at [86].

6.1 Illustrative Attacks

We demonstrate here how attackers can exploit existing DBMSs using standard SQL features. We classify these attacks as either *Integrity Attacks* or *Confidentiality Attacks*. In the former, an attacker makes unauthorized changes to the database, which stores the data, the policy, the triggers, and the views. In the latter, an attacker learns sensitive data by interacting with the system and observing the outcome. No existing access control mechanism prevents all the attacks we present. Moreover, many related attacks can be constructed using variants of the ideas presented here. We manually carried out the attacks against IBM DB2, Oracle Database, PostgreSQL, MySQL, SQL Server, and Firebird. We summarize our findings in Figures 6.1 and 6.2, and we discuss them at the end of this section. In the figures, ✓ indicates a successful attack, ✗ indicates a failed attack, and ∗ indicates that the DBMS does not support the features necessary to launch the attack.

DBMS	Integrity Attacks		
	Triggers with activator's privileges	Granting views	Revoking views
IBM DB2 (v. 10.5)	*	✗	✓
Oracle (v. 11g)	*	✗	✗
PostgreSQL (v. 9.3.5)	✓	✓	✓
MySQL (v. 14.14)	*	✗	✓
SQL Server (v. 12.0)	✓	*	*
Firebird (v. 2.5.2)	✓	✗	✓

FIGURE 6.1: Summary of the integrity attacks.

DBMS	Confidentiality Attacks	
	Table updates and integrity constraints	Triggers with owner's privileges
IBM DB2 (v. 10.5)	✓	✓
Oracle (v. 11g)	✓	✓
PostgreSQL (v. 9.3.5)	✓	✓
MySQL (v. 14.14)	✓	✓
SQL Server (v. 12.0)	✓	✓
Firebird (v. 2.5.2)	✓	✓

FIGURE 6.2: Summary of the confidentiality attacks.

6.1.1 Integrity Attacks

Our three integrity attacks combine different database features: `INSERT`, `DELETE`, `GRANT`, and `REVOKE` commands together with views and triggers. In the first attack, an attacker creates a trigger, i.e., a procedure automatically executed by the DBMS in response to user commands, that will be activated by an unaware user with a higher security clearance and will perform unauthorized changes to the database. The attack requires triggers to be executed under the privileges of the users activating them. Such triggers are supported by PostgreSQL, SQL Server, and Firebird.

Attack 6.1. Triggers with activator's privileges. Consider a database with two tables P and S and two users u_1 and u_2. The attacker is the user u_1, whose goal is to delete the content of S. The policy is that u_1 is not authorized[1] to alter S, u_1 can create triggers on P, and u_2 can read and modify S and P. The attack is as follows:

1. u_1 creates the trigger:

```
CREATE TRIGGER t ON P AFTER INSERT
  DELETE FROM S;
```

2. u_1 waits until u_2 inserts a tuple into the table P. The trigger will then be invoked using u_2's privileges and S's content will be deleted. ■

An attacker can use similar attacks to execute arbitrary commands with administrative privileges. Despite the threat posed by such simple attacks, the existing countermeasures [4] are unsatisfactory; they are either too restrictive, for instance completely disabling triggers in the database, or too time consuming and error prone, namely manually checking if "dangerous" triggers have been created.

In our second attack, an attacker escalates his privileges by delegating the read permission for a table without being authorized to delegate this permission. The attacker first creates a view over the table and, afterwards, delegates the access to the view to another user. This attack exploits DBMSs, such as PostgreSQL, where a user can grant any read permission over his own views. Note that `GRANT` and `REVOKE` commands are *write operations*, which target the database's internal configuration instead of the tables.

Attack 6.2. Granting views. Consider a database with a table S, two users u_1 and u_2, and the following policy: u_1 can create views and read S (without being able to delegate this permissions), and u_2 cannot read S. The attack is as follows:

1. u_1 creates the view: `CREATE VIEW` v `AS SELECT` $*$ `FROM` S.
2. u_1 issues the command `GRANT SELECT ON` v `TO` u_2. Now, u_2 can read S through v. However, u_1 is not authorized to delegate the read permission on S. ■

This attack exploits several subtleties in the commands' semantics: (a) users can create views over all tables they can read, (b) the views are executed under the owner's privileges, and (c) view's owners

[1] As is common in SQL, a user is authorized to execute a command if and only if the policy assigns him the corresponding permission.

can grant arbitrary permissions over their own views. These features give u_1 the implicit ability to delegate the read access over S. As a result, the overall system's behavior does not conform with the given policy. That is, u_1 should not be permitted to delegate the read access to S or to any view that depends on it. Note that the commands' semantics may vary between different DBMSs.

In our third attack, an attacker exploits the failure of access control mechanisms to propagate `REVOKE` commands.

Attack 6.3. Revoking views. Consider a database with a table S, three users u_1, u_2, and u_3, and the following policy: u_1 can read S and delegate this permission, u_2 can create views, and u_3 cannot read S. The attack proceeds as follows:

1. u_1 issues the command `GRANT SELECT ON` S `TO` u_2 `WITH GRANT OPTION`.
2. u_2 creates the view: `CREATE VIEW` v `AS SELECT` $*$ `FROM` S.
3. u_2 issues the command `GRANT SELECT ON` v `TO` u_3.
4. u_1 revokes the permission to read S (and to delegate the permission) from u_2: `REVOKE SELECT ON` S `FROM` u_2. Now, u_3 cannot read v because u_2, which is v's owner, cannot read S.
5. u_1 grants again the permission to read S to u_2: `GRANT SELECT ON` S `TO` u_2. Now, u_3 can again read v but u_2 can no longer delegate the read permission on v. ■

This attack succeeds because, in the fourth step, the `REVOKE` statement does not remove the `GRANT` granted by u_2 to u_3 to read v. This `GRANT` only becomes ineffective because u_2 is no longer authorized to read S. However, after the fifth step, this `GRANT` becomes effective again, even though u_2 can no longer delegate the read permission on v. Thus, the policy is left in an inconsistent state, i.e., it contains a privilege that should have been revoked according to the SQL access control model [137,162].

6.1.2 Confidentiality Attacks

We now present two attacks that use `INSERT` and `SELECT` commands together with triggers and integrity constraints. In our fourth attack, an attacker exploits integrity constraint violations to learn sensitive information. An integrity constraint is an invariant that must be satisfied for a database state to be considered *valid*. *Integrity constraint violations* arise when the execution of an SQL command leads the database from a valid state into an invalid one.

Attack 6.4. Table updates and integrity constraints. Consider a database with two tables P and S. Suppose the primary key of both tables is the user's identifier. Furthermore, the set of user identifiers in S is contained in the set of user identifiers in P, i.e., there is a foreign key from S to P. The attacker is the user u whose goal is to learn whether `Bob` is in S. The security policy is that u can read P and insert tuples in S. The attacker u can learn whether `Bob` is in S as follows:

1. He reads P and learns `Bob`'s identifier.
2. He issues an `INSERT` statement in S using `Bob`'s id.
3. If `Bob` is already in S, then u gets an error message about the primary key's violation. Alternatively, there is no violation and u learns that `Bob` is not in S. ■

Although similar attacks have been identified before [99,143], existing DBMSs are still vulnerable.

In our fifth attack, an attacker learns sensitive information by exploiting the system's triggers. The trigger in this attack is executed under the privileges of the trigger's owner. Such triggers are supported by IBM DB2, Oracle Database, PostgreSQL, MySQL, SQL Server, and Firebird.

Attack 6.5. Triggers with owner's privileges. Consider a database with three tables N, P, and T. The attacker is the user u, who wishes to learn whether v is in T. The policy is that u is not authorized to read the table T, and he can read and modify the tables N and P. Moreover, the following trigger has been defined by the administrator.

```
CREATE TRIGGER t ON P AFTER INSERT FOR EACH ROW
  IF exists(SELECT * FROM T WHERE id = NEW.id)
    INSERT INTO N VALUES (NEW.id);
```

The attack is as follows:

1. u deletes v from N.
2. u issues the command `INSERT INTO` P `VALUES (`v`)`.
3. u checks the table N. If it contains v's id, then v is in T. Otherwise, v is not in T. ■

This attack exploits that the trigger t conditionally modifies the database. Furthermore, the attacker can activate t, by inserting tuples in P, and then observe t's effects, by reading the table N. He therefore can exploit t's execution to learn whether t's condition holds. We assume here that the attacker knows the triggers in the system. This is, in general, a weak assumption as triggers usually describe the domain-specific rules regulating a system's behavior and users are usually aware of them.

6.1.3 Discussion

We manually carried out all five attacks against IBM DB2, Oracle Database, PostgreSQL, MySQL, SQL Server, and Firebird. Figures 6.1 and 6.2 summarize our findings. None of these systems prevent the confidentiality attacks. They are however more successful in preventing the integrity attacks. The most successful is Oracle Database, which prevents two of the three attacks, while Attack 6.1 cannot be carried out due to missing features. IBM DB2, MySQL, and Firebird prevent just one of the three attacks, namely Attack 6.2. However, they all fail to prevent Attack 6.3. Note that Firebird also fails to prevent Attack 6.1. In contrast, Attack 6.1 cannot be carried out against MySQL and IBM DB2 due to missing features. SQL Server also fails to prevent Attack 6.1; however the remaining two attacks cannot be carried out due to missing features. PostgreSQL fails to prevent all three attacks.

We argue that the dire state of database access control mechanisms, as illustrated by these attacks, comes from the lack of clearly defined security properties that such mechanisms ought to satisfy and the lack of a well-defined attacker model. We therefore develop a formal attacker model and precise security properties and we use them to design a provably secure access control mechanism that prevents all the above attacks.

6.2 Attacker Model

We next present our attacker model. In our setting, an *attacker* is a user, other than the administrator, with an assigned user account who attempts to violate the security policy. Namely, his goals are: (1) to read or infer data from the database for which he lacks the necessary `SELECT` privileges, and (2) to alter the system state in unauthorized ways, e.g., changing data in relations for which he lacks the necessary `INSERT` and `DELETE` privileges. The attacker can issue any command available to users and he sees the results of his commands (see Chapter 5 for all supported commands). The attacker's inference capabilities are specified using deduction rules.

6.2.1 Formal Attacker Model

We model attackers that interact with the system through SQL commands and infer information from the system's behavior by exploiting triggers, views, and integrity constraints. We argue that database access control mechanisms should be secure with respect to such strong attackers, as this reflects how (malicious) users may interact with modern databases. Furthermore, any mechanism secure against such strong attackers is also secure against weaker attackers.

Any user other than the administrator can be an attacker, and we assume that users do not collude to subvert the system. Note that our attacker model, the security properties in Sections 6.3 and 6.4, and the mechanism we develop in Section 6.5 can all easily be extended to support colluding users. We also assume that an attacker can issue any command available to the system's users, and he knows the system's operational semantics, the database schema, and the integrity constraints.

We assume that an attacker has access to the system's security policy, the set of users, and the definitions of the triggers and views in the system's state. In more detail, given an M-runtime state $\langle db, U, sec, T, V, ctx\rangle$, an attacker can access U, sec, T, and V. Users interacting with existing DBMSs typically have access to some, although not all, of this information. For instance, in PostgreSQL a user can read all the information about the triggers defined on the tables for which he has some non-`SELECT` privileges. Note that the more information an attacker has, the more attacks he can launch. Finally, we assume that an attacker knows whether any two of his commands c and c' have been executed consecutively by the system, i.e., if there are commands executed by other users occurring between c and c'. The attacker's knowledge about the sequential execution of his commands is needed to soundly propagate his knowledge about the system's state between his commands. Since the mechanism we develop in Section 6.5 is secure with respect to this attacker, it is also secure with respect to weaker attackers who have less information or cannot detect whether their commands have been executed consecutively.

An attacker model describes what information an attacker knows, how he interacts with the system, and what he learns about the system's data by observing the system's behavior. Since every user is a potential attacker, for each user $u \in \mathcal{U}$ we define an attacker model specifying u's inference capabilities. To represent u's knowledge, we introduce judgments. A judgment is a four-tuple $\langle r, i, u, \phi\rangle$, written $r, i \vdash_u \phi$, denoting that from the run r, which represents the system's behavior, the user u can infer that ϕ holds in the i-th state of r. An attacker model for u is thus a set of judgments associating to each position of each run, the sentences that u can infer from the system's behavior. The idea of representing the attacker's knowledge using sentences ϕ is inspired by existing formalisms for Database Inference Control [40, 72] and Controlled Query Evaluation [36].

Definition 6.1. Let P be an extended configuration, L be the P-LTS, and $u \in \mathcal{U}$ be a user. A (P, u)-*judgment* is a tuple $\langle r, i, u, \phi\rangle$, written $r, i \vdash_u \phi$, where $r \in traces(L)$, $1 \leq i \leq |r|$, and

$$\textsc{DELETE Success}\quad \frac{r^i = r^{i-1}\cdot\langle u, \texttt{DELETE}, R, \bar{t}\rangle\cdot s \qquad 1 < i \leq |r| \qquad s \in \Omega_M \qquad \mathit{secEx}(s) = \bot \qquad \mathit{Ex}(s) = \emptyset}{r, i \vdash_u \neg R(\bar{t})}$$

$$\textsc{SELECT Success}\quad \frac{r^i = r^{i-1}\cdot\langle u, \texttt{SELECT}, \phi\rangle\cdot s \qquad 1 < i \leq |r| \qquad s \in \Omega_M \qquad \mathit{secEx}(s) = \bot \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{res}(s) = \top}{r, i \vdash_u \phi}$$

$$\textsc{Learn INSERT Backward}\quad \frac{\begin{array}{c} r^{i+1} = r^i\cdot t\cdot s \qquad \mathit{invoker}(\mathit{last}(r^i)) = u \qquad s \in \Omega_M \qquad 1 \leq i < |r| \qquad \mathit{secEx}(s) = \bot \\ \mathit{Ex}(s) = \emptyset \qquad r, i \vdash_u \neg\psi \qquad r, i+1 \vdash_u \psi \qquad t = \langle id, ow, ev, R', \phi(\overline{x}), \langle \texttt{INSERT}, R, \bar{t}\rangle, m\rangle \end{array}}{r, i \vdash_u \phi[\overline{x} \mapsto \mathit{tpl}(\mathit{last}(r^i))]}$$

$$\textsc{Propagate Backward SELECT}\quad \frac{r^{i+1} = r^i\cdot\langle u, \texttt{SELECT}, \phi\rangle\cdot s \qquad r, i+1 \vdash_u \psi \qquad s \in \Omega_M \qquad 1 \leq i < |r|}{r, i \vdash_u \psi}$$

$$\textsc{Propagate Forward Update Success}\quad \frac{\begin{array}{c} r, i-1 \vdash_u \phi \qquad r^i = r^{i-1}\cdot\langle u, op, R, \bar{t}\rangle\cdot s \qquad s \in \Omega_M \\ 1 < i \leq |r| \qquad \mathit{secEx}(s) = \bot \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{revise}(r^{i-1}, \phi, r^i) = \top \qquad op \in \{\texttt{INSERT}, \texttt{DELETE}\} \end{array}}{r, i \vdash_u \phi}$$

FIGURE 6.3: Example of attacker inference rules, where $r, i \vdash_u \phi$ denotes that this judgment holds in $\mathcal{ATK}_u$.

$\phi \in \mathit{RC}_{\mathit{bool}}$. A (P, u)-*attacker model* is a set of (P, u)-judgments. A (P, u)-judgment $r, i \vdash_u \phi$ *holds in a* (P, u)-*attacker model* A iff $r, i \vdash_u \phi \in A$. □

For each user $u \in \mathcal{U}$, we now define the (P, u)-attacker model $\mathcal{ATK}_u$ that we use in the rest of the chapter. We formalize this model using a set of inference rules, where $\mathcal{ATK}_u$ is the smallest set of judgments satisfying the inference rules. Figure 6.3 shows five representative rules. The complete formalization of all rules is given in Section 6.8.1. In the following, when we say that a judgment $r, i \vdash_u \phi$ holds, we always mean with respect to the attacker model $\mathcal{ATK}_u$.

As we prove in Appendix C, $\mathcal{ATK}_u$ is sound with respect to the RC semantics, i.e., if $r, i \vdash_u \phi$ holds, then the formula ϕ holds in the i-th state of r. Intuitively, $\mathcal{ATK}_u$ models how u infers information from the system's behavior, namely (a) how u learns information from his commands and their results, (b) how u learns information from triggers, their execution, their interleavings, and their side effects, (c) how u propagates his knowledge along a run, and (d) how u learns information from exceptions caused by either integrity constraint violations or security violations. This model is substantially more powerful than the `SELECT`-only attacker studied in previous works [131, 165, 170].

The rules `DELETE` Success and `SELECT` Success describe how the user u infers information from his successful actions, i.e., those actions that generate neither security exceptions nor integrity violations. In the rules, $\mathit{secEx}(s) = \bot$ denotes that there were no security exceptions caused by the action leading to s, and $\mathit{Ex}(s) = \emptyset$ denotes that the action leading to s has not violated the integrity constraints. After a successful `DELETE`, u knows that the deleted tuple is no longer in the database, and after a successful `SELECT` he learns the query's result, denoted by $\mathit{res}(s)$.

The rules Propagate Backward `SELECT` and Propagate Forward Update Success describe how the user u propagates information along the run. Propagate Backward `SELECT` states that if the user u knows that ϕ holds after a `SELECT` command, then he knows that ϕ also holds just before the `SELECT` command because `SELECT` commands do not modify the database state. Propagate Forward Update Success states that if u knows that ϕ holds before a successful `INSERT` or `DELETE` command and he can determine that the command's execution does not influence ϕ's truth value, denoted by $\mathit{revise}(r^{i-1}, \phi, r^i) = \top$, then he also knows that ϕ holds after the command.

In our attacker model, we consider a very simple syntactic criterion for revising beliefs. Informally, the attacker is able to propagate the knowledge of a sentence ϕ after (or before) an `INSERT` or a `DELETE` action on a table R iff the predicate R does not occur in ϕ. Formally, the function $\mathit{revise} : \mathit{traces}(L) \times \mathit{RC}_{\mathit{bool}} \times \mathit{traces}(L) \to \{\top, \bot\}$ captures this belief revision approach and it is defined as follows:

$$\mathit{revise}(r, \phi, r\cdot \mathit{act}\cdot s) = \begin{cases} \top & \text{if } \mathit{act} = \langle u, op, R, \bar{t}\rangle \wedge R \notin \mathit{tables}(\phi) \wedge op \in \{\texttt{INSERT}, \texttt{DELETE}\} \\ \top & \text{if } \mathit{act} = \langle id, ow, ev, R', \phi, \langle op, R, \bar{t}\rangle, m\rangle \wedge R \notin \mathit{tables}(\phi) \wedge \\ & \quad op \in \{\texttt{INSERT}, \texttt{DELETE}\} \\ \top & \text{if } \mathit{act} = \langle id, ow, ev, R, \phi, \langle op, u, p\rangle, m\rangle \wedge op \in \{\oplus, \oplus^*, \ominus\} \\ \bot & \text{otherwise} \end{cases}$$

$$\dfrac{\dfrac{\dfrac{}{r, 2 \vdash_u \neg N(v)}\ \textsc{DELETE Success}}{r, 3 \vdash_u \neg N(v)}\ \textsc{Propagate Forward Update Success} \qquad \dfrac{\dfrac{}{r, 5 \vdash_u N(v)}\ \textsc{SELECT Success}}{r, 4 \vdash_u N(v)}\ \textsc{Propagate Backward SELECT}}{r, 3 \vdash_u T(v)}\ \textsc{Learn INSERT Backward}$$

FIGURE 6.4: Template Derivation of Attack 6.5 (contains just selected subgoals)

Observe that above we define *revise* only for the inputs r, ϕ, r' such that $\phi \in RC_{bool}$ is a sentence and $r' = r{\cdot}act{\cdot}s$, where $act \in \mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$ and $s \in \Omega_M$. If this is not the case, then $revise(r, \phi, r') = \bot$. Our attacker model, enforcement mechanism, and proofs can be easily extended to support other criteria for revising beliefs.

Finally, the rule LEARN `INSERT` BACKWARD models u's reasoning when he activates a trigger that successfully inserts a tuple in the database. If u knows that immediately before the trigger the formula ψ does not hold and immediately after the trigger the formula ψ holds, then the trigger's execution is the cause of the database state's change. Therefore, u can infer that the trigger's condition ϕ holds just before the trigger's execution. Note that $invoker(s)$ denotes the user who fired the trigger that is executed in the state s, whereas $tpl(s)$ denotes the tuple associated with the action that fired the trigger that is executed in the state s.

We developed an executable version of our attacker model, available at [86], using the Maude term rewriting framework [54]. Our executable models can be used for simulating the execution of our operational semantics, as well as computing the information that an attacker can infer from the system's behavior. We have executed and validated all of the examples in this chapter using these models.

Example 6.1. Consider the policy described in Attack 6.5. The database D has three tables: N, P, and T. The set U is $\{u, admin\}$ and the policy S contains the following grants: $\langle \oplus, u, \langle \texttt{SELECT}, P\rangle, admin\rangle$, $\langle \oplus, u, \langle \texttt{INSERT}, P\rangle, admin\rangle$, $\langle \oplus, u, \langle \texttt{DELETE}, P\rangle, admin\rangle$, $\langle \oplus, u, \langle \texttt{SELECT}, N\rangle, admin\rangle$, $\langle \oplus, u, \langle \texttt{INSERT}, N\rangle, admin\rangle$, and $\langle \oplus, u, \langle \texttt{DELETE}, N\rangle, admin\rangle$. Furthermore, the database state db is $db(N) = \{v\}$, $db(P) = \emptyset$, and $db(T) = \{v\}$, while the only trigger in the system is $t = \langle id, admin, INS, P, T(x_1), \langle \texttt{INSERT}, N, x_1\rangle, O\rangle$. The run r is as follows:

1. u deletes v from N.
2. u inserts v in P. This activates the trigger t, which inserts v in N.
3. u issues the `SELECT` query $N(v)$.

We used Maude to generate the following run, which illustrates how the system's state changes. Note that there are no exceptions during the run.

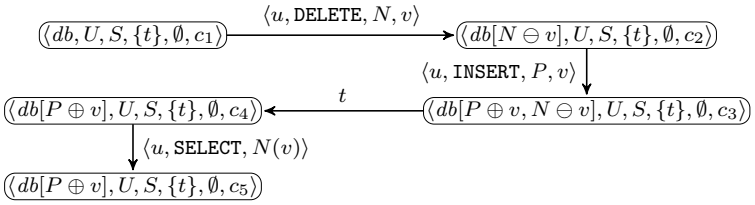


Figure 6.4 models u's reasoning in Attack 6.5. The user u first applies the `SELECT` SUCCESS rule to derive $r, 5 \vdash_u N(v)$, i.e., he learns the query's result. By applying the rule PROPAGATE BACKWARD `SELECT` to $r, 5 \vdash_u N(v)$, he obtains $r, 4 \vdash_u N(v)$, i.e., he learns that $N(v)$ holds before the `SELECT` query. Similarly, he applies the rule `DELETE` SUCCESS to derive $r, 2 \vdash_u \neg N(v)$, and he obtains $r, 3 \vdash_u \neg N(v)$ by applying the PROPAGATE FORWARD UPDATE SUCCESS rule. Finally, by applying the rule LEARN `INSERT` BACKWARD to $r, 3 \vdash_u \neg N(v)$ and $r, 4 \vdash_u N(v)$, he learns the value of the trigger's `WHEN` condition $r, 3 \vdash_u T(v)$. Since the user u should not be able to learn information about T, the attack violates the intended confidentiality guarantees. We used our executable attacker model [86] to derive the judgments. ■

6.3 Database Integrity

Here we define the first of our security properties: database integrity. Database integrity states that all actions modifying the system's state are authorized by the system's policy. An access control mechanism providing database integrity prevents non-authorized changes to the system's state and, thereby, prevents integrity attacks, such as Attacks 6.1–6.3.

Database integrity requires a formalization of authorized actions. We therefore define the relation $\rightsquigarrow_{auth}$ between runtime states and actions, modeling which actions are authorized in a given state.

We first introduce some notation. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ and f is an M-PDP. We denote by $\mathcal{VIEW}_D^{owner}$ the set of all D-views with owner's privileges,

i.e., $\mathcal{VIEW}_D^{owner} = \{\langle V, o, q, m\rangle \in \mathcal{VIEW}_D \mid m = O\}$ and by $\mathcal{PRIV}_D^{\texttt{SELECT}, \mathcal{VIEW}_D^{owner}}$ the set of all `SELECT` privileges associated with views in $\mathcal{VIEW}_D^{owner}$, i.e., $\mathcal{PRIV}_D^{\texttt{SELECT}, \mathcal{VIEW}_D^{owner}} = \{\langle \texttt{SELECT}, V\rangle \mid V \in \mathcal{VIEW}_D^{owner}\}$. Furthermore, given an M-runtime state $s = \langle db, U, sec, T, V, c\rangle$ and a `REVOKE` command $r = \langle \ominus, u, p, u'\rangle$, we denote by $applyRev(s, r)$ the state $\langle db, U, revoke(sec, u, p, u'), T, V, c\rangle$ obtained by executing the `REVOKE` command. Finally, given a query q, a set of views V with owner's privileges, and a set of tables T, we write $determines_M(T, V, q)$ to denote that $D, \Gamma \vdash Q \twoheadrightarrow q$, where Q is the set of queries containing the tables in T and the views in V (see Section 2.2.3 for a formalization of query determinacy).

We are now ready to formalize the notion of authorized actions. The relation $\rightsquigarrow_{auth} \subseteq \Omega_M \times (\mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D)$, specifying which actions are authorized in a given state, is the smallest relation satisfying the inference rules given in Figure 6.5. In the following, let $s = \langle db, T, sec, T, V, ctx\rangle$ be a runtime state. According to the Add User rule, the only authorized `ADD USER` commands are those issued by the administrator. In contrast, `SELECT` commands are always authorized, as they do not modify the system state. Furthermore, an `INSERT`, `DELETE`, `CREATE VIEW`, or `CREATE TRIGGER` command is authorized in two cases: (1) the user executing the command is the administrator (see the rules Insert-Delete admin, Create View admin, and Create Trigger admin), or (2) the current policy sec contains a grant g that grants the corresponding privilege to the user executing the command and such that $s \rightsquigarrow_{auth} g$ holds (see the rules Insert-Delete, Create View, and Create Trigger).

The Revoke rule says that a `REVOKE` statement is authorized if the resulting state, obtained using the function $applyRev$, has a consistent policy, namely one in which all the grants in the policy are authorized by $\rightsquigarrow_{auth}$. The rules Grant-1, Grant-2, Grant-3, Grant-4, and Grant-5, instead, formalize in which cases a `GRANT` command is authorized. Specifically, a `GRANT` command is authorized if one of the following conditions hold:

- Rule Grant-1: the user executing the command is not the administrator and there is a grant g (with grant option) that authorizes the command and $s \rightsquigarrow_{auth} g$.
- Rule Grant-2: the user executing the command is the administrator and the privilege is not associated with a view with the owner's privileges.
- Rule Grant-3: the privilege used in the command is a `SELECT` privilege over a view V in $\mathcal{VIEW}_D^{owner}$, the user u executing the command is V's owner, and u has the `SELECT` privilege with grant option over a set of tables and views that determine V.
- Rule Grant-4: the privilege used in the command is a `SELECT` privilege over a view V in $\mathcal{VIEW}_D^{owner}$, the user executing the command is the administrator, and V's owner has the `SELECT` privilege over a set of tables and views that determine V.
- Rule Grant-5: the privilege used in the command is a `SELECT` privilege over a view with the activator's privileges and the user executing the command is the view's owner.

The rules Grant-3 and Grant-4 use the function $hasAccess$ to decide whether a user has read access (with or without delegation) to a set of views and tables. Formally, the $hasAccess$ function is defined as follows:

$$hasAccess(\langle db, U, sec, T, V, c\rangle, S, u, op) = \begin{cases} \top & \text{if } u \neq admin \wedge \forall v \in S.\, \exists u'' \in U, g \in sec, op' \in \{op, \oplus^*\}.\\ & \quad g = \langle op', u, \langle \texttt{SELECT}, v\rangle, u''\rangle \wedge\\ & \quad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} g\\ \top & \text{if } u = admin \wedge \forall v \in S.\, \exists u'' \in U, op' \in \{op, \oplus^*\}.\\ & \quad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle op', u, \langle \texttt{SELECT}, v\rangle, u''\rangle\\ \bot & otherwise \end{cases}$$

The above function performs the following check. If the user u is not the administrator, $hasAccess$ checks that for all views and tables $v \in S$, there is a `GRANT` g in the current policy that is valid according to $\rightsquigarrow_{auth}$ and that grants the privilege $\langle \texttt{SELECT}, v\rangle$ (with or without `GRANT OPTION` depending on op) to the user u. In case the user u is the administrator, $hasAccess$ checks that for all views and tables in $v \in S$, the `GRANT` $\langle op', admin, \langle \texttt{SELECT}, v\rangle, u''\rangle$ is valid according to $\rightsquigarrow_{auth}$, where op' is $\oplus$ or $\oplus^*$ depending on op and u'' is a user in U.

Finally, the rules Execute Trigger-1, Execute Trigger-2, and Execute Trigger-3 regulate the execution of triggers. According to the rule Execute Trigger-1, a trigger with the owner's privileges whose `WHEN` condition is satisfied is authorized if the trigger's owner is authorized to execute the trigger's action. The Execute Trigger-2 rule says that the execution of a trigger (whose `WHEN` condition is satisfied) with the activator's privileges is authorized if both the invoker and the trigger's owner are authorized to execute the trigger's action according to $\rightsquigarrow_{auth}$. Finally, the Execute Trigger-3 rule says that the execution of triggers whose `WHEN` condition is not satisfied is always authorized, as these triggers do not modify the system state. Observe that the *action* function instantiates the action given in the trigger's definition to a concrete action by identifying the user performing the action and replacing the free variables with values from **dom**.

We now define database integrity. Intuitively, a PDP provides database integrity iff all the actions it authorizes are explicitly authorized by the policy, i.e., they are authorized by $\rightsquigarrow_{auth}$. This notion comes directly from the SQL standard, and it is reflected in existing enforcement mechanisms. Recall

$$\textsc{Add User}\quad \frac{u \in \mathcal{U} \qquad u' = admin}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle u', \texttt{ADD\_USER}, u\rangle} \qquad \textsc{Select}\quad \frac{u \in U \qquad q \in RC}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle u, \texttt{SELECT}, q\rangle}$$

$$\textsc{Insert-Delete}\quad \frac{\begin{array}{c} u, u' \in U \qquad R \in D \qquad \bar{t} \in \mathbf{dom}^{|R|} \\ g = \langle op, u, \langle op', R\rangle, u'\rangle \qquad g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} g \qquad op' \in \{\texttt{INSERT}, \texttt{DELETE}\} \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle u, op', R, \bar{t}\rangle}$$

$$\textsc{Create View}\quad \frac{u, u' \in U \qquad v \in \mathcal{VIEW}_D \qquad g = \langle op, u, \langle \texttt{CREATE VIEW}\rangle, u'\rangle \qquad g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} g}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle u, \texttt{CREATE}, v\rangle}$$

$$\textsc{Create Trigger}\quad \frac{\begin{array}{c} u, u' \in U \qquad t = \langle id, ow, ev, R, \phi, stmt, m\rangle \\ t \in \mathcal{TRIGGER}_D \qquad g = \langle op, u, \langle \texttt{CREATE TRIGGER}, R\rangle, u'\rangle \qquad g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} g \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle u, \texttt{CREATE}, t\rangle}$$

$$\textsc{Insert-Delete admin}\quad \frac{R \in D \qquad \bar{t} \in \mathbf{dom}^{|R|} \qquad op' \in \{\texttt{INSERT}, \texttt{DELETE}\}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle admin, op', R, \bar{t}\rangle} \qquad \textsc{Create View admin}\quad \frac{v \in \mathcal{VIEW}_D \qquad v = \langle id, admin, q, m\rangle}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle admin, \texttt{CREATE}, v\rangle}$$

$$\textsc{Create Trigger admin}\quad \frac{t = \langle id, admin, ev, R, \phi, stmt, m\rangle \qquad t \in \mathcal{TRIGGER}_D}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle admin, \texttt{CREATE}, t\rangle}$$

$$\textsc{Revoke}\quad \frac{\begin{array}{c} u, u' \in U \qquad priv \in \mathcal{PRIV}_D \qquad s = \langle db, U, sec, T, V, c\rangle \\ s' = \langle db, U, sec', T, V, c\rangle \qquad s' = applyRev(s, \langle \ominus, u, p, u'\rangle) \qquad \forall g \in sec'.\ s' \rightsquigarrow_{auth} g \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle \ominus, u, priv, u'\rangle}$$

$$\textsc{Grant-1}\quad \frac{\begin{array}{c} u, u', u'' \in U \\ op \in \{\oplus, \oplus^*\} \qquad priv \in \mathcal{PRIV}_D \qquad g = \langle \oplus^*, u', priv, u''\rangle \qquad g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} g \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle op, u, priv, u'\rangle}$$

$$\textsc{Grant-2}\quad \frac{u \in U \qquad op \in \{\oplus, \oplus^*\} \qquad priv \in \mathcal{PRIV}_D \setminus \mathcal{PRIV}_D^{\texttt{SELECT}, \mathcal{VIEW}^{owner}{}_D}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle op, u, priv, admin\rangle}$$

$$\textsc{Grant-3}\quad \frac{\begin{array}{c} u, owner \in U \\ op \in \{\oplus, \oplus^*\} \qquad priv = \langle \texttt{SELECT}, v\rangle \qquad v = \langle id, owner, q, O\rangle \qquad v \in V \qquad V' \subseteq V \cap \mathcal{VIEW}_D^{owner} \\ T' \subseteq D \qquad determines_M(T', V', q) \qquad hasAccess(\langle db, U, sec, T, V, c\rangle, V' \cup T', owner, \oplus^*) \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle op, u, priv, owner\rangle}$$

$$\textsc{Grant-4}\quad \frac{\begin{array}{c} u, owner \in U \qquad op \in \{\oplus, \oplus^*\} \\ priv = \langle \texttt{SELECT}, v\rangle \qquad v = \langle id, owner, q, O\rangle \qquad v \in V \qquad owner \neq admin \qquad V' \subseteq V \cap \mathcal{VIEW}_D^{owner} \\ T' \subseteq D \qquad determines_M(T', V', q) \qquad hasAccess(\langle db, U, sec, T, V, c\rangle, V' \cup T', owner, \oplus) \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle op, u, priv, admin\rangle}$$

$$\textsc{Grant-5}\quad \frac{u, owner \in U \qquad op \in \{\oplus, \oplus^*\} \qquad v \in V \qquad priv = \langle \texttt{SELECT}, v\rangle \qquad v = \langle id, owner, q, A\rangle}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} \langle op, u, priv, owner\rangle}$$

$$\textsc{Execute Trigger-1}\quad \frac{\begin{array}{c} t = \langle id, ow, ev, R, \phi, stmt, O\rangle \\ t \in T \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} action(stmt, ow, tpl(c)) \qquad [\phi[\overline{x}^{|R|} \mapsto tpl(c)]]^{db} = \top \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} t}$$

$$\textsc{Execute Trigger-2}\quad \frac{\begin{array}{c} t = \langle id, ow, ev, R, \phi, stmt, A\rangle \qquad t \in T \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} action(stmt, invoker(c), tpl(c)) \\ \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} action(stmt, ow, tpl(c)) \qquad [\phi[\overline{x}^{|R|} \mapsto tpl(c)]]^{db} = \top \end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} t}$$

$$\textsc{Execute Trigger-3}\quad \frac{t = \langle id, ow, ev, R, \phi, stmt, m\rangle \qquad t \in T \qquad [\phi[\overline{x}^{|R|} \mapsto tpl(c)]]^{db} = \bot}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} t}$$

Figure 6.5: Definition of the $\rightsquigarrow_{auth}$ relation.

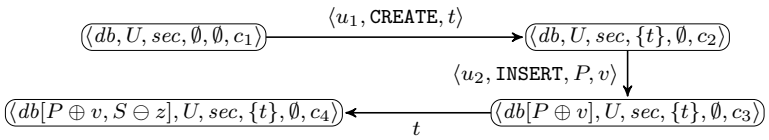
that, given a state s, $secEx(s) = \bot$ denotes that there were no security exceptions caused by the action or trigger leading to s.

Definition 6.2. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ and f is an M-PDP, and let L be the P-LTS. We say that f *provides database integrity with respect to* P iff for all reachable states $s, s' \in \Omega_M$, if s' is reachable in one step from s by an action $a \in \mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$ and $secEx(s') = \bot$, then $s \rightsquigarrow_{auth} a$. □

Example 6.2. We consider a run corresponding to Attack 6.1, which illustrates a violation of database integrity. The database db is such that $db(P) = \emptyset$ and $db(S) = \{z\}$, the policy sec is $\{\langle\oplus, u_1, \langle \texttt{CREATE TRIGGER}, P\rangle, admin\rangle, \langle\oplus, u_2, \langle \texttt{INSERT}, P\rangle, admin\rangle, \langle\oplus, u_2, \langle \texttt{DELETE}, S\rangle, admin\rangle, \langle\oplus, u_2, \langle \texttt{SELECT}, P\rangle, admin\rangle, \langle\oplus, u_2, \langle \texttt{SELECT}, S\rangle, admin\rangle\}$, and the set U is $\{u_1, u_2, admin\}$. The run r is as follows:

1. The user u_1 creates the trigger $t = \langle id, u_1, INS, P, \top, \langle \texttt{DELETE}, S, z\rangle, A\rangle$.
2. The user u_2 inserts the value v in P. This activates the trigger t and deletes the content of S, i.e., the value z.

We used Maude to generate the following run, which illustrates how the system's state changes. Note that there are no exceptions during the run.



Access control mechanisms that do not restrict the execution of triggers with activator's privileges violate database integrity because they do not throw security exceptions when $\langle db[P \oplus v], U, sec, \{t\}, \emptyset, c_3\rangle \not\rightsquigarrow_{auth} t$. ■

6.4 Data Confidentiality

We now introduce data confidentiality, our second security property. Data confidentiality states that all information that an attacker can learn by observing the system's behavior is authorized. Hence, by preventing the leakage of sensitive data, a mechanism providing data confidentiality prevents confidentiality attacks.

We first introduce indistinguishability and secure judgments, two key notions for data confidentiality, together with our security property. Afterwards, we formalize the notion of indistinguishability that we use in the rest of the chapter. We conclude the section with some examples.

6.4.1 Definition

To model data confidentiality, we first introduce the concept of indistinguishability of runs, which formalizes the desired confidentiality guarantees by specifying whether users can distinguish between different runs based on their observations. Formally, a *P-indistinguishability relation* is an equivalence relation over $traces(L)$, where P is an extended configuration and L is the P-LTS. Indistinguishable runs, intuitively, should disclose the same information.

We now define the concept of a secure judgment, which is a judgment that does not leak sensitive information or, equivalently, one that cannot be used to differentiate between indistinguishable runs.

Definition 6.3. Let P be an extended configuration, L be the P-LTS, and $\cong$ be a P-indistinguishability relation. A judgment $r, i \vdash_u \phi$ is *secure with respect to P and $\cong$*, written $secure_{P,\cong}(r, i \vdash_u \phi)$, iff for all $r' \in traces(L)$ such that $r^i \cong r'$, it holds that $[\phi]^{db} = [\phi]^{db'}$, where $last(r^i) = \langle db, U, sec, T, V, ctx\rangle$ and $last(r') = \langle db', U', sec', T', V', ctx'\rangle$. □

We are now ready to define data confidentiality. Intuitively, an access control mechanism provides data confidentiality iff all judgments that an attacker can derive are secure.

Definition 6.4. Let $P = \langle M, f\rangle$ be an extended configuration, L be the P-LTS, $u \in \mathcal{U}$ be a user, A be a (P, u)-attacker model, and $\cong$ be a P-indistinguishability relation. We say that f *provides data confidentiality with respect to P, u, A, and $\cong$* iff $secure_{P,\cong}(r, i \vdash_u \phi)$ for all judgments $r, i \vdash_u \phi$ that hold in A. □

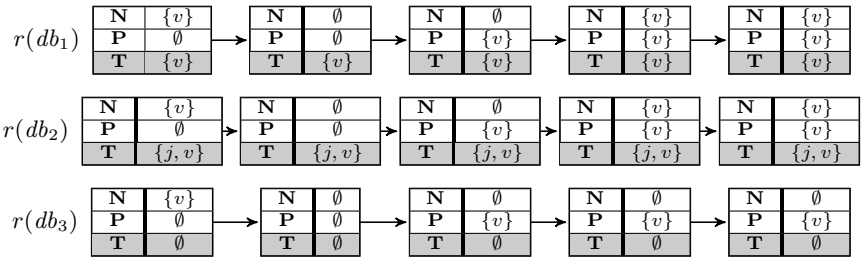


FIGURE 6.6: The runs $r(db_1)$ and $r(db_2)$ are indistinguishable, whereas $r(db_1)$ and $r(db_3)$ are not.

6.4.2 Indistinguishability

We now define the indistinguishability relation that we use in the rest of the chapter. This relation captures what each user can observe (as specified in Section 6.2) and the effects of the system's security policy. Let $P = \langle\langle D, \Gamma\rangle, f\rangle$ be an extended configuration, L be the P-LTS, and u be a user in $\mathcal{U}$. Given a run $r \in traces(L)$, the user u is aware only of his actions and not of the actions of the other users in r. This is represented by the u-projection of r, which is obtained by masking all sequences of actions that are not issued by u using a distinguished symbol $*$. Specifically, the *u-projection of r* is an alternating sequence of states in Ω_M and actions in $\mathcal{A}_{D,u} \cup \mathcal{TRIGGER}_D \cup \{*\}$ that is obtained from r by (1) replacing each action not issued by u with $*$, (2) replacing each trigger whose invoker is not u with $*$, and (3) replacing all non-empty sequences of $*$-transitions with a single $*$-transition. For each user $u \in \mathcal{U}$, we define the P-indistinguishability relation $\cong_{P,u}$. Informally, two runs r and r' are $\cong_{P,u}$-indistinguishable, denoted $r \cong_{P,u} r'$, iff (1) the labels of the u-projections of r and r' are the same, (2) u executes the same actions $a_1, \ldots, a_n$ in r and r', in the same order, and with the same results, and (3) before each action a_i, where $1 \leq i \leq n$, as well as in the last states of r and r', the policy, the views, the triggers, the users, and the data disclosed by the policy are the same in r and r'.

We remark that there is a close relation between $\cong_{P,u}$ and existing indistinguishability relations over database states, such as those in [131, 165] or the one from Section 3.4. For any two $\cong_{P,u}$-indistinguishable runs r and r', the database states that precede all actions issued by u as well as the last states in r and r' are pairwise indistinguishable with respect to existing state-based notions [131, 165].

Example 6.3 illustrates our indistinguishability notion.

Example 6.3. Let the schema, the set of users, the policy, and the triggers be as in Example 6.1. Consider the following run $r(db)$, parametrized by the initial database state db:

1. u deletes v from N.
2. u inserts v in P. If v is in T, this activates the trigger t, which, in turn, inserts v in N.
3. u issues the `SELECT` query $N(v)$.

Let db_1, db_2, and db_3 be three database states such that $db_1(T) = \{v\}$, $db_2(T) = \{j, v\}$, and $db_3(T) = \emptyset$, whereas $db_i(N) = \{v\}$ and $db_i(P) = \emptyset$, for $1 \leq i \leq 3$. Note that $r(db_1)$ is the run used in Example 6.1. Figure 6.6 depicts how the database's state changes during the runs $r(db_i)$, for $1 \leq i \leq 3$. Gray indicates those tables that the user u cannot read. The runs $r(db_1)$ and $r(db_2)$ are indistinguishable for the user u. The only difference between them is the content of the table T, which u cannot read. In contrast, u can distinguish between $r(db_1)$ and $r(db_3)$ because the trigger has been executed in the former and not in the latter.

Indistinguishability may also depend on the actions of the other users. Consider the runs r' and r'' obtained by extending $r(db_1)$ respectively with one and two `SELECT` queries issued by the administrator just after u's query. The user u can distinguish between $r(db_1)$ and r' because he knows that other users interacted with the system in r' but not in $r(db_1)$, i.e., the u-projections have different labels. In contrast, the runs r' and r'' are indistinguishable for u because he only knows that, after his own `SELECT` query, other users interacted with the system, i.e., the u-projections have the same labels. However, he does not know the number of commands, the commands themselves, or their results. ■

We are now ready to formalize the indistinguishability relation $\cong_{P,u}$. We first formalize the notion of u-projection. Afterwards, we define the notion of consistency between u-projections. Finally, we formalize the indistinguishability relation $\cong_{P,u}$.

Projections

Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ and f is an M-PDP, L be the P-LTS, and u be a user in $\mathcal{U}$. As already mentioned, the u-projection of a run r is obtained by (1) replacing each action not issued by u with $*$, (2) replacing each trigger whose invoker is not u with $*$, and (3) replacing all non-empty sequences of $*$-transitions with a single $*$-transition. Note that the $*$-transitions in the u-projections represent whether u's actions are executed consecutively or not. With a slight abuse of notation, we extend all the notation we use for runs also to u-projections. For instance, $r|^i_u$ denotes the prefix obtained by truncating $r|_u$ at its i-th state. Formally, the u-projection $r|_u$ is defined as $c(v(r,u))$. The function v takes as input a run r and a user u and returns another run in which all actions issued by users other than u are replaced with $*$.

$$v(r,u) = \begin{cases} v(r',u)\cdot a\cdot s & \text{if } r = r'\cdot a\cdot s \wedge r' \in \mathit{traces}(L) \wedge s \in \Omega_M \wedge a \in \mathcal{A}_{D,u} \wedge |r| > 1 \\ v(r',u)\cdot *\cdot s & \text{if } r = r'\cdot a\cdot s \wedge r' \in \mathit{traces}(L) \wedge s \in \Omega_M \wedge a \in \mathcal{A}_{D,u'} \wedge u' \neq u \wedge |r| > 1 \\ v(r',u)\cdot t\cdot s & \text{if } r = r'\cdot t\cdot s \wedge r' \in \mathit{traces}(L) \wedge s \in \Omega_M \wedge t \in \mathcal{TRIGGER}_D \wedge \\ & \quad \mathit{invoker}(\mathit{last}(r')) = u \wedge |r| > 1 \\ v(r',u)\cdot *\cdot s & \text{if } r = r'\cdot t\cdot s \wedge r' \in \mathit{traces}(L) \wedge s \in \Omega_M \wedge t \in \mathcal{TRIGGER}_D \wedge \\ & \quad \mathit{invoker}(\mathit{last}(r')) \neq u \wedge |r| > 1 \\ s & \text{if } r = s \text{ and } s \in \Omega_M \end{cases}$$

The function c takes as input a run r containing $*$-transitions and replaces each sequence of $*$-transitions with a single $*$-transition. Note that the function c is obtained by repeatedly applying the function c' until the computation reaches a fixed point. The function c' is as follows:

$$c'(r) = \begin{cases} c'(r')\cdot a\cdot s & \text{if } r = r'\cdot a\cdot s \wedge a \neq * \wedge r' \in \mathit{traces}(L) \wedge s \in \Omega_M \wedge |r| > 1 \\ c'(r')\cdot *\cdot s & \text{if } r = r'\cdot *\cdot s'\cdot *\cdot s \wedge r' \in \mathit{traces}(L) \wedge s, s' \in \Omega_M \wedge |r| > 2 \\ r & \text{if } r = s \wedge s \in \Omega_M \\ r & \text{if } r = s\cdot *\cdot s' \wedge s, s' \in \Omega_M \end{cases}$$

Consistency

We now define the notion of consistency between two u-projections. Observe that our consistency definition uses the function *labels* that takes as input a run r and returns as output the sequence of labels in the run, namely $\mathit{labels}(r)$ is obtained from r by dropping all the states.

Definition 6.5. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ and f is an M-PDP, L be the P-LTS, and u be a user in $\mathcal{U}$. Furthermore, let $r|_u$ and $r'|_u$ be the u-projections of the runs r and r' in $\mathit{traces}(L)$. We say that $r|_u$ *and* $r'|_u$ *are consistent* iff the following conditions hold:

1. $|r|_u| = |r'|_u|$.
2. $\mathit{labels}(r|_u) = \mathit{labels}(r'|_u)$.
3. $\mathit{triggers}(\mathit{last}(r|_u)) = \epsilon$ iff $\mathit{triggers}(\mathit{last}(r'|_u)) = \epsilon$.
4. for all i such that $1 \leq i \leq |r|_u|$, if $r|^i_u = r|^{i-1}_u\cdot a\cdot s$ and $a \neq *$, the following conditions hold:
   - $\mathit{res}(\mathit{last}(r|^i_u)) = \mathit{res}(\mathit{last}(r'|^i_u))$,
   - $\mathit{secEx}(\mathit{last}(r|^i_u)) = \mathit{secEx}(\mathit{last}(r'|^i_u))$,
   - if a is a trigger, then $\mathit{acC}(\mathit{last}(r|^i_u)) = \mathit{acC}(\mathit{last}(r'|^i_u))$,
   - $\mathit{invoker}(\mathit{last}(r|^i_u)) = \mathit{invoker}(\mathit{last}(r'|^i_u))$,
   - $\mathit{triggers}(\mathit{last}(r|^i_u)) = \mathit{triggers}(\mathit{last}(r'|^i_u))$,
   - $\mathit{tpl}(\mathit{last}(r|^i_u)) = \mathit{tpl}(\mathit{last}(r'|^i_u))$, and
   - $\mathit{Ex}(\mathit{last}(r|^i_u)) = \mathit{Ex}(\mathit{last}(r'|^i_u))$. □

Indistinguishability

Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, \mathit{sec}, T, V\rangle$ be an M-system state, and $u \in U$ be a user. The set $\mathit{permissions}(s,u)$ is $\mathit{permissions}(s,u) := \{\langle \oplus, \texttt{SELECT}, O\rangle \mid \exists u' \in U, \mathit{op} \in \{\oplus, \oplus^*\}.\, \langle \mathit{op}, u, \langle \texttt{SELECT}, O\rangle, u'\rangle \in \mathit{sec}\}$. We also fix $\mathit{permissions}(s, \mathit{admin})$ to be $D \cup V$ since the administrator has read access to the whole database. We extend permissions to M-states as follows. Given an M-state $s' = \langle db, U, \mathit{sec}, T, V, c\rangle$, $\mathit{permissions}(s', u) = \mathit{permissions}(\langle db, U, \mathit{sec}, T, V\rangle, u)$.

We are now ready to introduce the notion of indistinguishability between two runs. Intuitively, two runs r and r' are indistinguishable for a user u iff (1) their u-projections are consistent, and (2) for each action of the user u as well as for the last states in the two runs, the policy, the triggers, the views, the users, and the data disclosed by the policy are the same in r and r'.

Definition 6.6. Let $P = \langle M, f\rangle$ be an extended configuration, L be the P-LTS, and u be a user.

We say that two runs r and r' in $\mathit{traces}(L)$ are (P,u)-*indistinguishable*, written $r \cong_{P,u} r'$, iff

1. $r|_u$ and $r'|_u$ are consistent,
2. $\mathit{sysState}(\mathit{last}(r))$ and $\mathit{sysState}(\mathit{last}(r'))$ are (M,u)-data indistinguishable, and

3. for all i such that $1 \leq i \leq |r|_u|-1$, if $r|_u^{i+1} = r|_u^i \cdot a \cdot s$, $a \neq *$, and $s \in \Omega_M$, then $sysState(last(r|_u^i))$ and $sysState(last(r'|_u^i))$ are (M,u)-data indistinguishable.

We say that two M-system states $s = \langle db, U, sec, T, V\rangle$ and $s' = \langle db', U', sec', T', V'\rangle$ are (M,u)-*data indistinguishable*, written $s \cong_{M,u}^{data} s'$, iff (a) $U = U'$, (b) $sec = sec'$, (c) $T = T'$, (d) $V = V'$, (e) for all relation schema $R \in D$ for which $\langle \oplus, \texttt{SELECT}, R\rangle \in permissions(s,u)$, $db(R) = db'(R)$, and (f) for all views $v \in \mathcal{VIEW}_D^{owner}$ for which $\langle \oplus, \texttt{SELECT}, v\rangle \in permissions(s,u)$, $db(v) = db'(v)$. □

Given a run r, we denote by $[\![r]\!]_{P,u}$ the equivalence class of r defined by $\cong_{P,u}$ over $traces(L)$. Similarly, we denote by $[\![s]\!]_{M,u}^{data}$ the equivalence class of s defined by $\cong_{M,u}^{data}$ over Π_M.

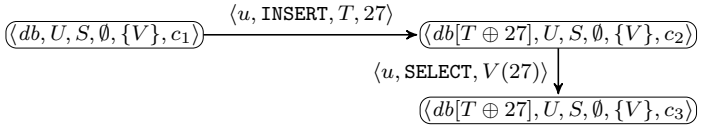
6.4.3 Examples

Example 6.4 shows that existing PDPs leak sensitive information and therefore do not provide data confidentiality.

Example 6.4. In Example 6.1, we showed how the user u derives $r, 3 \vdash_u T(v)$. The judgment is not secure because there is a run indistinguishable from r^3, i.e., the run $r^3(db_3)$ in Example 6.3, in which $T(v)$ does not hold. ■

Example 6.5 shows how views may leak information about the underlying tables. Even though this leakage might be considered legitimate, there is no way in our setting to distinguish between intended and unintended leakages. If this is desired, data confidentiality can be extended with the concept of *declassification* [20, 21].

Example 6.5. Consider a database with two tables T and Z and a view $V = \langle v, admin, \{x \mid T(x) \wedge Z(x)\}, O\rangle$. The set U is $\{u, admin\}$ and the policy S is $\{\langle \oplus, u, \langle \texttt{SELECT}, T\rangle, admin\rangle, \langle \oplus, u, \langle \texttt{SELECT}, V\rangle, admin\rangle, \langle \oplus, u, \langle \texttt{INSERT}, T\rangle, admin\rangle\}$. Consider the following run r, parametrized by the initial database state db, where u first inserts 27 into T and afterwards issues the `SELECT` query $V(27)$. We assume there are no exceptions in r.



We used Maude to generate the runs $r(d)$ and $r(d')$ with the initial database states d and d' such that $d(T) = d(Z) = d'(T) = \emptyset$ and $d'(Z) = \{27\}$. The runs $r^1(d)$ and $r^1(d')$ are indistinguishable for u because they differ only in the content of Z, which u cannot read. After the `INSERT`, u can distinguish between $r^2(d)$ and $r^2(d')$ by reading V. Indeed, $d[T \oplus 27](V) = \emptyset$, because $d(Z) = \emptyset$, whereas $d'[T \oplus 27](V) = \{27\}$. The user u derives $r(d'), 1 \vdash_u Z(27)$, which is not secure because $r^1(d)$ and $r^1(d')$ are indistinguishable for u, but $Z(27)$ holds just in the latter. ■

In contrast to existing security notions [131, 165], we have defined data confidentiality over runs. This is essential to model and detect attacks, such as those in Examples 6.4 and 6.5, where an attacker infers sensitive information from the transitions between states. For instance, the leakage in Example 6.5 is due to the execution of the `INSERT` command. Although the `SELECT` command is authorized by the policy, u can use it to infer sensitive information about the system's state before the `INSERT` execution.

6.5 A Provably Secure PDP

We now present a PDP that provides both database integrity and data confidentiality. We first explain the ideas behind it using examples. Afterwards, we show that it satisfies the desired security properties and has acceptable overhead. Finally, we argue that it is more permissive than existing access control solutions for commands that do not violate our security properties.

Figure 6.7 depicts our PDP f together with the functions f_{int} and f_{conf}. Additional details about the PDP are given in Section 6.8. The PDP takes as input a state s and an action a and outputs $\top$ iff both f_{int} and f_{conf} authorize a in s, i.e., iff a's execution neither violates database integrity nor data confidentiality. Note that our algorithm is not *complete* in that it may reject some secure commands. However, from the impossibility results in Chapter 3, it follows that no algorithm can be complete and provide database integrity and data confidentiality for the relational calculus.

Our PDP is invoked by the database system each time a user u issues an action a to check whether u is authorized to execute a. The PDP is also invoked whenever the database system executes a scheduled trigger t: once to check if the `SELECT` statement associated with t's `WHEN` condition is authorized and once, in case t is enabled, to check if t's action is authorized.

▷ *s is a state and a is an action*
function $f(s, a)$
1. **return** $f_{int}(s, a) \wedge f_{conf}(s, a, user(s, a))$

▷ *s is a state and a is an action*
function $f_{int}(s, a)$
1. **if** $trg(s) = \epsilon$ **return** $auth(s, a)$
2. **else if** $a = cond(trg(s), s)$ **return** $\top$
3. **else if** $a = act(trg(s), s)$
4. **return** $auth(s, trg(s))$
5. **else return** $\bot$

▷ *s is a state, a is an action, and u is a user*
function $f_{conf}(s, a, u)$
1. **switch** a
2. **case** $\langle u', \texttt{SELECT}, q\rangle$: **return** $secure(u, q, s)$
3. **case** $\langle u', \texttt{INSERT}, R, \bar{t}\rangle$: **case** $\langle u', \texttt{DELETE}, R, \bar{t}\rangle$:
4. **if** $leak(a, s, u) \vee \neg secure(u, getInfo(a), s)$
5. **return** $\bot$
6. **for** $\gamma \in Dep(a, \Gamma)$
7. **if** $(\neg secure(u, getInfoS(\gamma, a), s) \vee$ $\neg secure(u, getInfoV(\gamma, a), s))$
8. **return** $\bot$
9. **case** $\langle \oplus, u'', pr, u'\rangle, \langle \oplus^*, u'', pr, u'\rangle$:
10. **return** $\neg leak(a, s, u)$
11. **return** $\top$

FIGURE 6.7: The PDP f uses the two subroutines f_{int} and f_{conf}. The former provides database integrity and the latter provides data confidentiality with respect to the user $user(s, a)$, which denotes either the user issuing the action, when the system is not executing a trigger, or the trigger's invoker.

6.5.1 Enforcing Database Integrity

The function f_{int} takes as input a state s and an action a. If the system is not executing a trigger, denoted by $trg(s) = \epsilon$, f_{int} checks (line 1) whether a is authorized with respect to s. In line 2, f_{int} checks whether a is the current trigger's condition. If this is the case, it returns $\top$ because the triggers' conditions do not violate database integrity. Finally, the algorithm checks (line 3) whether a is the current trigger's action, and if this is the case, it checks whether the current trigger $trg(s)$ is authorized with respect to s (line 4). The function *auth* uses a sound and computable under-approximation of $\rightsquigarrow_{auth}$ to check if a is authorized with respect to s. Thus, any action authorized by f_{int} is authorized according to $\rightsquigarrow_{auth}$. This ensures database integrity. Note that $\rightsquigarrow_{auth}$ relies on query determinacy (see Chapter 2) to decide whether a query is determined by a set of views. Since determinacy is undecidable [124], in *auth* we implement a sound under-approximation of it, given in Section 6.8.2. Our approximation checks syntactically if a query is determined by a set of views.

Example 6.6. Consider a database with three tables: R, T, and Z. The set U is $\{u, u', admin\}$ and the policy S is $\{\langle \oplus, u, \langle \texttt{SELECT}, R\rangle, admin\rangle, \langle \oplus^*, u, \langle \texttt{SELECT}, T\rangle, admin\rangle, \langle \oplus^*, u, \langle \texttt{SELECT}, Z\rangle, admin\rangle\}$. There are two views $V = \langle v, admin, \{x \mid T(x) \wedge Z(x)\}, O\rangle$ and $W = \langle w, u, \{x \mid R(x) \vee V(x)\}, O\rangle$. The user u tries to grant to u' read access to W, i.e., he issues $\langle \oplus, u', \langle \texttt{SELECT}, W\rangle, u\rangle$. The PDP f_{int} rejects the command and raises a security exception because u is authorized to delegate the read access only for T and Z but W's result depends also on R, for which u cannot delegate read access. Assume now that the policy is $\{\langle \oplus^*, u, \langle \texttt{SELECT}, R\rangle, admin\rangle, \langle \oplus^*, u, \langle \texttt{SELECT}, T\rangle, admin\rangle, \langle \oplus^*, u, \langle \texttt{SELECT}, Z\rangle, admin\rangle\}$. In this case, f_{int} authorizes the `GRANT`. The reason is that W's definition can be equivalently rewritten as $\{x \mid R(x) \vee (T(x) \wedge Z(x))\}$ and u is authorized to delegate the read access for R, T, and Z. ■

6.5.2 Enforcing Data Confidentiality

The function f_{conf}, shown in Figure 6.7, takes as input an action a, a state s, and a user u. Note that any user other than the administrator is a potential attacker. The requirement for f_{conf} is that it authorizes only those commands that result in secure judgments for u as required by Definition 6.4. To achieve this, f_{conf} over-approximates the set of judgments that u can derive from a's execution. For instance, the algorithm assumes that u can always derive the trigger's condition from the run, even though this is not always the case. Then, f_{conf} authorizes a iff it can determine that all u's judgements are secure. This can be done by analysing just a finite subset of the over-approximated set of u's judgments.

In more detail, f_{conf} performs a case distinction on the action a (line 1). If a is a `SELECT` command (line 2), f_{conf} checks whether the query is secure with respect to the current state s and the user u using the *secure* procedure. If a is an `INSERT` or `DELETE` command (lines 3–8), f_{conf} checks (lines 4–5), using the *leak* procedure, whether a's execution may leak sensitive information through the views that u can read, as in Example 6.5. Afterwards, f_{conf} also checks (lines 4–5) whether the information u can learn from a's execution, modelled by the sentence computed by the procedure $getInfo(a)$, is secure. In line 6–8, f_{conf} computes the set of all integrity constraints that a's execution may violate, denoted by $Dep(a, \Gamma)$, and for all constraints γ, it checks whether the information that u may learn from γ is secure. The procedure $getInfoS$ (respectively $getInfoV$) computes the sentence

modelling the information learned by u from γ if a is executed successfully (respectively violates γ). If a is a `GRANT` command (lines 9–10), f_{conf} checks whether a's successful execution discloses sensitive information to u. In the remaining cases (line 11), f_{conf} authorizes a.

Secure judgments. Determining if a given judgment is secure is undecidable for RC (this directly follows from our results in Chapter 3). Hence, the *secure* procedure implements a sound and computable under-approximation of this notion. We now present our solution. Other sound under-approximations can alternatively be used without affecting f_{conf}'s data confidentiality guarantees.

Let $M = \langle D, \Gamma \rangle$ be a system configuration, $r, i \vdash_u \phi$ be a judgment, and $s = \langle db, U, sec, T, V, c \rangle$ be the i-th state in r. As a first under-approximation, instead of the set of all runs indistinguishable from r^i, we consider the larger set of all runs r' whose last state $s' = \langle db', U, sec, T, V, c' \rangle$ is such that the disclosed data in db and db' are the same. Note that if a judgment is secure with respect to this larger set, it is secure also with respect to the set of indistinguishable runs because the former set contains the latter. This larger set depends just on the database state db and the policy sec, not on the run or the attacker model $\mathcal{ATK}_u$. Determining judgment's security is, however, still undecidable even on this larger set. We therefore employ a second under-approximation that uses query rewriting. We rewrite the sentence ϕ to a sentence ϕ_{rw} such that if $r, i \vdash_u \phi$ is not secure for the user u, then $[\phi_{rw}]^{db} = \top$. The formula ϕ_{rw} is $\neg\phi^{\top}_{s,u} \wedge \phi^{\bot}_{s,u}$, where $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ are defined inductively over ϕ. A formal definition of *secure* is given in Section 6.8.3.

We now explain how we construct $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$. We assume that both ϕ and V contain only views with the owner's privileges. The extension to the general case is straightforward. First, for each table or view $o \in D \cup V$, we create additional views representing any possible projection of o. The *extended vocabulary* contains the tables in D, the views in V, and their projections. For instance, given a table $R(x, y)$, we create the views R_x and R_y representing respectively $\{y \mid \exists x.\, R(x, y)\}$ and $\{x \mid \exists y.\, R(x, y)\}$. Second, we compute the formula ϕ' by replacing each sub-formula $\exists \overline{x}.\, R(\overline{x}, \overline{y})$ in ϕ with the view $R_{\overline{x}}(\overline{y})$ associated with the corresponding projection. Third, for each predicate symbol R in the formula ϕ', we compute the sets $R^{\top}_{s,u}$ and $R^{\bot}_{s,u}$. The set $R^{\top}_{s,u}$ (respectively $R^{\bot}_{s,u}$) contains all the tables and views K in the extended vocabulary such that (1) K is contained in (respectively contains) R, and (2) the user u is authorized to read K in s, i.e., there is a grant $\langle op, u, \langle$`SELECT`, $K' \rangle, u' \rangle \in sec$ such that either $K' = K$ or K is obtained from K' through a projection. The formula $\phi^{v}_{s,u}$, where $v \in \{\top, \bot\}$, is defined as follows:

$$\phi^{v}_{s,u} = \begin{cases} \bigvee_{S \in R^{\top}_{s,u}} S(\overline{x}) & \text{if } \phi = R(\overline{x}) \text{ and } v = \top \\ \bigwedge_{S \in R^{\bot}_{s,u}} S(\overline{x}) & \text{if } \phi = R(\overline{x}) \text{ and } v = \bot \\ \neg\psi^{\neg v}_{s,u} & \text{if } \phi = \neg\psi \\ \psi^{v}_{s,u} * \gamma^{v}_{s,u} & \text{if } \phi = \psi * \gamma \text{ and } * \in \{\vee, \wedge\} \\ Q\, x.\, \psi^{v}_{s,u} & \text{if } \phi = Q\, x.\, \psi \text{ and } Q \in \{\exists, \forall\} \\ \phi & \text{otherwise} \end{cases}$$

The formulae are such that if $\phi^{\top}_{s,u}$ holds, then ϕ holds and if $\neg\phi^{\bot}_{s,u}$ holds, then $\neg\phi$ holds. To compute the sets $R^{\top}_{s,u}$ and $R^{\bot}_{s,u}$, we check the containment between queries. Since query containment is undecidable [10], we implement a sound under-approximation of it, described in Section 6.8.3. Other sound under-approximations can be used as well.

Our $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ rewritings share similarities with the *low* and *high evaluations* of Wang et al. [165]. Both try to approximate the result of a query just by looking at the authorized data. However, we use $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ to determine a judgment's security, whereas Wang et al. use evaluations to restrict the query's results only to authorized data.

Example 6.7. Consider a database with three tables S, R, and Q, and two views $V = \langle v, admin, \{x, y \mid S(x, y) \wedge (x = 1 \vee y = 3)\}, O \rangle$ and $W = \langle w, admin, \{x \mid R(x) \vee Q(x)\}, O \rangle$. The database state db is $db(S) = \{(1, 1), (2, 3), (4, 2)\}$, $db(R) = \{3\}$, and $db(Q) = \{4\}$, the set U is $\{u, admin\}$, and the policy sec is $\{\langle \oplus, u, \langle$`SELECT`$, V \rangle, admin \rangle, \langle \oplus, u, \langle$`SELECT`$, W \rangle, admin \rangle\}$. Let the state s be $\langle db, U, sec, \emptyset, \{V, W\}, \epsilon \rangle$ and the run r be s. We want to check the security of $r, 1 \vdash_u \phi$, where $\phi := (\exists y.\, S(2, y)) \wedge (\neg R(5) \vee \exists y.\, S(4, y))$, for the user u. Figure 6.8 depicts the database state db, the materializations of the views V and W, and the materializations of the views S_x, S_y, V_x, and V_y in the extended vocabulary. Gray indicates those tables and views that u cannot read.

The rewriting process, depicted in Figure 6.8, proceeds as follows. We first rewrite the formula ϕ as $S_y(2) \wedge (\neg R(5) \vee S_y(4))$. The sets ${S_y}^{\top}_{s,u}$, ${S_y}^{\bot}_{s,u}$, $R^{\top}_{s,u}$, and $R^{\bot}_{s,u}$ are respectively $\{V_y\}$, $\emptyset$, $\emptyset$, and $\{W\}$. The formulae $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ are respectively $S_y(2)^{\top}_{s,u} \wedge (\neg R(5)^{\bot}_{s,u} \vee S_y(4)^{\top}_{s,u})$, which is equivalent to $V_y(2) \wedge (\neg W(5) \vee V_y(4))$, and $S_y(2)^{\bot}_{s,u} \wedge (\neg R(5)^{\top}_{s,u} \vee S_y(4)^{\bot}_{s,u})$, which is equivalent to $\top$. They are both secure, as they depend only on V and W. Furthermore, since $\phi^{\top}_{s,u}$ holds in s, then ϕ holds as well. Finally, ϕ_{rw} is $\neg\phi^{\top}_{s,u} \wedge \phi^{\bot}_{s,u}$. Since ϕ_{rw} does not hold in s, it follows that $r, 1 \vdash_u \phi$ is secure. ■

Database State

S	
1	1
2	3
4	2

R
3

Q
4

Views

V	
1	1
2	3

W
3
4

$V = \{x, y \mid S(x, y) \wedge (x = 1 \vee y = 3)\}$

$W = \{x \mid R(x) \vee Q(x)\}$

Extended Vocabulary

$\mathbf{S}_x$
1
3
2

$\mathbf{S}_y$
1
2
4

$\mathbf{V}_x$
1
3

$\mathbf{V}_y$
1
2

Containment Sets

${S_y}^{\top}_{s,u} = \{V_y\}$ $\quad R^{\top}_{s,u} = \emptyset$

${S_y}^{\bot}_{s,u} = \emptyset$ $\quad R^{\bot}_{s,u} = \{W\}$

$S_x = \{y \mid \exists x.\, S(x, y)\}$ $\quad V_x = \{y \mid \exists x.\, V(x, y)\}$

$S_y = \{x \mid \exists y.\, S(x, y)\}$ $\quad V_y = \{x \mid \exists y.\, V(x, y)\}$

Original Sentence

$\phi := (\exists y.\, S(2, y)) \wedge (\neg R(5) \vee \exists y.\, S(4, y)) \equiv S_y(2) \wedge (\neg R(5) \vee S_y(4))$

Rewriting

$\phi_{rw} := \neg \phi^{\top}_{s,u} \wedge \phi^{\bot}_{s,u}$

$\phi^{\top}_{s,u} := {S_y(2)}^{\top}_{s,u} \wedge (\neg {R(5)}^{\bot}_{s,u} \vee {S_y(4)}^{\top}_{s,u}) \equiv V_y(2) \wedge (\neg W(5) \vee V_y(4))$

$\phi^{\bot}_{s,u} := {S_y(2)}^{\bot}_{s,u} \wedge (\neg {R(5)}^{\top}_{s,u} \vee {S_y(4)}^{\bot}_{s,u}) \equiv \top$

FIGURE 6.8: Checking the security of the judgment $r, 1 \vdash_u (\exists y.\, S(2, y)) \wedge (\neg R(5) \vee \exists y.\, S(4, y))$ from Example 6.7.

6.5.3 Theoretical Evaluation

Our PDP provides the desired security guarantees and its data complexity, i.e., the complexity of executing f when the action, the policy, the triggers, and the views are fixed, is AC^0. This means that f can be evaluated in logarithmic space in the database's size, as $AC^0 \subseteq LOGSPACE$, and evaluation is highly parallelizable. Note that *secure*'s data complexity is AC^0 because it relies on query evaluation, whose data complexity is AC^0 [10]. In contrast, all other operations in f are executed in constant time in terms of data complexity. Note also that on a single processor, f's data complexity is polynomial in the database's size. We believe that this is acceptable because the database is usually very large, whereas the query, which determines the degree of the polynomial, is small. The proof of Theorem 6.1 is given in Appendix C.

Theorem 6.1. *Let $P = \langle M, f \rangle$ be an extended configuration, where M is a system configuration and f is as above. The PDP f (1) provides data confidentiality with respect to P, u, $\mathcal{ATK}_u$, and $\cong_{P,u}$, for any user $u \in \mathcal{U}$, and (2) provides database integrity with respect to P. Moreover, the data complexity of f is AC^0.*

As the Examples 6.8 and 6.9 below show, f is more permissive than existing PDPs for those actions that violate neither database integrity nor data confidentiality.

Example 6.8. Our PDP is more permissive than existing mechanisms for commands of the form `GRANT SELECT ON` V `TO` u, where V is a view with owner's privileges, u is a user, and the statement is issued by the view's owner o. Such mechanisms, in general, authorize the `GRANT` iff o is authorized to delegate the read permission for all tables and views that occur in v's definition. Consider again Example 6.6. Our PDP authorizes $\langle \oplus, u', \langle \texttt{SELECT}, W \rangle, u \rangle$ under the policy S'. However, existing mechanisms reject it because u is not directly authorized to read V, although u can read the underlying tables. Our PDP also authorizes all the secure `GRANT` statements authorized by existing mechanisms. ■

Example 6.9. Our PDP is more permissive than the mechanisms used in existing DBMSs for secure `SELECT` statements. Such mechanisms, in general, authorize a `SELECT` statement issued by a user u iff u is authorized to read all tables and views used in the query. They will reject the query in Example 6.7 even though the query is secure. Furthermore, any secure `SELECT` statement authorized by them will be authorized by our solution as well. Also the PDP proposed by Rizvi et al. [131] rejects the query in Example 6.7 as insecure. However, our solution and the proposal of Rizvi et al. [131] are incomparable in terms of permissiveness, i.e., some secure `SELECT` queries are authorized by one mechanism and not by the other. ■

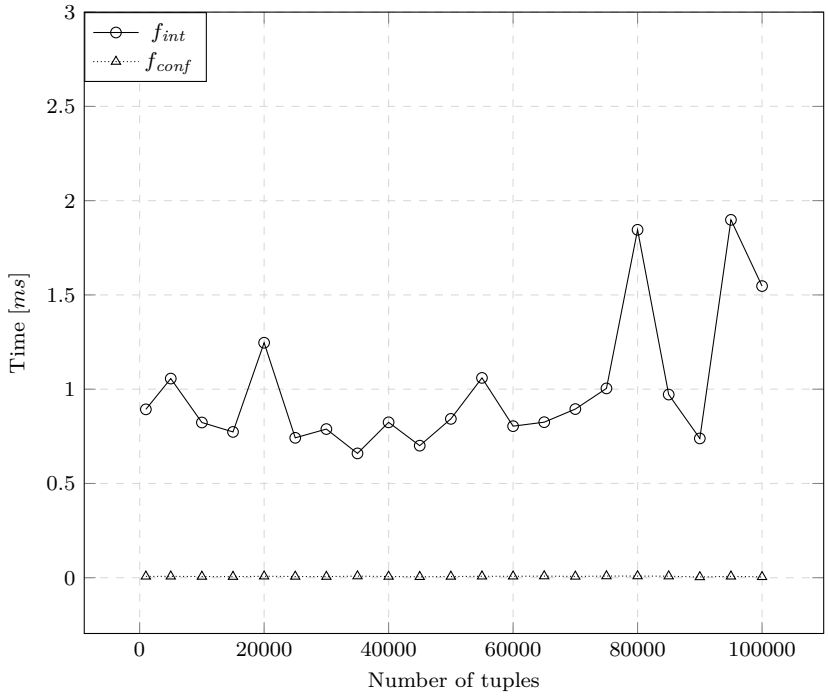


FIGURE 6.9: Example 6.6 – PDP execution time

6.5.4 Implementation

To evaluate the feasibility and security of our approach in practice, we implemented our PDP in Java. The prototype, available at [86], implements both our PDP and the operational semantics of our system model. It relies on the underlying PostgreSQL database for executing the `SELECT`, `INSERT`, and `DELETE` commands. Note that our prototype also handles all the differences between the relational calculus and SQL. For instance, it translates every relational calculus query into an equivalent `SELECT` SQL query over the underlying database. We performed a preliminary experimental evaluation of our prototype. Our experiments were run on a PC with an Intel i7 processor and 32GB of RAM. Note that we materialized the content of all the views.

Our evaluation has two objectives: (1) to empirically validate that the prototype provides the desired security guarantees, and (2) to evaluate its overhead. For (1), we ran the attacks in Section 6.1 against our prototype. As expected, our PDP prevents all the attacks. For (2), we simulated Examples 6.6 and 6.7 on database states where the number of tuples ranges from 1,000 to 100,000. Figures 6.9 and 6.10 shows the PDP's execution time for the Examples 6.6 and 6.7 respectively. Our results show that our solution is feasible. In more detail, f_{int}'s execution time does not depend on the database size, whereas f_{conf}'s execution time does. We believe that the overhead introduced by the PDP is acceptable for a proof of concept. Even with 100,000 tuples, the PDP's running time is under a second. In Example 6.7, f_{conf}'s execution time is comparable to the execution time of the user's query. As Figure 6.11 shows, f_{conf}'s query rewriting time does not depend on the database's size, whereas f_{conf}'s query execution time does.

To improve f_{conf}'s performance, one could strike a different balance between simple syntactic checks and our query rewriting solution. This, however, would result in more restrictive PDPs.

6.6 Related Work

We compare our work against two lines of research: database access control and information-flow control. Both of these have similar goals, namely preventing the leakage and corruption of sensitive information.

6.6.1 Database Access Control

Attacker Models for Database Access Control. Surprisingly, and in contrast to other areas of information security [66], there does not exist a well-defined attacker model for database access control. From the literature, we extracted the `SELECT`-only attacker model, where the attacker uses

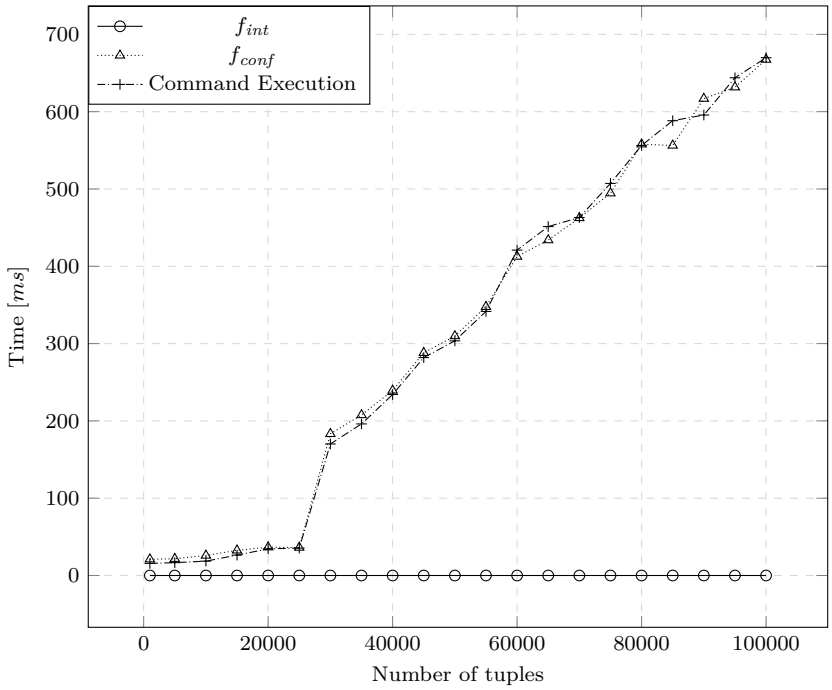


FIGURE 6.10: Example 6.7 – PDP execution time

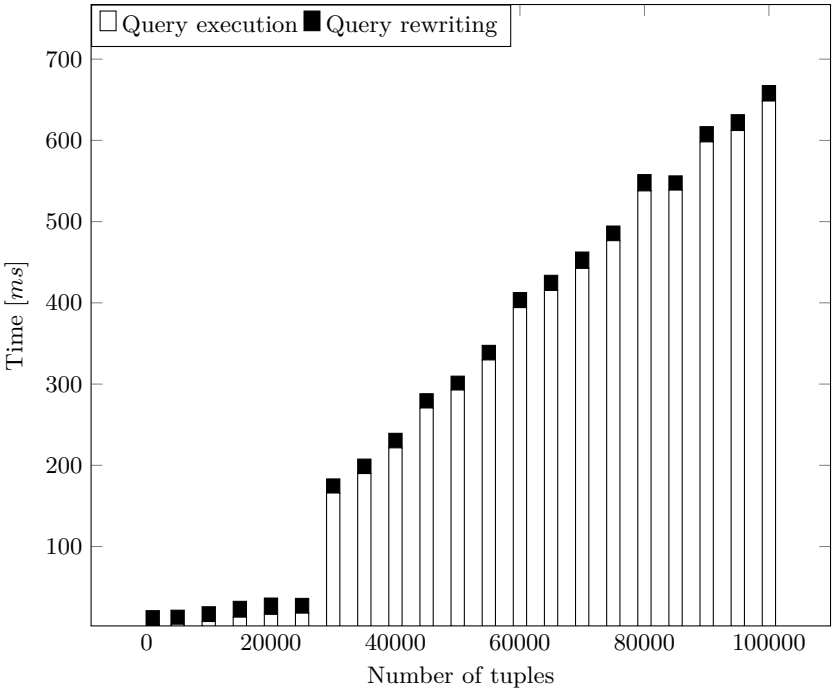


FIGURE 6.11: Example 6.7 – f_{conf}'s execution time.

just `SELECT` commands. A number of access control mechanisms, such as [8, 13, 26, 27, 41, 90, 110, 131, 145, 150, 165], implicitly consider this attacker model. The boundaries of this model are blurred and the attacker's capabilities are unclear. For instance, only a few works, such as [165], explicitly state that update commands are not supported, whereas others [13, 26, 27, 131] ignore what the attacker can learn from update commands. Works on Inference Control [40, 72, 158] and Controlled Query Evaluation [36] consider a variation of the `SELECT`-only attacker, in which the attacker additionally has some initial knowledge about the data and can derive new information from the query's results through inference rules. Note that while [158] supports update commands, it treats them just as a way of increasing data availability, rather than considering them as a possible attack vector.

Discretionary Database Access Control. Our framework builds on prior research in database access control [131,165] as well as established notions from database theory, such as determinacy [124] and instance-based determinacy [107].

Specifically, our notion of secure judgments extends instance-based determinacy from database states to runs, while data confidentiality extends existing security notions [131, 165] to dynamic settings, where both the database and the policy may change. Observe that existing security notions for the Truman [165] and Non-Truman [131] models are based on `SELECT`-only attackers and provide no security guarantees against realistic attackers that can alter the database and the policy or use advanced SQL features. The same consideration applies to the security criteria we present in Chapter 3. Furthermore, our indistinguishability notion extends the one proposed by Wang et al. [165] (as well as our indistinguishability notion from Chapter 3) from database states to runs. Finally, our formalization of $\rightsquigarrow_{auth}$ relies on determinacy to decide whether the content of a view is fully determined by a set of other views.

Griffiths and Wade propose a PDP [83] that prevents Attacks 6.2 and 6.3 by using syntactic checks and by removing all views whose owners lack the necessary permissions. In contrast, we prevent the execution of `GRANT` and `REVOKE` operations leading to inconsistent policies.

Mandatory Database Access Control. Research on mandatory database access control has historically focused on Multi-Level Security (MLS) [63], where both the data and the users are associated with security levels, which are compared to control data access. Our PDP extends the SQL discretionary access control model with additional *mandatory* checks to provide database integrity and data confidentiality. In the following, we compare our work with the access control policies and semantics used by MLS systems.

With respect to policies, our work uses the SQL access control model, where policies are sets of `GRANT` statements. In this model, users can dynamically modify policies by delegating permissions. In contrast, MLS policies are usually expressed by labelling each subject and object in the system with labels from a security lattice [138]. The policy is, in general, fixed (cf. the *tranquillity principle* [138]).

With respect to semantics, existing MLS solutions are based on the Truman model [131]: they transparently modify the commands issued by the users to restrict the access to only the authorized data. In contrast, we use the same semantics as SQL, that is, we execute only the secure commands. Namely, we adopt the Non-Truman model [131]. We refer the reader to Chapter 3 for a detailed comparison of these two access control models. Operationally, MLS mechanisms use poly-instantiation [99], which is neither supported by the relational model nor by the SQL standard, and requires ad hoc extensions [63, 140]. Furthermore, the operational semantics of MLS systems differs from the standard relational semantics. In contrast, our operational semantics supports the relational model and is directly inspired by SQL.

The above differences influence how security properties are expressed. Data confidentiality, which relies on a precise characterization of security based on a possible worlds semantics, is a key component of the Non-Truman model (and SQL) access control semantics. Similarly, database integrity requires that any "write" operation is authorized according to the policy and is directly inspired by the SQL access control semantics. The security properties in MLS systems, in contrast, combine the multilevel relational semantics [63, 140] with MLS and BIBA properties [138].

MLS systems prevent attacks similar to Attacks 6.4 and 6.5 using poly-instantiated tuples and triggers [140, 149], whereas attacks similar to Attack 6.1 cannot be carried out because triggers with activator's privileges are not supported [149]. The SeaView system [63], which combines discretionary access control and MLS, additionally prevents attacks similar to Attacks 6.2 and 6.3 by relying on Griffiths and Wade's PDP [83]. However, these solutions cannot be applied to SQL databases for the aforementioned reasons.

6.6.2 Information-flow Control

Various authors have applied ideas from information-flow control to databases. Davis and Chen [59] study how cross-application information flows can be tracked through databases. Other researchers [57, 111, 141] present languages for developing secure applications that use databases. They employ secure

type systems to track information flows through databases. However, they neither model nor prevent the attacks we identified because they do not account for the advanced database features and the strong attacker model we study in this chapter.

Schultz and Liskov [143] extend decentralized information-flow control [122] to databases, based on concepts from multi-level security [63]. They identify one attack on data confidentiality that exploits integrity constraints. Their solution relies on poly-instantiation [99] and cannot be applied to SQL databases that do not support multi-level security. Their mechanism neither prevents the other attacks we identify nor provides provable and precise security guarantees.

Several researchers have studied attacker models in information-flow control [18, 78]. Giacobazzi and Mastroeni [78] model attackers as data-flow analysers that observe the program's behaviour, whereas Askarov and Chong [18] model attackers as automata that observe the program's events. They both model passive attackers, who can observe, but do not influence, the program's execution. In contrast, our attacker is active and interacts with the database.

6.7 Conclusions

Motivated by practical attacks against existing databases, we have initiated several new research directions. First, we developed the idea that attacker models should be studied and formalized for databases. Rather than being implicit, the relevant models must be made explicit, just like when analyzing security in other domains. In this respect, we presented a concrete attacker model that accounts for relevant features of modern databases, like triggers and views, and attacker inference capabilities.

Second, access control mechanisms must be designed to be secure, and provably so, with respect to the formalized attacker capabilities. This requires research on mechanism design, complemented by a formal operational semantics of databases as a basis for security proofs. We presented such a mechanism, proved that it is secure, and built and evaluated a prototype of it in PostgreSQL.

6.8 Technical Details

In Section 6.8.1, we present our full attacker model. Afterwards, we provide further details on how we enforce database integrity (Section 6.8.2) and data confidentiality (Section 6.8.3). Finally, in Section 6.8.4 we formalize our enforcement mechanism as a PDP.

6.8.1 Full Attacker Model

In this section, we formalize our attacker model $\mathcal{ATK}_u$. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, and $u \in \mathcal{U}$ be a user. The set $\mathcal{ATK}_u$ is the smallest set of judgments satisfying the inference rules in Figures 6.12–6.24. With a slight abuse of notation, in the rules we use $r, i \vdash_u \phi$ to denote that this judgment holds in $\mathcal{ATK}_u$, i.e., $r, i \vdash_u \phi \in \mathcal{ATK}_u$. Note that we redefine here also the rules we presented before in Figure 6.3.

In our attacker model, we assume that the attacker knows the PDP's implementation. Hence, he can learn information from leaks caused by the security decision. The rules in Figure 6.23 say that whenever a PDP leaks information through its security decision, then the attacker may learn any sentence that differentiates between the database states. These rules over-approximate what an attacker may infer from leaks caused by the PDP's security decision.

In the rules, we use $\models_{fin}$ to denote the standard semantic entailment relation for first-order logic over finite models. Given a view $\langle V, ow, \{\overline{x} \mid \phi\}, m\rangle \in \mathcal{VIEW}_D$ and a relational calculus formula ψ, we denote by $replace(\psi, \langle V, ow, \{\overline{x} \mid \phi\}, m\rangle)$ the formula ψ' obtained from ψ by replacing all occurrences of $V(\overline{x})$ with $\phi(\overline{x})$. Note that ψ and $replace(\psi, \langle V, ow, \{\overline{x} \mid \phi\}, m\rangle)$ are semantically equivalent. Finally, given a database schema D, a state $s = \langle db, U, sec, T, V, ctx\rangle$, and an action $a \in \mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$, we denote by $user(s, a)$ the following function:

$$user(s,a) = \begin{cases} invoker(s) & \text{if } tr(s) \neq \epsilon \\ u & \text{if } tr(s) = \epsilon \wedge u \in \mathcal{U} \wedge a \in \mathcal{A}_{D,u} \end{cases}$$

In the following, we omit some details when dealing with integrity constraints. For instance, when we refer to functional dependencies of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, (R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}'$, we implicitly assume that $|\overline{y}| = |\overline{y}'|$ and $|\overline{z}| = |\overline{z}'|$. Furthermore, when we refer to tuples in R, we use the notation $(\overline{v}, \overline{w}, \overline{q})$ to stress that a tuple can be partitioned according to $\overline{x}, \overline{y}$, and $\overline{z}$, and we implicitly assume that $|\overline{v}| = |\overline{x}|$, $|\overline{w}| = |\overline{y}|$, and $|\overline{q}| = |\overline{z}|$. We make similar simplifications for the inclusion dependencies.

PROPAGATE FORWARD SELECT
$$\frac{r, i \vdash_u \psi \qquad r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \phi \rangle \cdot s \qquad 1 \leq i < |r| \qquad s \in \Omega_M}{r, i+1 \vdash_u \psi}$$

PROPAGATE FORWARD GRANT/REVOKE
$$\frac{r, i \vdash_u \psi \qquad r^{i+1} = r^i \cdot \langle op, u', pr, u \rangle \cdot s \qquad 1 \leq i < |r| \qquad op \in \{\oplus, \oplus^*, \ominus\} \qquad s \in \Omega_M}{r, i+1 \vdash_u \psi}$$

PROPAGATE FORWARD CREATE
$$\frac{r, i \vdash_u \psi \qquad r^{i+1} = r^i \cdot \langle u, \texttt{CREATE}, o \rangle \cdot s \qquad 1 \leq i < |r| \qquad o \in \mathcal{TRIGGER}_D \cup \mathcal{VIEW}_D \qquad s \in \Omega_M}{r, i+1 \vdash_u \psi}$$

PROPAGATE BACKWARD SELECT
$$\frac{r, i+1 \vdash_u \psi \qquad r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \phi \rangle \cdot s \qquad 1 \leq i < |r| \qquad s \in \Omega_M}{r, i \vdash_u \psi}$$

PROPAGATE BACKWARD GRANT/REVOKE
$$\frac{r, i+1 \vdash_u \psi \qquad r^{i+1} = r^i \cdot \langle op, u', pr, u \rangle \cdot s \qquad 1 \leq i < |r| \qquad op \in \{\oplus, \oplus^*, \ominus\} \qquad s \in \Omega_M}{r, i \vdash_u \psi}$$

PROPAGATE BACKWARD CREATE TRIGGER
$$\frac{r, i+1 \vdash_u \psi \qquad r^{i+1} = r^i \cdot \langle u, \texttt{CREATE}, o \rangle \cdot s \qquad 1 \leq i < |r| \qquad o \in \mathcal{TRIGGER}_D \qquad s \in \Omega_M}{r, i \vdash_u \psi}$$

PROPAGATE BACKWARD CREATE VIEW
$$\frac{\begin{array}{c} r, i+1 \vdash_u \psi \\ r^{i+1} = r^i \cdot \langle u, \texttt{CREATE}, o \rangle \cdot s \qquad 1 \leq i < |r| \qquad o \in \mathcal{VIEW}_D \qquad s \in \Omega_M \qquad \psi' = replace(\psi, o) \end{array}}{r, i \vdash_u \psi'}$$

FIGURE 6.12: Rules defining how the attacker propagates the knowledge.

PROPAGATE FORWARD INSERT/DELETE SUCCESS
$$\frac{\begin{array}{c} 1 < i \leq |r| \qquad r, i-1 \vdash_u \phi \qquad r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s \qquad s \in \Omega_M \qquad secEx(s_n) = \bot \\ Ex(s_n) = \emptyset \qquad s = \langle db, U, sec, T, V, h, actEff, tr \rangle \qquad revise(r^{i-1}, \phi, r^i) = \top \qquad op \in \{\texttt{INSERT}, \texttt{DELETE}\} \end{array}}{r, i \vdash_u \phi}$$

PROPAGATE FORWARD INSERT SUCCESS - 1
$$\frac{\begin{array}{c} 1 < i \leq |r| \qquad r, i-1 \vdash_u \phi \qquad r, i-1 \vdash_u R(\bar{t}) \qquad r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, \bar{t} \rangle \cdot s \\ s \in \Omega_M \qquad secEx(s_n) = \bot \qquad Ex(s_n) = \emptyset \qquad s = \langle db, U, sec, T, V, h, actEff, tr \rangle \end{array}}{r, i \vdash_u \phi}$$

PROPAGATE FORWARD DELETE SUCCESS - 1
$$\frac{\begin{array}{c} 1 < i \leq |r| \qquad r, i-1 \vdash_u \phi \qquad r, i-1 \vdash_u \neg R(\bar{t}) \qquad r^i = r^{i-1} \cdot \langle u, \texttt{DELETE}, R, \bar{t} \rangle \cdot s \\ s \in \Omega_M \qquad secEx(s_n) = \bot \qquad Ex(s_n) = \emptyset \qquad s = \langle db, U, sec, T, V, h, actEff, tr \rangle \end{array}}{r, i \vdash_u \phi}$$

PROPAGATE BACKWARD INSERT/DELETE SUCCESS
$$\frac{\begin{array}{c} 1 < i \leq |r| \qquad r, i \vdash_u \phi \qquad r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s \qquad s \in \Omega_M \qquad secEx(s_n) = \bot \\ Ex(s_n) = \emptyset \qquad s = \langle db, U, sec, T, V, h, actEff, tr \rangle \qquad revise(r^{i-1}, \phi, r^i) = \top \qquad op \in \{\texttt{INSERT}, \texttt{DELETE}\} \end{array}}{r, i-1 \vdash_u \phi}$$

PROPAGATE BACKWARD INSERT SUCCESS - 1
$$\frac{\begin{array}{c} 1 < i \leq |r| \qquad r, i \vdash_u \phi \qquad r, i-1 \vdash_u R(\bar{t}) \qquad r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, \bar{t} \rangle \cdot s \\ s \in \Omega_M \qquad secEx(s_n) = \bot \qquad Ex(s_n) = \emptyset \qquad s = \langle db, U, sec, T, V, h, actEff, tr \rangle \end{array}}{r, i-1 \vdash_u \phi}$$

PROPAGATE BACKWARD DELETE SUCCESS - 1
$$\frac{\begin{array}{c} 1 < i \leq |r| \qquad r, i \vdash_u \phi \qquad r, i-1 \vdash_u \neg R(\bar{t}) \qquad r^i = r^{i-1} \cdot \langle u, \texttt{DELETE}, R, \bar{t} \rangle \cdot s \\ s \in \Omega_M \qquad secEx(s_n) = \bot \qquad Ex(s_n) = \emptyset \qquad s = \langle db, U, sec, T, V, h, actEff, tr \rangle \end{array}}{r, i-1 \vdash_u \phi}$$

FIGURE 6.13: Rules regulating how information propagates in case of successful INSERT and DELETE.

$$\textsc{SELECT Success - 1}\quad \frac{1 < i \leq |r| \qquad r^i = r^{i-1}\cdot\langle u, \texttt{SELECT}, \phi\rangle\cdot s \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{SELECT}, \phi\rangle, \top, \top, \emptyset\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}{r, i \vdash_u \phi}$$

$$\textsc{SELECT Success - 2}\quad \frac{1 < i \leq |r| \qquad r^i = r^{i-1}\cdot\langle u, \texttt{SELECT}, \phi\rangle\cdot s \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{SELECT}, \phi\rangle, \top, \bot, \emptyset\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}{r, i \vdash_u \neg\phi}$$

$$\textsc{INSERT Success}\quad \frac{\begin{array}{c}1 < i \leq |r| \\ r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \overline{t}\rangle\cdot s \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{INSERT}, R, \overline{t}\rangle, \top, \top, \emptyset\rangle, \langle rS', \overline{t}, u, tr\rangle\rangle\end{array}}{r, i \vdash_u R(\overline{t})}$$

$$\textsc{INSERT Success - FD}\quad \frac{\begin{array}{c}1 < i \leq |r| \qquad r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \overline{t}\rangle\cdot s \\ l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{INSERT}, R, \overline{t}\rangle, \top, \top, E\rangle, \langle rS', \overline{t}, u, tr\rangle\rangle \\ \forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}' \in \Gamma \setminus E \qquad \overline{t} = (\overline{v}, \overline{w}, \overline{q})\end{array}}{r, l \vdash_u \neg\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}}$$

$$\textsc{INSERT Success - ID}\quad \frac{\begin{array}{c}1 < i \leq |r| \qquad r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \overline{t}\rangle\cdot s \\ l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{INSERT}, R, \overline{t}\rangle, \top, \top, E\rangle, \langle rS', \overline{t}, u, tr\rangle\rangle \\ \forall \overline{x}, \overline{z}.\, (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{y}.\, S(\overline{x}, \overline{y})) \in \Gamma \setminus E \qquad \overline{t} = (\overline{v}, \overline{w})\end{array}}{r, l \vdash_u \exists \overline{y}.\, S(\overline{v}, \overline{y})}$$

$$\textsc{DELETE Success}\quad \frac{\begin{array}{c}1 < i \leq |r| \\ r^i = r^{i-1}\cdot\langle u, \texttt{DELETE}, R, \overline{t}\rangle\cdot s \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{DELETE}, R, \overline{t}\rangle, \top, \top, \emptyset\rangle, \langle rS', \overline{t}, u, tr\rangle\rangle\end{array}}{r, i \vdash_u \neg R(\overline{t})}$$

$$\textsc{DELETE Success - ID}\quad \frac{\begin{array}{c}1 < i \leq |r| \qquad r^i = r^{i-1}\cdot\langle u, \texttt{DELETE}, R, \overline{t}\rangle\cdot s \\ l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{DELETE}, R, \overline{t}\rangle, \top, \top, E\rangle, \langle rS', \overline{t}, u, tr\rangle\rangle \\ \forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{y}.\, R(\overline{x}, \overline{y})) \in \Gamma \setminus E \qquad \overline{t} = (\overline{v}, \overline{w})\end{array}}{r, l \vdash_u \forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \overline{x} \neq \overline{v}) \vee \exists \overline{y}.\, (R(\overline{v}, \overline{y}) \wedge \overline{y} \neq \overline{w})}$$

$$\textsc{INSERT Exception}\quad \frac{\begin{array}{c}1 < i \leq |r| \qquad r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \overline{t}\rangle\cdot s \\ l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{INSERT}, R, \overline{t}\rangle, \top, \bot, E\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle \qquad E \neq \emptyset\end{array}}{r, l \vdash_u \neg R(\overline{t})}$$

$$\textsc{DELETE Exception}\quad \frac{\begin{array}{c}1 < i \leq |r| \qquad r^i = r^{i-1}\cdot\langle u, \texttt{DELETE}, R, \overline{t}\rangle\cdot s \\ l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{DELETE}, R, \overline{t}\rangle, \top, \bot, E\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle \qquad E \neq \emptyset\end{array}}{r, l \vdash_u R(\overline{t})}$$

$$\textsc{INSERT FD Exception}\quad \frac{\begin{array}{c}1 < i \leq |r| \\ r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \overline{t}\rangle\cdot s \qquad l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{INSERT}, R, \overline{t}\rangle, \top, \bot, E\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle \\ (\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}') \in E \qquad \overline{t} = (\overline{v}, \overline{w}, \overline{q})\end{array}}{r, l \vdash_u \exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}}$$

$$\textsc{INSERT ID Exception}\quad \frac{\begin{array}{c}1 < i \leq |r| \\ r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \overline{t}\rangle\cdot s \qquad l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{INSERT}, R, \overline{t}\rangle, \top, \bot, E\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle \\ \forall \overline{x}, \overline{z}.\, (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{y}.\, S(\overline{x}, \overline{y})) \in E \qquad \overline{t} = (\overline{v}, \overline{w})\end{array}}{r, l \vdash_u \forall \overline{x}, \overline{y}.\, S(\overline{x}, \overline{y}) \rightarrow \overline{x} \neq \overline{v}}$$

$$\textsc{DELETE ID Exception}\quad \frac{\begin{array}{c}1 < i \leq |r| \\ r^i = r^{i-1}\cdot\langle u, \texttt{DELETE}, R, \overline{t}\rangle\cdot s \qquad l \in \{i, i-1\} \qquad s = \langle db, U, sec, T, V, h, \langle\langle u, \texttt{DELETE}, R, \overline{t}\rangle, \top, \bot, E\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle \\ \forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{y}.\, R(\overline{x}, \overline{y})) \in E \qquad \overline{t} = (\overline{v}, \overline{w})\end{array}}{r, l \vdash_u \exists \overline{z}.\, S(\overline{v}, \overline{z}) \wedge \forall \overline{y}.\, (R(\overline{v}, \overline{y}) \rightarrow \overline{y} = \overline{w})}$$

$$\textsc{Integrity Constraint}\quad \frac{1 \leq i \leq |r| \qquad \gamma \in \Gamma}{r, i \vdash_u \gamma} \qquad \textsc{View}\quad \frac{1 \leq i \leq |r| \qquad v \in \mathit{last}(r^i).V \qquad r, i \vdash_u \psi \qquad \psi' = \mathit{replace}(\psi, v)}{r, i \vdash_u \psi'}$$

Figure 6.14: Rules defining how the attacker extracts knowledge from the run.

ROLLBACK BACKWARD - 1

$$\frac{r, i \vdash_u \phi \qquad n+1 < i \leq |r| \qquad s_1, s_2, \ldots, s_n \in \Omega_M \qquad t_1, \ldots, t_n \in \mathcal{TRIGGER}_D \qquad secEx(s_n) = \top \vee Ex(s_n) \neq \emptyset \qquad r^i = r^{i-n-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s_1 \cdot t_1 \cdot s_2 \cdot \ldots \cdot t_n \cdot s_n \qquad s_n = \langle db, U, sec, T, V, h, \langle t_n, when, stmt \rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon \rangle \rangle \qquad op \in \{\texttt{INSERT}, \texttt{DELETE}\}}{r, i-n-1 \vdash_u \phi}$$

ROLLBACK BACKWARD - 2

$$\frac{r, i \vdash_u \phi \qquad 1 < i \leq |r| \qquad secEx(s) = \top \vee Ex(s) \neq \emptyset \qquad op \in \{\texttt{INSERT}, \texttt{DELETE}\} \qquad r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s \qquad s = \langle db, U, sec, T, V, h, \langle \langle u, \texttt{op}, R, \bar{t} \rangle, v, v', E \rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon \rangle \rangle}{r, i-1 \vdash_u \phi}$$

ROLLBACK FORWARD - 1

$$\frac{r, i-n-1 \vdash_u \phi \qquad n+1 < i \leq |r| \qquad s_1, s_2, \ldots, s_n \in \Omega_M \qquad t_1, \ldots, t_n \in \mathcal{TRIGGER}_D \qquad secEx(s_n) = \top \vee Ex(s_n) \neq \emptyset \qquad r^i = r^{i-n-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s_1 \cdot t_1 \cdot s_2 \cdot \ldots \cdot t_n \cdot s_n \qquad s_n = \langle db, U, sec, T, V, h, \langle t_n, when, stmt \rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon \rangle \rangle \qquad op \in \{\texttt{INSERT}, \texttt{DELETE}\}}{r, i \vdash_u \phi}$$

ROLLBACK FORWARD - 2

$$\frac{r, i-1 \vdash_u \phi \qquad 1 < i \leq |r| \qquad secEx(s) = \top \vee Ex(s) \neq \emptyset \qquad op \in \{\texttt{INSERT}, \texttt{DELETE}\} \qquad r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s \qquad s = \langle db, U, sec, T, V, h, \langle \langle u, \texttt{op}, R, \bar{t} \rangle, v, v', E \rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon \rangle \rangle}{r, i \vdash_u \phi}$$

FIGURE 6.15: Rules regulating how information propagates in case of rollbacks.

REASONING

$$\frac{1 \leq i \leq |r| \qquad \Phi \subseteq \{\phi \mid r, i \vdash_u \phi\} \qquad \Phi \models_{fin} \gamma}{r, i \vdash_u \gamma}$$

FIGURE 6.16: Rules regulating the reasoning.

LEARN INSERT BACKWARD - 3

$$\frac{r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, \bar{t} \rangle \cdot s \qquad 1 < i \leq |r| \qquad s = \langle db, U, sec, T, V, h, aE, tr \rangle \qquad secEx(s) = \bot \qquad Ex(s) = \emptyset \qquad r, i-1 \vdash_u \psi \qquad r, i \vdash_u \neg\psi}{r, i-1 \vdash_u \neg R(\bar{t})}$$

LEARN DELETE BACKWARD - 3

$$\frac{r^i = r^{i-1} \cdot \langle u, \texttt{DELETE}, R, \bar{t} \rangle \cdot s \qquad 1 < i \leq |r| \qquad s = \langle db, U, sec, T, V, h, aE, tr \rangle \qquad secEx(s) = \bot \qquad Ex(s) = \emptyset \qquad r, i-1 \vdash_u \psi \qquad r, i \vdash_u \neg\psi}{r, i-1 \vdash_u R(\bar{t})}$$

FIGURE 6.17: Rules describing how the attacker learns facts about INSERT and DELETE commands.

PROPAGATE FORWARD DISABLED TRIGGER

$$\frac{r, i-1 \vdash_u \phi \qquad r^i = r^{i-1} \cdot t \cdot s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, stmt \rangle, tr \rangle \qquad secEx(s) = \bot \qquad t = \langle id, ow, ev, R, \psi, act, m \rangle \qquad r, i-1 \vdash_u \neg\psi[\overline{x}^{|R|} \mapsto tpl(last(r^{i-1}))]}{r, i \vdash_u \phi}$$

PROPAGATE BACKWARD DISABLED TRIGGER

$$\frac{r, i \vdash_u \phi \qquad r^i = r^{i-1} \cdot t \cdot s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, stmt \rangle, tr \rangle \qquad secEx(s) = \bot \qquad t = \langle id, ow, ev, R, \psi, act, m \rangle \qquad r, i-1 \vdash_u \neg\psi[\overline{x}^{|R|} \mapsto tpl(last(r^{i-1}))]}{r, i-1 \vdash_u \phi}$$

FIGURE 6.18: Rules regulating the propagation of information through disabled triggers.

Learn INSERT Forward

$$\frac{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R', \phi, act, m \rangle}{r, i \vdash_u R(\overline{t})}$$

Learn INSERT FD

$$\frac{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \quad l \in \{i, i-1\} \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R', \phi, act, m \rangle \quad \forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}' \in \Gamma \quad \overline{t} = (\overline{v}, \overline{w}, \overline{q})}{r, l \vdash_u \neg \exists \overline{y}, \overline{z}. R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}}$$

Learn INSERT FD - 1

$$\frac{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \overline{t} \rangle, \top, \top, E \rangle \rangle, tr \rangle \quad \overline{t} = (\overline{v}, \overline{w}, \overline{q}) \quad secEx(s) = \bot \quad t = \langle id, ow, ev, R', \phi, act, m \rangle \quad \forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}' \in \Gamma \setminus E}{r, i-1 \vdash_u \neg \exists \overline{y}, \overline{z}. R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}}$$

Learn INSERT ID

$$\frac{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \quad l \in \{i, i-1\} \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R', \phi, act, m \rangle \quad (\forall \overline{x}, \overline{z}. (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. S(\overline{x}, \overline{w})) \in \Gamma \quad \overline{t} = (\overline{v}, \overline{w})}{r, l \vdash_u \exists \overline{y}. S(\overline{v}, \overline{y})}$$

Learn INSERT ID - 1

$$\frac{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \overline{t} \rangle, \top, \top, E \rangle \rangle, tr \rangle \quad \overline{t} = (\overline{v}, \overline{w}) \quad secEx(s) = \bot \quad t = \langle id, ow, ev, R', \phi, act, m \rangle \quad (\forall \overline{x}, \overline{z}. (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. S(\overline{x}, \overline{w})) \in \Gamma \setminus E}{r, i-1 \vdash_u \exists \overline{y}. S(\overline{v}, \overline{y})}$$

Learn INSERT Backward - 1

$$\frac{1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R', \phi, act, m \rangle \quad r, i-1 \vdash_u \psi \quad r, i \vdash_u \neg \psi}{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

Learn INSERT Backward - 2

$$\frac{1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R', \phi, act, m \rangle \quad r, i-1 \vdash_u \psi \quad r, i \vdash_u \neg \psi}{r, i-1 \vdash_u \neg R(\overline{t})}$$

Figure 6.19: Extracting knowledge from triggers – part 1.

Learn DELETE Forward

$$\frac{\begin{array}{c} r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \qquad 1 < i \leq |r| \\ r^i = r^{i-1} \cdot t \cdot s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{DELETE}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \\ secEx(s) = \bot \qquad Ex(s) = \emptyset \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle \end{array}}{r, i \vdash_u \neg R(\overline{t})}$$

Learn DELETE - ID

$$\frac{\begin{array}{l} r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad invoker(last(r^{i-1})) = u \\ s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{DELETE}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \qquad l \in \{i, i-1\} \qquad secEx(s) = \bot \\ Ex(s) = \emptyset \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle \qquad (\forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})) \in \Gamma \qquad \overline{t} = (\overline{v}, \overline{w}) \end{array}}{r, l \vdash_u \forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \overline{x} \neq \overline{v}) \vee \exists \overline{y}.\, (R(\overline{v}, \overline{y}) \wedge \overline{y} \neq \overline{w})}$$

Learn DELETE ID - 1

$$\frac{\begin{array}{c} r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))] \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \\ invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{DELETE}, R, \overline{t} \rangle, \top, \top, E \rangle \rangle, tr \rangle \\ \overline{t} = (\overline{v}, \overline{w}) \qquad secEx(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle \qquad (\forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})) \in \Gamma \setminus E \end{array}}{r, i-1 \vdash_u \forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \overline{x} \neq \overline{v}) \vee \exists \overline{y}.\, (R(\overline{v}, \overline{y}) \wedge \overline{y} \neq \overline{w})}$$

Learn DELETE Backward - 1

$$\frac{\begin{array}{c} 1 < i \leq |r| \\ r^i = r^{i-1} \cdot t \cdot s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{DELETE}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \\ secEx(s) = \bot \qquad Ex(s) = \emptyset \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle \qquad r, i-1 \vdash_u \psi \qquad r, i \vdash_u \neg\psi \end{array}}{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

Learn DELETE Backward - 2

$$\frac{\begin{array}{c} 1 < i \leq |r| \\ r^i = r^{i-1} \cdot t \cdot s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{DELETE}, R, \overline{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \\ secEx(s) = \bot \qquad Ex(s) = \emptyset \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle \qquad r, i-1 \vdash_u \psi \qquad r, 1 \vdash_u \neg\psi \end{array}}{r, i-1 \vdash_u R(\overline{t})}$$

Learn GRANT/REVOKE Backward

$$\frac{\begin{array}{l} 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad invoker(last(r^{i-1})) = u \\ s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle op, u'', pr, u' \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \qquad secEx(s) = \bot \qquad Ex(s) = \emptyset \\ t = \langle id, ow, ev, R', \phi, act, m \rangle \qquad u', u'' \in U \qquad op \in \{\oplus, \oplus^*, \ominus\} \qquad last(r^{i-1}).sec \neq last(r^i).sec \end{array}}{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

Figure 6.20: Extracting knowledge from triggers – part 2.

PROPAGATE FORWARD TRIGGER ACTION

$$\frac{r, i-1 \vdash_u \psi \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad \mathit{invoker}(\mathit{last}(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, stmt \rangle, tr \rangle \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{secEx}(s) = \bot \qquad t = \langle id, ow, ev, R, \phi, act, m \rangle \qquad \mathit{revise}(r^{i-1}, \psi, r^i) = \top}{r, i \vdash_u \psi}$$

PROPAGATE BACKWARD TRIGGER ACTION

$$\frac{r, i \vdash_u \psi \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad \mathit{invoker}(\mathit{last}(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, stmt \rangle, tr \rangle \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{secEx}(s) = \bot \qquad t = \langle id, ow, ev, R, \phi, act, m \rangle \qquad \mathit{revise}(r^{i-1}, \psi, r^i) = \top}{r, i-1 \vdash_u \psi}$$

PROPAGATE FORWARD `INSERT` TRIGGER ACTION

$$\frac{r, i-1 \vdash_u \psi \qquad r, i-1 \vdash_u R(\bar{t}) \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad \mathit{invoker}(\mathit{last}(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \bar{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{secEx}(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle}{r, i \vdash_u \psi}$$

PROPAGATE FORWARD `DELETE` TRIGGER ACTION

$$\frac{r, i-1 \vdash_u \psi \qquad r, i-1 \vdash_u \neg R(\bar{t}) \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad \mathit{invoker}(\mathit{last}(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{DELETE}, R, \bar{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{secEx}(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle}{r, i \vdash_u \psi}$$

PROPAGATE BACKWARD `INSERT` TRIGGER ACTION

$$\frac{r, i \vdash_u \psi \qquad r, i-1 \vdash_u R(\bar{t}) \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad \mathit{invoker}(\mathit{last}(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{INSERT}, R, \bar{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{secEx}(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle}{r, i-1 \vdash_u \psi}$$

PROPAGATE BACKWARD `DELETE` TRIGGER ACTION

$$\frac{r, i \vdash_u \psi \qquad r, i-1 \vdash_u \neg R(\bar{t}) \qquad 1 < i \leq |r| \qquad r^i = r^{i-1} \cdot t \cdot s \qquad \mathit{invoker}(\mathit{last}(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle \langle u', \texttt{DELETE}, R, \bar{t} \rangle, \top, \top, \emptyset \rangle \rangle, tr \rangle \qquad \mathit{Ex}(s) = \emptyset \qquad \mathit{secEx}(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m \rangle}{r, i-1 \vdash_u \psi}$$

FIGURE 6.21: Rules for propagating knowledge through triggers.

TRIGGER FD INSERT DISABLED BACKWARD

$$\frac{1 < i \leq |r| \quad r^i = r^{i-1}{\cdot}t{\cdot}s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, stmt\rangle, tr\rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R, \phi, act, m\rangle \quad action(act, user(last(r^{i-1}), t), tpl(last(r^{i-1})) = \langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle \quad r, i-1 \vdash_u \exists \overline{y}, \overline{z}. R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w} \quad \forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. (R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}' \in \Gamma}{r, i-1 \vdash_u \neg\phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

TRIGGER ID INSERT DISABLED BACKWARD

$$\frac{1 < i \leq |r| \quad r^i = r^{i-1}{\cdot}t{\cdot}s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, stmt\rangle, tr\rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R, \phi, act, m\rangle \quad action(act, user(last(r^{i-1}), t), tpl(last(r^{i-1})) = \langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w})\rangle \quad r, i-1 \vdash_u \forall \overline{x}, \overline{y}. S(\overline{x}, \overline{y}) \rightarrow \overline{x} \neq \overline{v} \quad \forall \overline{x}, \overline{z}. R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. S(\overline{x}, \overline{w}) \in \Gamma}{r, i-1 \vdash_u \neg\phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

TRIGGER ID DELETE DISABLED BACKWARD

$$\frac{1 < i \leq |r| \quad r^i = r^{i-1}{\cdot}t{\cdot}s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, stmt\rangle, tr\rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R, \phi, act, m\rangle \quad action(act, user(last(r^{i-1}), t), tpl(last(r^{i-1})) = \langle u', \texttt{DELETE}, R, (\overline{v}, \overline{w})\rangle \quad r, i-1 \vdash_u \exists \overline{z}. S(\overline{v}, \overline{z}) \wedge \forall \overline{y}. (R(\overline{x}, \overline{y}) \rightarrow \overline{y} = \overline{w}) \quad \forall \overline{x}, \overline{z}. S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. R(\overline{x}, \overline{w}) \in \Gamma}{r, i-1 \vdash_u \neg\phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

TRIGGER GRANT DISABLED BACKWARD

$$\frac{1 < i \leq |r| \quad r^i = r^{i-1}{\cdot}t{\cdot}s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, stmt\rangle, tr\rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R', \phi, act, m\rangle \quad action(act, user(last(r^{i-1}), t), tpl(last(r^{i-1})) = \langle op, u'', p, u'\rangle \quad u', u'' \in U \quad op \in \{\oplus, \oplus^*\} \quad \langle op, u'', p, u'\rangle \notin last(r^{i-1}).sec \quad last(r^{i-1}).sec = sec}{r, i-1 \vdash_u \neg\phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

TRIGGER REVOKE DISABLED BACKWARD

$$\frac{1 < i \leq |r| \quad r^i = r^{i-1}{\cdot}t{\cdot}s \quad invoker(last(r^{i-1})) = u \quad s = \langle db, U, sec, T, V, h, \langle t, when, stmt\rangle, tr\rangle \quad secEx(s) = \bot \quad Ex(s) = \emptyset \quad t = \langle id, ow, ev, R', \phi, act, m\rangle \quad action(act, user(last(r^{i-1}), t), tpl(last(r^{i-1})) = \langle \ominus, u'', p, u'\rangle \quad u', u'' \in U \quad op \in \{\oplus, \oplus^*\} \quad \langle op, u'', p, u'\rangle \in last(r^{i-1}).sec \quad last(r^{i-1}).sec = sec}{r, i-1 \vdash_u \neg\phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

Figure 6.22: Extracting knowledge from disabled triggers.

LEARN FROM DENY - ACTIONS

$$\frac{r, r', r'' \in traces(L) \quad a \in \mathcal{A}_{D,u} \quad s, s' \in \Omega_M \quad 1 < i \leq |r| \quad r^i = r^{i-1}{\cdot}a{\cdot}s \quad r' = r''{\cdot}a{\cdot}s' \quad r^{i-1} \cong_{P,u} r'' \quad secEx(s) \neq secEx(s') \quad [\phi]^{last(r^{i-1}).db} = \top \quad [\phi]^{last(r''.db} = \bot}{r, i-1 \vdash_u \phi}$$

LEARN FROM DENY - TRIGGERS

$$\frac{r, r', r'' \in traces(L) \quad t \in \mathcal{TRIGGERS}_D \quad s, s' \in \Omega_M \quad 1 < i \leq |r| \quad r^i = r^{i-1}{\cdot}t{\cdot}s \quad r' = r''{\cdot}t{\cdot}s' \quad r^{i-1} \cong_{P,u} r'' \quad invoker(last(r^{i-1})) = u \quad invoker(last(r'')) = u \quad acC(s) \neq acC(s') \vee secEx(s) \neq secEx(s') \quad [\phi]^{last(r^{i-1}).db} = \top \quad [\phi]^{last(r''.db} = \bot}{r, i-1 \vdash_u \phi}$$

Figure 6.23: Extracting knowledge from the PDP.

Trigger INSERT FD Exception

$$\frac{1 < i \leq |r| \qquad r^i = r^{i-1}{\cdot}t{\cdot}s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle\langle u', \texttt{INSERT}, R, \overline{t}\rangle, \top, \top, E\rangle, tr\rangle \qquad secEx(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m\rangle \qquad (\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\ ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}') \in Ex(s) \qquad \overline{t} = (\overline{v}, \overline{w}, \overline{q})}{r, i-1 \vdash_u \exists \overline{y}, \overline{z}.\ R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}}$$

Trigger INSERT ID Exception

$$\frac{1 < i \leq |r| \qquad r^i = r^{i-1}{\cdot}t{\cdot}s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle\langle u', \texttt{INSERT}, R, \overline{t}\rangle, \top, \top, E\rangle, tr\rangle \qquad secEx(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m\rangle \qquad (\forall \overline{x}, \overline{z}.\ (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\ S(\overline{x}, \overline{w})) \in Ex(s) \qquad \overline{t} = (\overline{v}, \overline{w})}{r, i-1 \vdash_u \forall \overline{x}, \overline{y}.\ S(\overline{x}, \overline{y}) \rightarrow \overline{x} \neq \overline{v}}$$

Trigger DELETE ID Exception

$$\frac{1 < i \leq |r| \qquad r^i = r^{i-1}{\cdot}t{\cdot}s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, when, \langle\langle u', \texttt{DELETE}, R, \overline{t}\rangle, \top, \top, E\rangle, tr\rangle \qquad secEx(s) = \bot \qquad t = \langle id, ow, ev, R', \phi, act, m\rangle \qquad (\forall \overline{x}, \overline{z}.\ (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\ R(\overline{x}, \overline{w})) \in Ex(s) \qquad \overline{t} = (\overline{v}, \overline{w})}{r, i-1 \vdash_u \exists \overline{z}.\ S(\overline{v}, \overline{z}) \wedge \forall \overline{y}.\ (R(\overline{v}, \overline{y}) \rightarrow \overline{y} = \overline{w})}$$

Trigger Exception

$$\frac{1 < i \leq |r| \qquad r^i = r^{i-1}{\cdot}t{\cdot}s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, \langle\langle u', \texttt{SELECT}, \phi[\overline{x} \mapsto tpl(last(r^{i-1}))]\rangle, \top, \top, \emptyset\rangle, stmt, tr\rangle \qquad secEx(s) = \top \vee Ex(s) \neq \emptyset \qquad t = \langle id, ow, ev, R, \phi, act, m\rangle}{r, i-1 \vdash_u \phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]}$$

Trigger INSERT Exception

$$\frac{1 < i \leq |r| \qquad r^i = r^{i-1}{\cdot}t{\cdot}s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, \langle\langle u', \texttt{SELECT}, \phi\rangle, \top, \top, \emptyset\rangle, \langle\langle u', \texttt{INSERT}, R, \overline{t}\rangle, res, aC, E\rangle, tr\rangle \qquad secEx(s) = \bot \qquad Ex(s) \neq \emptyset \qquad t = \langle id, ow, ev, R', \phi, act, m\rangle}{r, i-1 \vdash_u \neg R(\overline{t})}$$

Trigger DELETE Exception

$$\frac{1 < i \leq |r| \qquad r^i = r^{i-1}{\cdot}t{\cdot}s \qquad invoker(last(r^{i-1})) = u \qquad s = \langle db, U, sec, T, V, h, \langle t, \langle\langle u', \texttt{SELECT}, \phi\rangle, \top, \top, \emptyset\rangle, \langle\langle u', \texttt{DELETE}, R, \overline{t}\rangle, res, aC, E\rangle, tr\rangle \qquad secEx(s) = \bot \qquad Ex(s) \neq \emptyset \qquad t = \langle id, ow, ev, R', \phi, act, m\rangle}{r, i-1 \vdash_u R(\overline{t})}$$

Trigger Rollback INSERT

$$\frac{n+1 < i \leq |r| \qquad s_1, s_2, \ldots, s_n \in \Omega_M \qquad t_1, \ldots, t_n \in \mathcal{TRIGGER}_D \qquad secEx(s_n) = \top \vee Ex(s_n) \neq \emptyset \qquad r^i = r^{i-n-1}{\cdot}\langle u, \texttt{INSERT}, R, \overline{t}\rangle{\cdot}s_1{\cdot}t_1{\cdot}s_2{\cdot}\ldots{\cdot}t_n{\cdot}s_n \qquad s_n = \langle db, U, sec, T, V, h, \langle t_n, when, stmt\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}{r, i \vdash_u \neg R(\overline{t})}$$

Trigger Rollback DELETE

$$\frac{n+1 < i \leq |r| \qquad s_1, s_2, \ldots, s_n \in \Omega_M \qquad t_1, \ldots, t_n \in \mathcal{TRIGGER}_D \qquad secEx(s_n) = \top \vee Ex(s_n) \neq \emptyset \qquad r^i = r^{i-n-1}{\cdot}\langle u, \texttt{DELETE}, R, \overline{t}\rangle{\cdot}s_1{\cdot}t_1{\cdot}s_2{\cdot}\ldots{\cdot}t_n{\cdot}s_n \qquad s_n = \langle db, U, sec, T, V, h, \langle t_n, when, stmt\rangle, \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle\rangle}{r, i \vdash_u R(\overline{t})}$$

Figure 6.24: Extracting knowledge from trigger's exceptions.

6.8.2 Enforcing Database Integrity

Here, we define the access control function f_{int}, which models the f_{int} procedure described in Section 6.5. The function f_{int} is as follows:

$$f_{int}(s,a) = \begin{cases} \top & \text{if } trigger(s) = \epsilon \wedge s \rightsquigarrow^{appr}_{auth} a \\ \top & \text{if } trigger(s) = t \wedge t \neq \epsilon \wedge a = trigCond(s) \\ \top & \text{if } trigger(s) = \langle id, ow, e, R, \phi, st, A\rangle \wedge a = trigAct(s) \wedge \\ & \quad s \rightsquigarrow^{appr}_{auth} getAction(st, invoker(s), tpl(s)) \wedge s \rightsquigarrow^{appr}_{auth} getAction(st, ow, tpl(s)) \\ \top & \text{if } trigger(s) = \langle id, ow, e, R, \phi, st, O\rangle \wedge a = trigAct(s) \wedge \\ & \quad s \rightsquigarrow^{appr}_{auth} getAction(st, ow, tpl(s)) \\ \bot & \text{otherwise} \end{cases}$$

The function $trigCond(s)$ (respectively $trigAct(s)$) returns the condition (respectively the action) associated with the trigger $trigger(s)$. If $trigger(s) = \langle id, ow, e, R, \phi, st, O\rangle$, then $trigAct(s) = getAction(st, ow, tpl(s))$ and $trigCond(s) = \langle ow, \texttt{SELECT}, \phi[\overline{x}^{|R|} \mapsto tpl(s)]\rangle$. If $trigger(s) = \langle id, ow, e, R, \phi, st, A\rangle$, then $trigAct(s) = getAction(st, invoker(s), tpl(s))$ and $trigCond(s) = \langle invoker(s), \texttt{SELECT}, \phi[\overline{x}^{|R|} \mapsto tpl(s)]\rangle$.

The relation $\rightsquigarrow^{appr}_{auth} \subseteq \Omega_M \times (\mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D)$ is the smallest relation satisfying the inference rules given in Figure 6.25. Informally, $\rightsquigarrow^{appr}_{auth}$ is obtained from $\rightsquigarrow_{auth}$ by replacing determinacy with a sound under-approximation. As we show in Appendix C, $\rightsquigarrow^{appr}_{auth}$ is a sound and computable under-approximation of the relation $\rightsquigarrow_{auth}$. In the rules, we use a number of auxiliary functions. The most important ones are:

(a) The aT (respectively aV) function that takes as input a database state, an operator op in $\{\oplus, \oplus^*\}$, and a user, and returns the set of tables (respectively views) that the user is authorized to read (if $op = \oplus$) or to delegate the read access to other users (if $op = \oplus^*$) according to our approximation of $\rightsquigarrow_{auth}$. These functions are defined as follows:

$$\begin{aligned} aT(\langle db,U, sec, T, V, c\rangle, op, u) = \{R \in D \mid \\ (u = admin \wedge \exists u' \in U, op' \in \{\oplus^*, op\}.\ \langle db, U, sec, T, V, c\rangle \rightsquigarrow^{appr}_{auth} \langle op', u, \langle \texttt{SELECT}, R\rangle, u'\rangle) \vee \\ \exists u' \in U, g \in sec, op' \in \{\oplus^*, op\}.\ g = \langle op', u, \langle \texttt{SELECT}, R\rangle, u'\rangle \wedge \langle db, U, sec, T, V, c\rangle \rightsquigarrow^{appr}_{auth} g\} \end{aligned}$$

$$\begin{aligned} aV(\langle db,U, sec, T, V, c\rangle, op, u) = \{V \in V \cap \mathcal{VIEW}^{owner}_D \mid \\ (u = admin \wedge \exists u' \in U, op' \in \{\oplus^*, op\}.\ \langle db, U, sec, T, V, c\rangle \rightsquigarrow^{appr}_{auth} \langle op', u, \langle \texttt{SELECT}, V\rangle, u'\rangle) \vee \\ \exists u' \in U, g \in sec, op' \in \{\oplus^*, op\}.\ g = \langle op', u, \langle \texttt{SELECT}, V\rangle, u'\rangle \wedge \langle db, U, sec, T, V, c\rangle \rightsquigarrow^{appr}_{auth} g\} \end{aligned}$$

(b) The *apprDet* function is used to determine whether a set of tables and a set of views completely determine the result of a formula ϕ in all possible database states. Note that the function *apprDet* is a sound under-approximation of *query determinacy* [124].

A sound under-approximation of query determinacy

Here we define our sound under-approximation of query determinacy, implemented in the function *apprDet*. In the following, we assume that both the formula ϕ and the set of views V in the state s contain just views with owner's privileges. This is without loss of generality since views with activator's privileges are just syntactic sugar (they do not disclose additional information to a user u other than what he is already authorized to read because they are executed under u's privileges).

Let M be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, and $\langle v, o, q, O\rangle$ be a view with owner's privileges. We denote by $inline_M(\langle v, o, q, O\rangle, s)$ the view $\langle v, o, q', O\rangle$ where q' is obtained from q by replacing all occurrences of views in V with owner's privileges with their definitions. Note that $inline_M$ does not compute a fix-point, i.e., if a view's definition refers to another view, the latter is not replaced with its definition.

The *apprDet* function takes as input a system configuration M, a set of tables T, a set of views with owner's privileges V, a query ϕ, and a partial M-state, and returns $\top$ if the tables and views in T and V determine the result of ϕ. The function is formalized below, and it relies on $extend(M, s, V)$, which is the smallest set satisfying the following recurrence relation: $extend(M, s, V) = V \cup \{inline(v, s) \mid v \in extend(M, s, V)\}$.

$$apprDet(T,V,\phi,s,M) = \begin{cases} \top & \text{if } \exists \langle v,o,q,O\rangle \in extend(M,s,V).\, q = \{\overline{x} \mid \phi(\overline{x})\} \\ \top & \text{if } \phi = (x = v) \vee \phi = \top \vee \phi = \bot \\ \top & \text{if } \phi = R(\overline{x}) \wedge R \in T \\ \top & \text{if } \phi = V'(\overline{x}) \wedge \exists u \in \mathcal{U}, q \in RC.\, \langle V', u, q, O\rangle \in V \\ \top & \text{if } \phi = (\psi \wedge \gamma) \wedge apprDet(T,V,\psi,s,M) = \top \wedge \\ & \quad apprDet(T,V,\gamma,s,M) = \top \\ \top & \text{if } \phi = (\psi \vee \gamma) \wedge apprDet(T,V,\psi,s,M) = \top \wedge \\ & \quad apprDet(T,V,\gamma,s,M) = \top \\ \top & \text{if } \phi = (\neg\psi) \wedge apprDet(T,V,\psi,s,M) = \top \\ \top & \text{if } \phi = (\exists x.\, \psi) \wedge apprDet(T,V,\psi,s,M) = \top \\ \top & \text{if } \phi = (\forall x.\, \psi) \wedge apprDet(T,V,\psi,s,M) = \top \\ \bot & \text{otherwise} \end{cases}$$

Observe that our under-approximation of query determinacy ignores the integrity constraints. This is always sound since if $D, \emptyset \vdash Q \twoheadrightarrow q$, then $D, \Gamma \vdash Q \twoheadrightarrow q$.

6.8.3 Enforcing Data Confidentiality

Here, we define the PDP $f^u_{conf}(s,a)$, which models the function $f_{conf}(s,a,u)$ from Section 6.5. The PDP f^u_{conf} is shown in Figure 6.26. The function is parametrized by the user u against which the PDP provides data confidentiality. The mapping between the PDP f^u_{conf} and the pseudo-code shown in Figure 6.7 is immediate.

The PDP f^u_{conf} uses a number of auxiliary functions. Recall that the function tr takes as input an M-state $s \in \Omega_M$ and returns the definition of the trigger that the system is executing. If the system is not executing any trigger, then $tr(s) = \epsilon$. Equivalently, $tr(s)$ is the first trigger in the sequence of triggers returned by $triggers(s)$.

The function $tDet$ takes as input a view $v = \langle i, o, \{\overline{x}|\phi\}, m\rangle \in \mathcal{VIEW}_D$, a state $s \in \Omega_M$, and a system configuration $M = \langle D, \Gamma\rangle$ and returns as output the smallest set of tables in D that determines v, namely the smallest set $T \in 2^D$ such that $apprDet(T, \emptyset, \phi, s, M)$ holds, where $apprDet$ is defined in Section 6.8.2. Note that such a set is always unique.

The function $noLeak$ takes as input a state s, an `INSERT` or `DELETE` action a, and a user u and it checks whether the execution of the action a may leak sensitive information through the views that the user u can read, as shown in Example 6.5. Note that the function $noLeak$ returns $\top$ if there is no leakage of sensitive information and returns $\bot$ if the action a may leak sensitive information through the views the user u can read in the state s. Formally, $noLeak(s, \langle u', op, R, \overline{t}\rangle, u)$ is defined as follows:

$$\begin{cases} \top & \text{if } u' = u \wedge trigger(s) = \epsilon \wedge \forall v \in \mathcal{VIEW}_D.\, ((\langle \oplus, \texttt{SELECT}, v\rangle \in permissions(s,u) \wedge \\ & \quad R \in tDet(v,s,M)) \rightarrow (\forall o \in tDet(v,s,M).\, \langle \oplus, \texttt{SELECT}, o\rangle \in permissions(s,u))) \\ \top & \text{if } invoker(s) = u \wedge trigger(s) \neq \epsilon \wedge \forall v \in \mathcal{VIEW}_D.\, ((\langle \oplus, \texttt{SELECT}, v\rangle \in permissions(s,u) \wedge \\ & \quad R \in tDet(v,s,M)) \rightarrow (\forall o \in tDet(v,s,M).\, \langle \oplus, \texttt{SELECT}, o\rangle \in permissions(s,u))) \\ \bot & \text{otherwise} \end{cases}$$

We now define the Dep, $getInfoS$, $getInfoV$, and $getInfo$ functions. The function Dep retrieves all integrity constraints that may be violated by executing an `INSERT` or `DELETE` action. Formally, $Dep(\langle u, \texttt{INSERT}, R, \overline{t}\rangle, \Gamma)$ returns the set containing all the formulae in Γ of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, (R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}'$ or $\forall \overline{x}, \overline{z}.\, R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w})$, whereas $Dep(\langle u, \texttt{DELETE}, R, \overline{t}\rangle, \Gamma)$ returns the set containing all the formulae in Γ of the form $\forall \overline{x}, \overline{z}.\, S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})$.

The function $getInfoS$ returns the sentence capturing what an attacker may learn from the integrity constraint ϕ in case an action is executed successfully. The function is defined as follows:

- $getInfoS(\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle, \phi^R_{funct})$ is the formula $\neg\exists \overline{y}, \overline{z}.R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$, where ϕ^R_{funct} is a formula of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, (R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}'$.
- $getInfoS(\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w})\rangle, \phi^{R,S}_{incl})$ is the formula $\exists \overline{y}.\, S(\overline{v}, \overline{y})$, where $\phi^{R,S}_{incl}$ is a formula of the form $\forall \overline{x}, \overline{z}.\, R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w})$.
- $getInfoS(\langle u, \texttt{DELETE}, R, (\overline{v}, \overline{w})\rangle, \phi^{S,R}_{incl})$ is the formula $\forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \overline{x} \neq \overline{v}) \vee \exists \overline{y}.\, (R(\overline{v}, \overline{y}) \wedge \overline{y} \neq \overline{w})$, where $\phi^{S,R}_{incl}$ is a formula of the form $\forall \overline{x}, \overline{z}.\, S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})$.
- $getInfoS(act, \phi) = \top$ otherwise.

The function $getInfoV$ returns the sentence capturing what an attacker may learn from the integrity constraint ϕ in case an action violates the constraint ϕ. The function is as follows:

- $getInfoV(\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle, \phi^R_{funct})$ is the formula $\exists \overline{y}, \overline{z}.R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$, where ϕ^R_{funct} is a formula of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, (R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}'$.

SELECT
$$\frac{u \in U \qquad q \in RC}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle u, \texttt{SELECT}, q\rangle}$$

ADD USER
$$\frac{u \in \mathcal{U} \qquad u' = admin}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle u', \texttt{ADD\_USER}, u\rangle}$$

INSERT-DELETE
$$\frac{\begin{array}{c}u, u' \in U \qquad R \in D \qquad \bar{t} \in \mathbf{dom}^{|R|} \qquad g = \langle op, u, \langle op', R\rangle, u'\rangle \\ g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} g \qquad op \in \{\oplus, \oplus^*\} \qquad op' \in \{\texttt{INSERT}, \texttt{DELETE}\}\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle u, op', R, \bar{t}\rangle}$$

CREATE VIEW
$$\frac{\begin{array}{c}u, u' \in U \\ v \in \mathcal{VIEW}_D \qquad g = \langle op, u, \langle \texttt{CREATE VIEW}\rangle, u'\rangle \qquad g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} g \qquad op \in \{\oplus, \oplus^*\}\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle u, \texttt{CREATE}, v\rangle}$$

CREATE TRIGGER
$$\frac{\begin{array}{c}u, u' \in U \qquad t = \langle id, ow, ev, R, \phi, stmt, m\rangle \qquad t \in \mathcal{TRIGGER}_D \\ g = \langle op, u, \langle \texttt{CREATETRIGGER}, R\rangle, u'\rangle \qquad g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} g \qquad op \in \{\oplus, \oplus^*\}\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle u, \texttt{CREATE}, t\rangle}$$

INSERT-DELETE ADMIN
$$\frac{R \in D \qquad \bar{t} \in \mathbf{dom}^{|R|} \qquad op' \in \{\texttt{INSERT}, \texttt{DELETE}\}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle admin, op', R, \bar{t}\rangle}$$

CREATE VIEW ADMIN
$$\frac{v \in \mathcal{VIEW}_D}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle admin, \texttt{CREATE}, v\rangle}$$

CREATE TRIGGER ADMIN
$$\frac{t \in \mathcal{TRIGGER}_D}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle admin, \texttt{CREATE}, t\rangle}$$

REVOKE
$$\frac{\begin{array}{c}u, u' \in U \qquad priv \in \mathcal{PRIV}_D \qquad s = \langle db, U, sec, T, V, c\rangle \\ s' = \langle db, U, sec', T, V, c\rangle \qquad s' = applyRev(s, \langle \ominus, u, p, u'\rangle) \qquad \forall g \in sec'.\, s' \rightsquigarrow_{auth}^{appr} g\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle \ominus, u, priv, u'\rangle}$$

GRANT-1
$$\frac{\begin{array}{c}u, u', u'' \in U \\ op \in \{\oplus, \oplus^*\} \qquad priv \in \mathcal{PRIV}_D \qquad g = \langle \oplus^*, u', priv, u''\rangle \qquad g \in sec \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} g\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle op, u, priv, u'\rangle}$$

GRANT-2
$$\frac{u \in U \qquad op \in \{\oplus, \oplus^*\} \qquad priv \in \mathcal{PRIV}_D \setminus \mathcal{PRIV}_D^{\texttt{SELECT}, \mathcal{VIEW}^{owner}{}_D}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle op, u, priv, admin\rangle}$$

GRANT-3
$$\frac{\begin{array}{c}u, owner \in U \qquad op \in \{\oplus, \oplus^*\} \\ priv = \langle \texttt{SELECT}, v\rangle \qquad v = \langle id, owner, q, O\rangle \qquad v \in V \qquad T' = aT(\langle db, U, sec, T, V, c\rangle, \oplus^*, owner) \\ V' = aV(\langle db, U, sec, T, V, c\rangle, \oplus^*, owner) \qquad apprDet(T', V', q) = \top\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle op, u, priv, owner\rangle}$$

GRANT-4
$$\frac{\begin{array}{c}u, owner \in U \qquad op \in \{\oplus, \oplus^*\} \qquad priv = \langle \texttt{SELECT}, v\rangle \\ v = \langle id, owner, q, O\rangle \qquad v \in V \qquad owner \neq admin \qquad T' = aT(\langle db, U, sec, T, V, c\rangle, \oplus, owner) \\ V' = aV(\langle db, U, sec, T, V, c\rangle, \oplus, owner) \qquad apprDet(T', V', q) = \top\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle op, u, priv, admin\rangle}$$

GRANT-5
$$\frac{u, owner \in U \qquad op \in \{\oplus, \oplus^*\} \qquad v \in V \qquad priv = \langle \texttt{SELECT}, v\rangle \qquad v = \langle id, owner, q, A\rangle}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} \langle op, u, priv, owner\rangle}$$

EXECUTE TRIGGER-1
$$\frac{\begin{array}{c}t = \langle id, ow, ev, R, \phi, stmt, O\rangle \\ t \in T \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} getAction(stmt, ow, tpl(c)) \qquad [\phi[\overline{x}^{|R|} \mapsto tpl(c)]]^{db} = \top\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} t}$$

EXECUTE TRIGGER-2
$$\frac{\begin{array}{c}t = \langle id, ow, ev, R, \phi, stmt, A\rangle \qquad t \in T \qquad \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} getAction(stmt, invoker(c), tpl(c)) \\ \langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} getAction(stmt, ow, tpl(c)) \qquad [\phi[\overline{x}^{|R|} \mapsto tpl(c)]]^{db} = \top\end{array}}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} t}$$

EXECUTE TRIGGER-3
$$\frac{t = \langle id, ow, ev, R, \phi, stmt, m\rangle \qquad t \in T \qquad [\phi[\overline{x}^{|R|} \mapsto tpl(c)]]^{db} = \bot}{\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth}^{appr} t}$$

FIGURE 6.25: Definition of the $\rightsquigarrow_{auth}^{appr}$ relation.

$$f^u_{conf}(s, act) = \begin{cases} f^u_{conf,\mathtt{S}}(s, act) & \text{if } act = \langle u', \mathtt{SELECT}, q\rangle \\ f^u_{conf,\mathtt{I,D}}(s, act) & \text{if } act = \langle u', \mathtt{INSERT}, R, \bar{t}\rangle \\ f^u_{conf,\mathtt{I,D}}(s, act) & \text{if } act = \langle u', \mathtt{DELETE}, R, \bar{t}\rangle \\ f^u_{conf,\mathtt{G,R}}(s, act) & \text{if } act = \langle op, u'', p, u'\rangle \wedge op \in \{\oplus, \oplus^*\} \\ \top & \text{if } u = admin \\ \top & \text{otherwise} \end{cases}$$

$$f^u_{conf,\mathtt{I,D}}(s, act) = \begin{cases} secure(u, getInfo(act), s) \wedge & \text{if } act = \langle u, op, R, \bar{t}\rangle \wedge trigger(s) = \epsilon \\ \bigwedge_{\gamma \in Dep(act,\Gamma)} secure(u, getInfoS(\gamma, act), s) & \quad \wedge noLeak(s, act, u) = \top \\ \quad \wedge\ secure(u, getInfoV(\gamma, act), s) & \\ \bot & \text{if } act = \langle u, op, R, \bar{t}\rangle \wedge trigger(s) = \epsilon \\ & \quad \wedge\ noLeak(s, act, u) = \bot \\ secure(u, getInfo(act), s) \wedge & \text{if } invoker(s) = u \wedge trigger(s) \neq \epsilon \\ \bigwedge_{\gamma \in Dep(act,\Gamma)} secure(u, getInfoS(\gamma, act), s) & \quad \wedge\ noLeak(s, act, u) = \top \\ \quad \wedge\ secure(u, getInfoV(\gamma, act), s) & \\ \bot & \text{if } invoker(s) = u \wedge trigger(s) \neq \epsilon \\ & \quad \wedge\ noLeak(s, act, u) = \bot \\ \top & \text{otherwise} \end{cases}$$

$$f^u_{conf,\mathtt{S}}(s, \langle u', \mathtt{SELECT}, q\rangle) = \begin{cases} secure(u, q, s) & \text{if } u' = u \wedge trigger(s) = \epsilon \\ secure(u, q, s) & \text{if } invoker(s) = u \wedge trigger(s) \neq \epsilon \\ \top & \text{otherwise} \end{cases}$$

$$f^u_{conf,\mathtt{G}}(s, \langle op, u'', p, u'\rangle) = \begin{cases} \bot & \text{if } u'' = u \wedge u' = u \wedge trigger(s) = \epsilon \wedge op \in \{\oplus, \oplus^*\} \wedge \\ & \quad p = \langle \mathtt{SELECT}, O\rangle \wedge \langle \oplus, \mathtt{SELECT}, O\rangle \notin permissions(s, u) \\ \bot & \text{if } u'' = u \wedge invoker(s) = u \wedge trigger(s) \neq \epsilon \wedge op \in \{\oplus, \oplus^*\} \wedge \\ & \quad p = \langle \mathtt{SELECT}, O\rangle \wedge \langle \oplus, \mathtt{SELECT}, O\rangle \notin permissions(s, u) \\ \top & \text{otherwise} \end{cases}$$

FIGURE 6.26: Access control function f^u_{conf}.

$$\frac{free(\phi \wedge \psi) = free(\phi) \quad M = \langle D, \Gamma \rangle \quad free(\phi) \neq \emptyset}{\phi \wedge \psi \subseteq_M \phi} \text{ Conjunction} \qquad \frac{free(\phi) = free(\phi \vee \psi) \quad M = \langle D, \Gamma \rangle \quad free(\phi) \neq \emptyset}{\phi \subseteq_M \phi \vee \psi} \text{ Disjunction}$$

$$\frac{M = \langle D, \Gamma \rangle \quad n > 1 \quad free(\phi) = \{x_1, \ldots, x_n\} \quad free(\psi) = \{y_1, \ldots, y_n\} \quad 1 \leq i \leq n \quad \phi \subseteq_M \psi}{\exists x_i.\phi \subseteq_M \exists y_i.\phi} \text{ Projection} \qquad \frac{M = \langle D, \Gamma \rangle \quad n > 0 \quad free(\phi) = \{x_1, \ldots, x_n\} \quad free(\psi) = \{y_1, \ldots, y_n\} \quad \phi[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n] = \psi}{\phi \subseteq_M \psi} \text{ Identity}$$

$$\frac{M = \langle D, \Gamma \rangle \quad |\overline{x}| > 0 \quad \forall \overline{x}, \overline{z}.\, (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w})) \in \Gamma}{\exists \overline{z}.\, R(\overline{x}, \overline{z}) \subseteq_M \exists \overline{w}.\, S(\overline{x}, \overline{w})} \begin{array}{l} \text{Inclusion} \\ \text{Dependency} \end{array}$$

$$\frac{\phi \subseteq_M \gamma \quad \gamma \subseteq_M \psi \quad M = \langle D, \Gamma \rangle \quad free(\phi) \neq \emptyset \quad free(\psi) \neq \emptyset \quad free(\gamma) \neq \emptyset}{\phi \subseteq_M \psi} \text{ Transitivity}$$

FIGURE 6.27: Containment rules.

- $getInfoV(\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w})\rangle, \phi^{R,S}_{incl})$ is the formula $\forall \overline{x}, \overline{y}.\, S(\overline{x}, \overline{y}) \rightarrow \overline{x} \neq \overline{v}$, where $\phi^{R,S}_{incl}$ is a formula of the form $\forall \overline{x}, \overline{z}.\, R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w})$.
- $getInfoV(\langle u, \texttt{DELETE}, R, (\overline{v}, \overline{w})\rangle, \phi^{S,R}_{incl})$ is the formula $\exists \overline{z}.\, S(\overline{v}, \overline{z}) \wedge \forall \overline{y}.\, (R(\overline{v}, \overline{y}) \rightarrow \overline{y} = \overline{w})$, where $\phi^{S,R}_{incl}$ is a formula of the form $\forall \overline{x}, \overline{z}.\, S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})$.
- $getInfoV(act, \phi) = \top$ otherwise.

Finally, the function *getInfo*, which returns the information an attacker may learn by the successful execution of an `INSERT` or `DELETE` command, is as follows:

$$getInfo(\langle u, op, R, \overline{t}\rangle) = \begin{cases} \neg R(\overline{t}) & \text{if } op = \texttt{INSERT} \\ R(\overline{t}) & \text{if } op = \texttt{DELETE} \end{cases}$$

Checking a judgment's security

We now define the $secure : \mathcal{U} \times RC_{bool} \times \Omega_M \rightarrow \{\top, \bot\}$ function that determines whether a given judgment is secure. In more detail, the *secure* function is as follows:

$$secure(u, \phi, s) = \begin{cases} \top & \text{if } [\phi^{rw}_{s,u}]^{s.db} = \bot \\ \bot & \text{otherwise} \end{cases}$$

Again, in the following, we assume that both the formula ϕ and the set of views V in the state s contain just views with owner's privileges. The extension to the general case is straightforward.

Before defining the $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ rewritings, we define query containment. Let $M = \langle D, \Gamma \rangle$ be a system configuration. Given two formulae $\phi(\overline{x})$ and $\psi(\overline{y})$, we write $\phi \subseteq_M \psi$ to denote that ϕ is contained in ψ, i.e., $\forall db \in \Omega^{\Gamma}_D.\, [\{\overline{x} \mid \phi\}]^{db} \subseteq [\{\overline{y} \mid \psi\}]^{db}$. Determining whether $\phi \subseteq_M \psi$ holds is undecidable for the relational calculus [10]. Hence, we develop a sound, under-approximation of query containment. Figure 6.27 describes the rules defining our under-approximation. For simplicity's sake, the rules are defined only for relational calculus formulae that do not use views. To check whether $\phi \subseteq_M \psi$ holds for two formulae ϕ and ψ that use views, we first compute the formulae ϕ' and ψ', obtained by replacing views' identifiers with their definitions, and then we check whether $\phi' \subseteq_M \psi'$ using the rules in Figure 6.27. This preserves containment since ϕ and ψ are semantically equivalent to ϕ' and ψ'. Observe that in the rules, we assume that there is a total ordering $\preceq_{var}$ over the set of all possible variable identifiers. This ensures that, given a formula ϕ, there is a unique non-boolean query $\{\overline{x} \mid \phi\}$ associated with it, where the variables in $\overline{x}$ are those in $free(\phi)$ ordered according to $\preceq_{var}$.

Given a table or a view O and a sequence of distinct integers $\overline{i} = i_1 \cdot \ldots \cdot i_n$ such that $1 \leq i_j \leq |O|$ for all $1 \leq j \leq n$, where $0 \leq n < |O|$, the *$\overline{i}$-projection of O*, denoted by $O_{\overline{i}}$, is the formula $\exists x_{i_1}, \ldots, x_{i_n}.\, O(x_1, \ldots, x_{|O|})$. Given a database schema D and a set of views V defined over D, we denote by $extVocabulary(D, V)$ the extended vocabulary obtained by defining all possible projections of tables in D and views in V, i.e., for each $O \in D \cup V$, we define a predicate $O_{\overline{i}}$ for each projection $\exists x_{i_1}, \ldots, x_{i_n}.\, O(x_1, \ldots, x_{|O|})$ of O. Furthermore, given a relational calculus formula ϕ over D, we denote by $extVoc_{V,D}(\phi)$ the formula obtained by replacing all sub-formulae of the form $\exists \overline{x}.R(\overline{x}, \overline{y})$ with the predicates in $extVocabulary(D, V)$ representing the corresponding projections $R_{\overline{i}}$. Finally, we denote by $inline_{D,V}(\phi)$, where ϕ is a relational calculus formula over $extVocabulary(D, V)$, the formula ϕ' obtained by replacing all predicates associated with projections with the corresponding formulae.

Let S be a predicate in $extVocabulary(D, V)$ and s be an M-state. We denote by $S_s^\top$ the set of all projections of tables in D and views in V that are contained in S, i.e., $S_s^\top := \{R \in extVocabulary(D, V) \mid R(\overline{x}) \subseteq_M S(\overline{y})\}$[2]. Similarly, we denote by $S_s^\bot$ the set of all projections of tables in D and views in V that contains S, i.e., $S_s^\bot := \{R \in extVocabulary(D, V) \mid S(\overline{x}) \subseteq_M R(\overline{y})\}$. Furthermore, we denote by $AUTH_{s,u}$ the set of all tables and views that u is authorized to read in s, i.e., $AUTH_{s,u} := \{o \mid \langle \oplus, \texttt{SELECT}, o\rangle \in permissions(s, u)\}$, and by $AUTH^*_{s,u}$ the set of all the projections obtained from tables and views in $AUTH_{s,u}$.

We are now ready to formally define the $\phi^\top_{s,u}$ and $\phi^\bot_{s,u}$ rewritings.

Definition 6.7. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V, c\rangle$ be an M-state, u be a user, and ϕ be a relational calculus sentence over $extVocabulary(D, V)$.

The function *bound* takes as input a formula ϕ, a state s, a user u, a variable identifier x, and a value v in $\{\top, \bot\}$. It is inductively defined as follows:

- $bound(R(\overline{y}), s, u, x, v)$, where R is a predicate symbol in $extVocabulary(D, V)$, is $\top$ iff (a) x occurs in $\overline{y}$, and (b) the set $R^v_{s,u} := R^v_s \cap AUTH^*_{s,u}$, is not empty.
- $bound(y = z, s, u, x, v)$ is $\top$ iff $x = y$ and z is a constant symbol or $x = z$ and y is a constant symbol.
- $bound(\top, s, u, x, v) := \bot$.
- $bound(\bot, s, u, x, v) := \bot$.
- $bound(\neg\psi, s, u, x, v) := bound(\psi, s, u, x, \neg v)$, where ψ is a relational calculus formula.
- $bound(\psi \wedge \gamma, s, u, x, v) := bound(\psi, s, u, x, v) \vee bound(\gamma, s, u, x, v)$, where ψ and γ are relational calculus formulae.
- $bound(\psi \vee \gamma, s, u, x, v) := bound(\psi, s, u, x, v) \wedge bound(\gamma, s, u, x, v)$, where ψ and γ are relational calculus formulae.
- $bound(\exists y.\, \psi, s, u, x, v)$, where ψ is a relational calculus formula, is $bound(\psi, s, u, x, v) \wedge bound(\psi, s, u, y, v)$ if $x \neq y$, and $bound(\exists y.\, \psi, s, u, x, v) := \bot$ otherwise.
- $bound(\forall y.\, \psi, s, u, x, v)$, where ψ is a relational calculus formula, is $bound(\psi, s, u, x, v) \wedge bound(\psi, s, u, y, v)$ if $x \neq y$, and $bound(\forall y.\, \psi, s, u, x, v) := \bot$ otherwise.

The formula $\phi^\top_{s,u}$ is inductively defined as follows:

- $R(\overline{x})^\top_{s,u} := \bigvee_{S \in R^\top_{s,u}} S(\overline{x})$, where R is a predicate symbol in $extVocabulary(D, V)$ and $R^\top_{s,u} := R^\top_s \cap AUTH^*_{s,u}$.
- $(x = v)^\top_{s,u} := (x = v)$, where x and v are either variable identifiers or constant symbols.
- $(\top)^\top_{s,u} := \top$.
- $(\bot)^\top_{s,u} := \bot$.
- $(\neg\psi)^\top_{s,u} := \neg\psi^\bot_{s,u}$, where ψ is a relational calculus formula.
- $(\psi \wedge \gamma)^\top_{s,u} := \psi^\top_{s,u} \wedge \gamma^\top_{s,u}$, where ψ and γ are relational calculus formulae.
- $(\psi \vee \gamma)^\top_{s,u} := \psi^\top_{s,u} \vee \gamma^\top_{s,u}$, where ψ and γ are relational calculus formulae.
- $(\exists x.\, \psi)^\top_{s,u}$, where ψ is a relational calculus formula and x is a variable identifier, is $\exists x.\, \psi^\top_{s,u}$ if $bound(\psi, s, u, x, \top) = \top$ and $(\exists x.\, \psi)^\top_{s,u} := \bot$ otherwise.
- $(\forall x.\, \psi)^\top_{s,u}$, where ψ is a relational calculus formula and x is a variable identifier, is $\forall x.\, \psi^\top_{s,u}$ if $bound(\psi, s, u, x, \top) = \top$ and $(\forall x.\, \psi)^\top_{s,u} := \bot$ otherwise.

The formula $\phi^\bot_{s,u}$ is inductively defined as follows:

- $R(\overline{x})^\bot_{s,u} := \bigwedge_{S \in R^\bot_{s,u}} S(\overline{x})$, where R is a predicate symbol in $extVocabulary(D, V)$ and $R^\bot_{s,u} := R^\bot_s \cap AUTH^*_{s,u}$.
- $(x = v)^\bot_{s,u} := (x = v)$, where x and v are either variable identifiers or constant symbols.
- $(\top)^\bot_{s,u} := \top$.
- $(\bot)^\bot_{s,u} := \bot$.
- $(\neg\psi)^\bot_{s,u} := \neg\psi^\top_{s,u}$, where ψ is a relational calculus formula.
- $(\psi \wedge \gamma)^\bot_{s,u} := \psi^\bot_{s,u} \wedge \gamma^\bot_{s,u}$, where ψ and γ are relational calculus formulae.
- $(\psi \vee \gamma)^\bot_{s,u} := \psi^\bot_{s,u} \vee \gamma^\bot_{s,u}$, where ψ and γ are relational calculus formulae.
- $(\exists x.\, \psi)^\bot_{s,u}$, where ψ is a relational calculus formula and x is a variable identifier, is $\exists x.\, \psi^\bot_{s,u}$ if $bound(\psi, s, u, x, \bot) = \top$ and $(\exists x.\, \psi)^\bot_{s,u} := \top$ otherwise.
- $(\forall x.\, \psi)^\bot_{s,u}$, where ψ is a relational calculus formula and x is a variable identifier, is $\forall x.\, \psi^\bot_{s,u}$ if $bound(\psi, s, u, x, \bot) = \top$ and $(\forall x.\, \psi)^\bot_{s,u} := \top$ otherwise. □

Finally, we define the formula $\phi^{rw}_{s,u}$ which represents the overall rewritten formula.

Definition 6.8. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V, c\rangle$ be an M-state, u be a user, and ϕ be a relational calculus sentence over D. The formula $\phi^{rw}_{s,u}$ is defined as $inline_{V,D}(\neg\psi^\top_{s,u} \wedge \psi^\bot_{s,u})$, where $\psi := extVoc_{V,D}(\phi)$. □

[2] With a slight abuse of notation, we write $R(\overline{x}) \subseteq_M S(\overline{y})$ instead of $inline_{D,V}(R(\overline{x})) \subseteq_M inline_{D,V}(S(\overline{y}))$.

6.8.4 Enforcement Mechanism

Here, we model the PDP f, presented in Section 6.5, which is obtained by composing the PDPs f_{int} and f^u_{conf} presented above. The PDP f is obtained by composing f_{int} and f^u_{conf} as follows:

$$f(s, act) = f_{int}(s, act) \wedge f_{conf}^{user(act,s)}(s, act)$$

We recall that the function *user* takes as input an action and a state and returns the actual user executing the action. It is defined as follows, where i denotes the *invoker* function and tr denotes the *trigger* function.

$$user(act, s) = \begin{cases} invoker(s) & \text{if } trigger(s) \neq \epsilon \\ u & \text{if } trigger(s) = \epsilon \text{ and } act \in \mathcal{A}_{D,u} \end{cases}$$

Chapter 7

Reconciling Database Access Control and Information-flow Control

Access control formulations suffer from a number of difficulties. First, because they are described in terms of a mechanism for enforcing security, they provide no guidance in circumstances where those mechanisms prove inadequate. Second, it is easy to construct perverse interpretations of access control policies that satisfy the letter, but not the intent of the policy, to the point of being obviously insecure.

John Rushby – Noninterference, transitivity, and channel-control security policies

Database access control (DBAC) and information-flow control (IFC) share the same goal: protecting the confidentiality and integrity of sensitive information. However, they have different foundations, security notions, and enforcement mechanisms. In this chapter, we reconcile these two, seemingly disparate, areas by developing a framework for reasoning about the security of database-backed programs. First, we reduce database access control to determining whether programs leak information, thereby providing a way of applying IFC techniques to database access control. Second, we develop a security monitor based on a novel combination of information-flow tracking with concepts from database theory like disclosure lattices and query determinacy.

Structure. We overview the relationships between DBAC and IFC in Section 7.1. In Section 7.2, we introduce our setting. We present WhileSql in Section 7.3 and our security model in Section 7.4. In Section 7.5, we present our reduction from DBAC to IFC, whereas we present our IFC mechanism for securing database-backed applications in Section 7.6. We discuss related work in Section 7.7 and draw conclusions in Section 7.8. We present some technical details in Section 7.9. The proofs of all our results are given in Appendix D.

7.1 Overview

Database access control and information-flow control have a similar objective: to protect the confidentiality and integrity of sensitive information. Yet they typically do so for different components of a system: databases and applications, respectively. Furthermore, the enforcement techniques used in these two settings are different. IFC employs techniques ranging from security type systems [135, 163] to multi-execution [64], whereas DBAC mechanisms mostly rely on query rewriting [110, 131, 150, 165].

Unfortunately, these mechanisms do not compose naturally. Even if both DBAC and IFC are in place, application-level information, such as a sensitive context of a function call that executes a query, is lost at the time of DBAC enforcement. Conversely, database-level information, such as fine-grained table-level security labels, is lost at the time of IFC enforcement, when information from the database is manipulated by the application. This poses a challenge of tackling insecurities arising at component boundaries.

More fundamentally, DBAC and IFC build on different foundations. DBAC's foundations lie in database theory and access control, whereas IFC builds on programming language theory. Hence, the security properties studied in these two settings are quite different.

Despite their differences, IFC and DBAC are starting to overlap in their scopes. Various IFC approaches [24, 49, 50, 57, 59, 111, 143, 167] have taken databases into account. These approaches extend IFC foundations with database models and apply standard IFC techniques, such as security type systems [57, 141], symbolic execution [49] or faceted values [167], to track how information flows through databases with the goal of providing end-to-end security guarantees. In contrast, DBAC

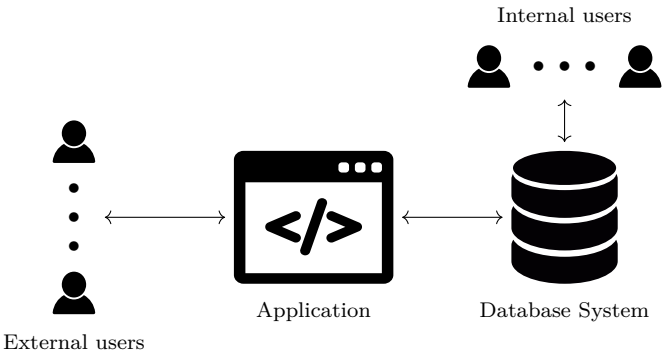


FIGURE 7.1: System model.

mechanisms for databases that support features like triggers or stored procedures [140, 149] face problems like preventing *implicit flows*, which are typical of the IFC setting.

Surprisingly, the relationship between DBAC and IFC remains largely unexplored. This gap hinders the development of both lines of research and limits the integration and re-usability of existing results and mechanisms. For instance, IFC solutions that support databases [57, 59, 141, 167] consider only simplified database models that ignore advanced features, which have been studied in DBAC [140, 149] and can be exploited by attackers. In contrast, DBAC mechanisms could benefit from existing research in IFC.

7.2 Problem Setting

This section presents the system and attacker models associated with DBAC and IFC. We motivate these models with concrete examples and discuss their security requirements.

7.2.1 System and Attacker Model

Figure 7.1 depicts our system model. The system consists of users that interact with the database either directly, by issuing SQL commands, or indirectly, by interacting with programs (e.g., web applications) that, in turn, issue commands to the database. We refer to users directly interacting with the database as *internal users*, and we call *external users* those whose interaction with the database is mediated by a program. For simplicity, we assume that each internal and external user is uniquely associated with a user account that is used to retrieve information from the database. Furthermore, we assume that external users execute all their programs using their own user accounts. Users execute commands concurrently, and a scheduler mediates the interaction with the database system.

In our system model, the security policy is defined at the database level using access control policies, which specify the access permissions of each user account for the database tables and views. The database system has a distinguished user, the administrator, who defines the database schema, the database's integrity constraints, and the database's initial access control policy. We assume that the users' commands, the programs, the database schema, and the database configuration, which consists of the security policy, the views, and the triggers in the system, are publicly known.

To accommodate both DBAC and IFC in the same system model, we consider internal and external attackers. An *internal attacker* is a database user, different from the administrator, who directly issues SQL commands to the database and observes the results of these commands. Namely, an internal attacker is one of the internal users in Figure 7.1. His goal is learning information about the tables and views he cannot read. This reflects the usual attacker model adopted in DBAC (see, for instance, Chapter 6). We call a sequence of commands secure with respect to an internal attacker if the attacker cannot learn sensitive information by observing the results of his own commands.

We remark that our system model does not include an access control system that enforces the security policy. This allows us to study the security condition, i.e., what it means, semantically, for a sequence of commands to be secure, without focusing on the enforcement mechanism, namely an algorithm that only authorizes commands that satisfy the security condition. We sometimes refer to sequences of commands as programs.

An *external attacker* is an external user that cannot directly issue commands to the database, since communication between the attacker and the database is mediated by a program. An external attacker, therefore, can only observe the program's output. He cannot observe the results of the

queries issued by the program unless these results are part of the program's output. Securing applications with respect to this attacker model involves tracking the propagation of data across the application and database, thus enforcing the security policy in an end-to-end fashion.

The external attacker corresponds to the information-flow setting. As is standard, we say that a program is secure with respect to an external attacker iff executing the program does not leak sensitive information to the attacker. For simplicity, we ignore application-level policies for external users. These can easily be modeled using database tables with appropriate security policies.

7.2.2 Overview of Security Conditions

By using examples, we illustrate the intuition behind our security conditions, which we formalize in Section 7.4.

Internal Attackers. Consider a movie rental database where the customer `Bob` pays for a subscription to an online media provider like Netflix. `Bob` is allowed to watch any movie in the table `MOVIE` if he has a valid subscription. For simplicity, we model the valid subscription as a read permission on `MOVIE`. When the subscription expires (and the policy changes), `Bob` should no longer be able to access the movie, even if he has previously streamed a copy of it.

Suppose the database administrator can modify the access control policy of the table `MOVIE`, and `Bob` queries the database to watch the movie. The following program should be considered insecure as `Bob` accesses the table `MOVIE` after the subscription has expired (we write $\texttt{u} : q$ to denote that the user `u` executes the command q).

$$\texttt{admin} : \texttt{GRANT SELECT ON MOVIE TO Bob} \quad (1)$$
$$\texttt{Bob} : \texttt{SELECT} * \texttt{FROM MOVIE} \quad (2)$$
$$\texttt{admin} : \texttt{REVOKE SELECT ON MOVIE FROM Bob} \quad (3)$$
$$\texttt{Bob} : \texttt{SELECT} * \texttt{FROM MOVIE} \quad (4)$$

The building blocks for our security condition for internal attackers are the *attacker knowledge*, the *security policy*, and the notion of an *epoch*. The attacker knowledge describes the set of database states consistent with the attacker's observations, e.g., the results of database queries issued by the attacker. The security policy determines the attacker's initial knowledge, namely the set of databases the attacker initially considers as possible. An epoch represents a portion of the program's execution. Our security condition for internal attackers ensures that the attacker's knowledge remains *constant* inside each epoch, thus satisfying the current security policy.

In our example, epochs are determined by the commands executed by `Bob`. In particular, the above program involves two epochs corresponding respectively to the commands (2) and (4). In the first epoch, `Bob` is authorized to read the table `MOVIE`. Hence, his initial knowledge consists of the content of the table `MOVIE`. Therefore, after the `SELECT` command, `Bob`'s knowledge does not change as he already knows the content of table `MOVIE`. In the second epoch, our condition *resets* `Bob`'s initial knowledge and *forgets* any information that `Bob` may have learned in the past. Now, `Bob` is not authorized to read the table `MOVIE`. Hence, he considers as possible any value for the table `MOVIE`. However, after the `SELECT` command, `Bob` learns the content of `MOVIE`, thus refining his knowledge and violating the current policy. Our security condition therefore considers the program as insecure.

Our security condition for internal attackers characterizes security with respect to the database state at the beginning of each epoch. This is in line with current DBAC requirements [131, 165] that interpret the security policy with respect to the *current state* of the database.

External Attackers. A classical example of an external attacker is a user interacting with a server-side application that communicates with a database through programming language constructs. The goal, here, is to track information across application-database boundaries, ensuring that the user does not learn sensitive information about the initial state of the database. In contrast to internal attackers, an external attacker can only learn information through output statements and he cannot observe the results of the queries. Moreover, the attacker has perfect recall in the sense that he remembers all prior observations.

Consider a web application allowing teaching assistants (TAs) and instructors to grade students' assignments. The application consists of a server-side application and a database system storing the grades. The application performs a `SELECT` query to access the students' grades and then outputs them to a TA.

$$x := \texttt{SELECT} * \texttt{FROM GRADE}$$
$$\textbf{out}(\texttt{TA}, x)$$

Basic Types

(Table Ids)	$T \in \mathbb{T}$	*(Variables)*	$x \in \mathit{Vars}$	*(Trigger Ids)*	$tr \in \mathbb{TR}$
(View Ids)	$V \in \mathbb{V}$	*(Values)*	$n \in \mathit{Vals}$	*(Formulae)*	$\varphi \in \mathit{RC}$
(Relation Ids)	$R \in \mathbb{T} \cup \mathbb{V}$	*(User identifiers)*	$u \in \mathit{UID}$	*(Error Messages)*	$em \in \mathit{EM}$

Syntax

(User Context)	uc	$:=$	`OWNER` | `INVOKER`
(Privileges)	p	$:=$	`SELECT ON` R | `INSERT ON` T | `DELETE ON` T | `CREATE VIEW` | `CREATE TRIGGER ON` T
(Actions)	a	$:=$	`INSERT` $e_1, \ldots, e_n$ `INTO` T | `DELETE` $e_1, \ldots, e_n$ `FROM` T
			| `GRANT` p `TO` u | `GRANT` p `TO` u `WITH GRANT OPTION` | `REVOKE` p `FROM` u
(SQL commands)	q	$:=$	a | `SELECT` φ | `ADD USER` u | `CREATE VIEW` V `:` `SELECT` φ `AS` uc
			| `CREATE TRIGGER` tr `ON` T `AFTER (INS|DEL) IF` φ `DO` a `AS` uc
(Expressions)	e	$:=$	$n \mid x \mid \neg e \mid e_1 \oplus e_2$
(Statements)	c	$:=$	$\varepsilon \mid x \leftarrow q \mid x := e \mid \mathbf{out}(u, e) \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid c_1\ ;\ c_2$

FIGURE 7.2: WHILESQL's syntax.

Assume now that while executing this application, the administrator changes the policy by revoking the read permission on `GRADE`.

```
GRANT SELECT ON GRADE TO TA
REVOKE SELECT ON GRADE FROM TA
```

Finally, assume that the interleaving produced by the executions of the two programs is as follows. The policy initially allows the TA to learn the grades. Then, the application retrieves the students' grades using the `SELECT` query, and, immediately after that, the administrator executes the `REVOKE` command, thereby disallowing the TA to learn the grades. The grades are then sent to the TA through an output command. Although the query was performed at a time when it was allowed by the policy, the TA learns the grade when this is no longer allowed. Hence, this program should be considered insecure.

We propose a security condition ensuring that information release is always done in accordance with the current security policy. Since information release happens through output statements, a program is considered secure if whenever the attacker learns some information, then (i) the attacker already learned that information in the past when this was allowed by the policy, or (ii) the current security policy allows the attacker to learn that information. In this scenario, the TA learns the grade when this is no longer allowed by the policy and, since he did not learn the grade in the past, the program is considered insecure. In contrast to internal attackers, the security condition neither resets the attacker's knowledge nor forgets any information that the attacker may have learned in the past. Furthermore, as is common in the IFC setting [24,49,56,143,167], we interpret the security policy with respect to the *initial state* of the database.

7.3 WhileSql

WHILESQL is a simple language that captures the main features of both programming languages extended with querying constructs and procedural extensions of the SQL standard, such as Oracle's PL/SQL or Microsoft TRANSACT-SQL. At the same time, it simplifies some subtle aspects of their semantics, while still capturing the main security-critical features.

7.3.1 Syntax

Figure 7.2 depicts WHILESQL's syntax. Let $\mathbb{T}$, $\mathbb{V}$, and $\mathbb{TR}$ be three countably infinite sets representing table identifiers, view identifiers, and trigger identifiers. Furthermore, let *Vars* and *Vals* be countably infinite sets of variables and values. We assume that all these sets are pairwise disjoint.

As shown in Figure 7.2, a WHILESQL program is an imperative program extended with querying capabilities, i.e., statements of the form $x \leftarrow q$. A statement $x \leftarrow q$ executes the SQL command q and assigns its result to the variable x. WHILESQL supports SQL's core features, such as `SELECT`, `INSERT`, `DELETE`, `GRANT`, and `REVOKE` commands, as well as advanced database features such as triggers and views. Additionally, WHILESQL programs support assignments and standard control flow statements. WHILESQL also supports $\mathbf{out}(u, e)$ statements to output the value of the expression e to the user u. For simplicity, we assume that all expressions are well-typed and all SQL statements refer either to tables in the database schema or to previously created views.

WHILESQL builds on top of the database operational semantics from Chapter 5. Hence, it supports the same fragment of SQL supported by the database semantics. We now recall the restrictions and simplifications inherited from our operational semantics. For `SELECT` commands, instead of using

SQL, we rely on the relational calculus. Moreover, we support only `INSERT` and `DELETE` commands that explicitly identify the tuple to be inserted or deleted. More complex `INSERT` and `DELETE` commands, as well as `UPDATE` commands, can be simulated by combining `SELECT` queries with `INSERT` and `DELETE` commands. Finally, we support only triggers that are executed in response to `INSERT` and `DELETE` commands. We assume that a trigger's body has the form `IF` φ `DO` a `AS` uc, where φ is a boolean query, a is an `INSERT`, `DELETE`, `GRANT`, or `REVOKE` command, and uc specifies whether the trigger is executed under the owner's or the activator's privileges.

Each SQL command either returns the query result or an error message $em \in EM$. Error messages indicate whether queries (or triggers) violate security constraints, like a query that is not allowed by the current security policy, or integrity constraints, such as an `INSERT` statement that violates a primary key constraint. Note that error messages are values, i.e., $EM \subseteq Vals$.

7.3.2 Local Semantics

Here, we define the semantics of a WhileSql program executed in isolation. A WhileSql program is defined with respect to a *system configuration* $M = \langle D, \Gamma \rangle$, where $D = \langle \Sigma, \mathbf{dom} \rangle$ is a database schema and Γ is a set of integrity constraints. We assume that $\mathbf{dom} \subseteq Vals$ and that only values in $\mathbf{dom}$ are used to construct queries. For simplicity, we fix a configuration $M = \langle D, \Gamma \rangle$ for the rest of the section.

Databases. WhileSql reuses the database model and the notion of system and runtime states from Chapter 5. Here, we recall only the main concepts that we use in the rest of the chapter. We refer the reader to Chapter 5 for more details on security policies and our database model.

A *security policy* is a finite set of `GRANT` statements. Given a policy sec and a user u, we denote by $auth(sec, u)$ the set of all tables and views with the owner's privileges that u is authorized to read according to the `GRANT` statements in sec. A *system state* is a tuple $\langle db, U, sec, T, V \rangle$ where db is a database state, $U \subset UID$ is a finite set of users, T is a finite set of triggers, V is a finite set of views, and sec is a security policy. Note that we lift $auth$ from policies to system states, i.e., $auth(\langle db, U, sec, T, V \rangle, u) = auth(sec, u)$. A *context* ctx describes the database's history, the scheduled triggers that must be executed, and how to modify the database's state in case a roll-back occurs. A *runtime state* is a tuple $\langle s, ctx \rangle$ where s is a system state and ctx is a context. The set of all runtime states is denoted by Ω_M and we denote by ϵ the empty context. In the following, we use s to refer to both system states and runtime states whenever this is clear from the context, and we use the notation $\langle s, ctx \rangle$ otherwise.

Memories and Configurations. A *memory* $m \in Mem$ is a partial function mapping variables to values, i.e. $Mem : Vars \to Vals$. A *local configuration* $\langle c, m, \langle s, ctx \rangle \rangle$ consists of a command $c \in Com$, a memory $m \in Mem$, and a runtime state $\langle s, ctx \rangle \in \Omega_M$. A local configuration is *initial* iff its context ctx is ϵ and the database state s is an initial database state as defined in Chapter 5. We denote by $Conf$ the set of all configurations.

Users. Let UID be a countably infinite set representing all user identifiers. In addition to users in UID, we add a designated user *public* that can observe only public events (i.e., changes to the database configuration) and for each user $u \in UID$, we add a user $db(u)$ that represents the internal user corresponding to u. Hence, the set $\mathcal{U}$ of all users is $UID \cup \{db(u) \mid u \in UID\} \cup \{public\}$. We define a partial order $\preceq_{\mathcal{U}}$ over $\mathcal{U}$ by $u_1 \preceq_{\mathcal{U}} u_2$ iff $u_1 = public$, $u_1 = u_2$, or $u_2 = db(u_1)$.

Observations. We model program-level and database-level observations. The former are generated using **out** statements while the latter are generated by queries. A program-level observation is a pair $\langle u, o \rangle$, where $u \in UID$ is a user identifier and $o \in Vals$ is a value. A database-level observation, instead, is a 4-tuple $\langle u, q, o, \tau \rangle$, where $u \in \{db(u) \mid u \in UID\} \cup \{public\}$, q is either an SQL command or a trigger, o is either a value or an SQL command, and τ is a (possibly empty) sequence of database-level observations. We use τ to represent the observations caused by triggers executed in response to an SQL command. We write $\langle u, q, o \rangle$ instead of $\langle u, q, o, \tau \rangle$ if $\tau = \epsilon$. We denote by Obs the set of all observations.

Semantics. Given a user $u \in UID$, the relation $\to_u \subseteq (Com \times Mem \times \Omega_M) \times Obs \times (Com \times Mem \times \Omega_M)$ formalizes the small-step local operational semantics of WhileSql programs executed by u. A *run* r is an alternating sequence of configurations and observations that starts with an initial configuration and respects the rules defining $\to_u$. Given a run r, we denote by r^i, where $i \in \mathbb{N}$, the run obtained by truncating r at the i-th state. A *trace* is an element of Obs^*. The trace τ associated to a run r, denoted by $trace(r)$, is obtained by concatenating all observations in the run.

As noted above, the operational semantics of SQL statements relies on the operational semantics given in Chapter 5. We use the function $[\![q]\!](\langle s, ctx \rangle, u)$ (defined in Section 7.9.1) to connect WhileSql's operational semantics with the database operational semantics from Chapter 5. The function $[\![q]\!](\langle s, ctx \rangle, u)$ takes as input an SQL statement q, a runtime state $\langle s, ctx \rangle \in \Omega_M$, and the user $u \in UID$ executing the command, and it returns a tuple $\langle \langle s, ctx \rangle', r, em, \tau \rangle$, where $\langle s', ctx' \rangle \in \Omega_M$

$$\textsc{E-Skip}\quad \frac{}{\langle \mathbf{skip}, m, s\rangle \to_u \langle \varepsilon, m, s\rangle}$$

$$\textsc{E-Assign}\quad \frac{}{\langle x := e, m, s\rangle \to_u \langle \varepsilon, m[x \mapsto [\![e]\!](m)], s\rangle}$$

$$\textsc{E-QueryOk}\quad \frac{\begin{array}{c}\{v_1,\ldots,v_n\} = vars(q)\\ q' = q[v_1 \mapsto [\![v_1]\!](m),\ldots,v_n \mapsto [\![v_n]\!](m)]\\ [\![q']\!](s,u) = \langle s', r, \epsilon, \tau\rangle\\ u' = Usr(u,q)\end{array}}{\langle x \leftarrow q, m, s\rangle \xrightarrow{\langle u', q', r, \tau\rangle}_u \langle \varepsilon, m[x \mapsto r], s'\rangle}$$

$$\textsc{E-QueryEx}\quad \frac{\begin{array}{c}\{v_1,\ldots,v_n\} = vars(q)\\ q' = q[v_1 \mapsto [\![v_1]\!](m),\ldots,v_n \mapsto [\![v_n]\!](m)]\\ [\![q']\!](s,u) = \langle s', r, em, \tau\rangle\\ em \neq \epsilon\end{array}}{\langle x \leftarrow q, m, s\rangle \xrightarrow{\langle db(u), q', em, \tau\rangle}_u \langle \varepsilon, m[x \mapsto em], s'\rangle}$$

$$\textsc{E-Out}\quad \frac{}{\langle \mathbf{out}(u', e), m, s\rangle \xrightarrow{\langle u', [\![e]\!](m)\rangle}_u \langle \varepsilon, m, s\rangle}$$

$$\textsc{E-IfTrue}\quad \frac{[\![e]\!](m) = \mathbf{tt}}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, m, s\rangle \to_u \langle c_1, m, s\rangle}$$

$$\textsc{E-IfFalse}\quad \frac{[\![e]\!](m) = \mathbf{ff}}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, m, s\rangle \to_u \langle c_2, m, s\rangle}$$

$$\textsc{E-WhileTrue}\quad \frac{[\![e]\!](m) = \mathbf{tt}}{\langle \mathbf{while}\ e\ \mathbf{do}\ c, m, s\rangle \to_u \langle c\ ;\ \mathbf{while}\ e\ \mathbf{do}\ c, m, s\rangle}$$

$$\textsc{E-WhileFalse}\quad \frac{[\![e]\!](m) = \mathbf{ff}}{\langle \mathbf{while}\ e\ \mathbf{do}\ c, m, s\rangle \to_u \langle \varepsilon, m, s\rangle}$$

$$\textsc{E-Seq}\quad \frac{\langle c_1, m, s\rangle \xrightarrow{\tau}_u \langle c_1', m', s'\rangle}{\langle c_1\ ;\ c_2, m, s\rangle \xrightarrow{\tau}_u \langle c_1'\ ;\ c_2, m', s'\rangle}$$

$$\textsc{E-SeqEmpty}\quad \frac{}{\langle \varepsilon\ ;\ c, m, s\rangle \to_u \langle c, m, s\rangle}$$

Figure 7.3: WhileSql's local operational semantics.

is the new runtime state, r is q's result, em is an error message, and τ is a trace of database-level observations produced by triggers executed in response to q. We also write $[\![e]\!](m)$ to denote the evaluation of an expression e in memory m. It is always clear from context if $[\![\cdot]\!](\cdot)$ refers to query or expression evaluation.

Figure 7.3 depicts the rules defining WhileSql's local semantics. Most of the rules are standard. The only non-standard rules are E-QueryOk and E-QueryEx, which regulate the execution of SQL commands. The rule E-QueryOk models the successful execution of SQL commands. It first replaces the free variables in the query with their actual values. Afterwards, it executes the query, it produces the database-level observation associated with the query, and it stores the query result in the memory. The rule relies on the function $Usr(u, q)$, which takes as input a user $u \in UID$ and a query and returns the user $db(u)$ if q is a `SELECT`, `INSERT`, or `DELETE` command, and *public* otherwise (since queries that modify the database's configuration produce observations that are visible to all users). The rule E-QueryEx, instead, models the failed execution of a query. The rule executes the query, retrieves the error message, and stores it in the memory.

7.3.3 Global Semantics

To model realistic scenarios, where attackers and honest users each run their own programs which may access a common database, we assume that programs do not share memory, whereas the database is shared. We now present a global semantics capturing the parallel execution of WhileSql programs.

Schedulers. We model a *scheduler* as an infinite sequence of natural numbers $\mathcal{S} \in \mathbb{N}^\omega$. In the global semantics, we use the scheduler to determine which program has to be executed at each point in the execution.

Global Configurations. We denote the set of commands together with the executing user by $Com_{UID} = UID \times Com$ and the set of pairs of users and memories as $Mem_{UID} = UID \times Mem$. To model a system state where multiple WhileSql programs run in parallel and share a common database, we introduce global configurations. A *global configuration* is a tuple $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \in GlConf$, where $C \in Com^*_{UID}$ is a sequence of WhileSql programs with the executing users, $M \in Mem^*_{UID}$ is a sequence of memories (one per program in C), $\langle s, ctx\rangle \in \Omega_M$ is the runtime state of the shared database, and $\mathcal{S}$ is a scheduler formalizing the interleaving of the programs in C. We consider only configurations $\langle C, M, s, \mathcal{S}\rangle$ such that $|C| = |M|$ and for all $1 \leq i \leq |C|$, $C(i) = \langle u, c\rangle$, $M(i) = \langle u', m\rangle$, and $u = u'$. Furthermore, a *global state* is a pair $\langle M, s\rangle$, where $M \in Mem^*_{UID}$ and s is a system state.

Semantics. The relation $\to \subseteq GlConf \times Obs \times GlConf$, shown in Figure 7.4, formalizes the global operational semantics of a database system that runs multiple WhileSql programs in parallel.

M-Eval-Step
$$\frac{\forall i \in \{1,\ldots,|C|\}, u' \in \mathit{UID}.\ C(i) \neq \langle u', \varepsilon\rangle \quad n = 1 + (n' \bmod |C|) \quad C(n) = \langle u, c\rangle \quad M(n) = \langle u, m\rangle \quad |C| = |M| \quad \langle c, m, s\rangle \xrightarrow{\tau}_u \langle c', m', s'\rangle \quad C' = C(1)\cdot\ldots\cdot C(n-1)\cdot\langle u, c'\rangle\cdot C(n+1)\cdot\ldots\cdot C(|C|) \quad M' = M(1)\cdot\ldots\cdot M(n-1)\cdot\langle u, m'\rangle\cdot M(n+1)\cdot\ldots\cdot M(|C|)}{\langle C, M, s, n'\cdot\mathcal{S}\rangle \xrightarrow{\tau} \langle C', M', s', \mathcal{S}\rangle}$$

M-Eval-End
$$\frac{1 \leq n \leq |C| \quad \forall n' < n, u' \in \mathit{UID}.\ C(n') \neq \langle u', \varepsilon\rangle \quad C(n) = \langle u, \varepsilon\rangle \quad |C| = |M| \quad C' = C(1)\cdot\ldots\cdot C(n-1)\cdot C(n+1)\cdot\ldots\cdot C(|C|) \quad M' = M(1)\cdot\ldots\cdot M(n-1)\cdot M(n+1)\cdot\ldots\cdot M(|C|)}{\langle C, M, s, \mathcal{S}\rangle \xrightarrow{\tau} \langle C', M', s, \mathcal{S}\rangle}$$

Figure 7.4: WhileSql's global operational semantics

Given a global configuration $\langle C, M, s, \mathcal{S}\rangle$, the global operational semantics uses the scheduler $\mathcal{S}$ to select which of the programs in C to execute. This is done by extracting the first number n' from the scheduler $\mathcal{S}$ and identifying the associated program $\langle u, c\rangle$ and memory $\langle u, m\rangle$ in C and M respectively. The rule M-Eval-Step identifies the WhileSql program that should be executed according to the scheduler, it executes one step of the local semantics, and it updates the global state accordingly. The rule M-Eval-End, instead, removes the terminated programs from the global configuration. Given a run r, we denote by $\mathit{conf}(r)$ the global configuration in the last state in the run. Furthermore, $\mathit{trace}(r)$ denotes the trace associated with the run r, and $\mathit{db}(r)$ denotes the database state in the global configuration $\mathit{conf}(r)$.

7.4 Security Model

Here we formalize the security conditions for internal and external attackers as presented in Section 7.2. For consistency, we define our security conditions over general WhileSql programs, noting that internal attackers can execute only programs consisting of sequences of queries. The security condition for internal attackers characterizes typical database users that may execute SQL commands and learn information from the database iff the current database policy authorizes them to do so. In contrast, the security condition for external attackers adopts an end-to-end interpretation of security policies that accounts for both the database and the application. In particular, if the program issues a query when authorized by the policy but subsequently outputs the result of that query when this is forbidden by the policy, our condition rejects the program as insecure.

7.4.1 Preliminaries

Before presenting our security conditions, we introduce some notation.

Equivalence of memories. Given a user u, two sequences of memories $M_1, M_2 \in \mathit{Mem}^*_{\mathit{UID}}$ are *u-equivalent*, written $M_1 \approx_u M_2$, iff $|M_1| = |M_2|$ and for all $1 \leq i \leq |M_1|$, $u_1^i = u_2^i$ and if $u_1^i = u$, then $m_1^i = m_2^i$, where $M_1(i) = \langle u_1^i, m_1^i\rangle$ and $M_2(i) = \langle u_2^i, m_2^i\rangle$.

Equivalence of databases. Given two database states db and db' and a set S of tables and views, we say that db and db' are S-equivalent, written $\mathit{db} \approx_S \mathit{db}'$, iff the content of all tables and views (with owner's privileges) in S is the same in db and db'. For the indistinguishability notions between system states, we reuse data-indistinguishability from Chapter 6. Given a user u, two system states $s = \langle \mathit{db}, U, \mathit{sec}, T, V\rangle$ and $s' = \langle \mathit{db}', U', \mathit{sec}', T', V'\rangle$ are *u-equivalent*, written $s \approx_u s'$, iff (1) $U = U'$, (2) $\mathit{sec} = \mathit{sec}'$, (3) $T = T'$, (4) $V = V'$, and (5) $\mathit{db} \approx_{\mathit{auth}(\mathit{sec},u)} \mathit{db}'$.

Equivalence of global states. Two global states $\langle M, s\rangle$ and $\langle M', s'\rangle$ are *u-equivalent* for a user $u \in \mathit{UID}$, written $\langle M, s\rangle \approx_u \langle M', s'\rangle$, iff $M \approx_u M'$ and $s \approx_u s'$. Furthermore, we write $\langle M, s\rangle \approx_{\mathit{db}(u)} \langle M', s'\rangle$ iff $\langle M, s\rangle \approx_u \langle M', s'\rangle$. We denote by $[\langle M, s\rangle]_{\approx_u}$ the set of all global states $\langle M', s'\rangle$ such that $\langle M', s'\rangle \approx_u \langle M, s\rangle$.

Equivalence of traces. The projection of a trace τ for a user u, written $\tau\restriction_u$, is as follows: $\epsilon\restriction_u = \epsilon$, $(\langle u', o\rangle\cdot\tau')\restriction_u = \langle u', o\rangle\cdot\tau'\restriction_u$ if $u' \preceq_{\mathcal{U}} u$, $(\langle u', q, o, \tau''\rangle\cdot\tau')\restriction_u = \langle u', q, o, \tau''\restriction_u\rangle\cdot\tau'\restriction_u$ if $u' \preceq_{\mathcal{U}} u$ or there is a $u'' \in \mathit{users}(\tau'')$ such that $u'' \preceq_{\mathcal{U}} u$, and $(\langle u', o\rangle\cdot\tau')\restriction_u = (\langle u', q, o, \tau''\rangle\cdot\tau')\restriction_u = \tau'\restriction_u$ otherwise, where $\mathit{users}(\tau)$ is the set of all users appearing in the observations in τ. Finally, we say that two traces τ_1 and τ_2 are *u-progress-insensitive-equivalent*, for a user u, written $\tau_1 \sim_u \tau_2$, iff $\tau_1\restriction_u \preceq \tau_2\restriction_u$ or $\tau_2\restriction_u \preceq \tau_1\restriction_u$.

Knowledge. Following [20], we semantically characterize what an attacker can infer from an execution as the set of global states that are consistent with an observed trace.

Definition 7.1. The *knowledge* $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$ *of a user* u for a global state $\langle M_0, s_0\rangle$, a sequence of programs C, a scheduler $\mathcal{S}$, and a trace τ is $\{\langle M, s\rangle \mid s \approx_u s_0 \wedge M \approx_u M_0 \wedge \forall ctx', \tau', C', M', s', \mathcal{S}'.\ (\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle \Rightarrow \tau \sim_u \tau')\}$. □

The knowledge of an attacker u is the set of initial global states that the attacker cannot distinguish based on the observation of the trace $\tau{\restriction_u}$. Thus, a smaller knowledge set indicates that u has a more precise knowledge. The definition of the attacker's knowledge is *progress-insensitive*, that is we ignore information leaks due to the progress of computation, e.g., program divergence and termination [19]. We achieve this by requiring that any execution starting from a u-equivalent global state only produces traces τ' that are progress-insensitive equivalent with the original trace τ.

7.4.2 Internal Attackers

To properly account for internal attackers, we introduce the notion of epoch, which represents a portion of a run. An internal attacker forgets the accumulated knowledge each time a new epoch starts. We use epochs to model the non-permanent release of information as is common in database access control. Intuitively, by forgetting all accumulated knowledge, we prevent any release of information whenever that information is considered sensitive, although the same information might have been non-sensitive in the past or might become non-sensitive in the future.

Epochs. We formalize epochs using an *epoch predicate*, which is a relation over $GlConf \times GlConf$ that, for each pair of (consecutive) global configuration gc_1 and gc_2, specifies whether transitioning from gc_1 to gc_2 starts a new epoch. Given an epoch predicate E and a (terminating) run r, we denote by $[\![E]\!](r)$ the set of epochs defined by E over the run r. Namely, $[\![E]\!](C_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_n} C_{n+1}) = \{C_1 \xrightarrow{\tau}{}^{j} C_{j+1}\} \cup [\![E]\!](C_{j+1} \xrightarrow{\tau_{j+1}} \ldots \xrightarrow{\tau_n} C_{n+1})$, where $1 \leq j \leq n$ is the smallest natural number such that $\langle C_j, C_{j+1}\rangle \in E$ (if such number exists), and $[\![E]\!](C_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_n} C_{n+1}) = \{C_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_n} C_{n+1}\}$ if there is no $j \in \{1, \ldots, n\}$ such that $\langle C_j, C_{j+1}\rangle \in E$. We also introduce selection predicates. These identify a subset of the epochs in a run that satisfies a given condition. Formally, a *selection predicate* is a set of epochs. Given an epoch predicate E, a (terminating) run r, and a selection predicate S, $[\![E, S]\!](r)$ denotes the set $[\![E]\!](r) \cap S$.

To illustrate the concept of epochs, we now introduce epoch and selection predicates capturing the attacker model of Chapter 6, where users do not know the commands executed by other users. We capture this through *user-based epochs*, which represent portions of a run where a user continuously interacted with the system, without interleaved commands executed by other users. Note that this is the notion of epoch used in Section 7.2. We model this using the epoch predicate E^{user}, which splits runs into "user sessions", and the selection predicate S_u^{user}, which selects only u's sessions. Formally, the epoch predicate E^{user} is defined as follows: $\langle\langle C, M, s, n{\cdot}\mathcal{S}\rangle, \langle C', M', s', n'{\cdot}\mathcal{S}'\rangle\rangle \in E^{user}$ iff $C(1 + (n \mod |C|)) = \langle u, c\rangle$, $C'(1 + (n' \mod |C'|)) = \langle u', c'\rangle$, and $u \neq u'$. Additionally, the selection predicate S_u^{user}, for a user u, is defined as follows: $(\langle C_1, M_1, s_1, n_1{\cdot}\mathcal{S}_1\rangle \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_k} \langle C_{k+1}, M_{k+1}, s_{k+1}, n_{k+1}{\cdot}\mathcal{S}_{k+1}\rangle) \in S_u^{user}$ iff for all $1 \leq i \leq k$, there is a command c such that $C_i(1 + (n_i \mod |C_i|)) = \langle u, c\rangle$.

Security Condition. We now define our security condition for internal attackers. Intuitively, this notion corresponds to an attacker that "forgets" all knowledge whenever the epoch changes. Otherwise, an attacker's knowledge remains constant inside each epoch, thus satisfying the security policy at the beginning of the epoch.

Definition 7.2. A sequence of programs $C \in Com_{UID}^{*}$ is *secure with respect to an internal attacker* $u \in UID$ for a scheduler $\mathcal{S}$, a system state s_0, a sequence of memories $M_0 \in Mem_{UID}^{*}$, an epoch predicate E, and a selection predicate S iff whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$, for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{*} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E, S]\!](r)$, then $K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau) = [\langle M_i, s_i\rangle]_{\approx_u}$. □

Example 7.1. Consider (a variant of) the program from Section 7.2 where the commands are scheduled as follows:

$$\texttt{GRANT SELECT ON MOVIE TO Bob} \tag{1}$$
$$x := \texttt{SELECT}\ \exists x.\, \texttt{MOVIE}(x) \tag{2}$$
$$\texttt{REVOKE SELECT ON MOVIE FROM Bob} \tag{3}$$
$$x := \texttt{SELECT}\ \exists x.\, \texttt{MOVIE}(x) \tag{4}$$

Suppose `Bob` is an attacker who wants to learn whether there are any movies in the table `MOVIE`. Abusing notation, we report only `Bob`'s memory and write m instead of $\langle \texttt{Admin}, m_{\texttt{Admin}}\rangle{\cdot}\langle \texttt{Bob}, m\rangle$. Furthermore, we denote by C_i the code at the i-th step in the execution.

Consider an initial local state $\langle m, s\rangle$ such that $m(x) = \bot$ and the table `MOVIE` is empty in s. There are two user-based epochs for the user `Bob`: the former corresponds to the execution of line (2), and the latter corresponds to the execution of line (4). In the first epoch, since the policy allows `Bob` to learn the data in the table `MOVIE`, we have that $K_{db(\texttt{Bob})}(\langle m, s_1\rangle, C_1, \mathcal{S}, \tau) = \{\langle m, s\rangle\}$ where $m(x) = \bot$ and $s_1 = s$. Similarly, $[\langle m, s\rangle]_{\approx_{\texttt{Bob}}} = \{\langle m, s\rangle\}$. The equality holds both before and after the execution of line (2); the execution of the statement in line (2) does not change `Bob`'s knowledge since he can read the table `MOVIE`. In the second epoch, the security policy prevents `Bob` from learning the data in the table `MOVIE`, hence $[\langle m_3, s_3\rangle]_{\approx_{\texttt{Bob}}} = \{\langle m_3, s_3\rangle, \langle m_3, s_3'\rangle, \langle m_3, s_3''\rangle, \ldots\}$, where $s_3, s_3', s_3'', \ldots$ are all possible database states and $s_3 = s$. In contrast, after the `SELECT` statement in line (4), `Bob` learns that the table `MOVIE` is empty. Namely, $K_{db(\texttt{Bob})}(\langle m_3, s_3\rangle, C_3, \mathcal{S}_3, \tau) = \{\langle m_3, s_3\rangle\}$. Since the two sets are different, the program is rejected as insecure. ■

7.4.3 External Attackers

We now formalize our security condition for external attackers. The condition is based on an attacker-centric view of security. It keeps track of the attacker's knowledge during a program execution and it ensures that the knowledge complies with the current security policy. The security condition is inspired by existing conditions for information-flow in the presence of dynamic policies [18, 39].

Allowed knowledge. We interpret security policies with respect to the initial global states. The *allowed knowledge* $A_{u,sec}$ determines the set of initial global states that a user u considers possible given a security policy *sec*. Given a sequence of memories $M_0 \in Mem^*_{UID}$, a system state $s_0 = \langle db_0, U_0, sec_0, T_0, V_0\rangle$, a security policy *sec*, and a user u, we define the set $A_{u,sec}(M_0, s_0)$ as $\{\langle M, s\rangle \mid s \approx_{sec,u} s_0 \wedge M \approx_u M_0\}$, where $\langle db', U', sec', T', V'\rangle \approx_{sec,u} \langle db'', U'', sec'', T'', V''\rangle$ iff $db' \approx_{auth(sec,u)} db''$. We call $A_{u,sec}(M_0, s_0)$ the *allowed* knowledge since it represents the knowledge of the initial global state that the user u is permitted to learn given the policy *sec*.

Security Condition. We now introduce our security condition.

Definition 7.3. A sequence of programs $C \in Com^*_{UID}$ is *secure with respect to an external attacker* $u \in UID$ for a scheduler $\mathcal{S}$, a system state s_0, and a sequence of memories $M_0 \in Mem^*_{UID}$ iff whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \le i \le n$, $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{u,sec}(M_0, s_0) \subseteq K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^i))$, where the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$. □

In a nutshell, the security condition ensures that the attacker's knowledge after observing $trace(r^i)$ is at most as precise as the previous knowledge combined with the allowed knowledge of the execution r^{i-1}, i.e., the knowledge increase resulting from observing $trace(r^i)$ is allowed by the current policy.

Example 7.2. Consider (a variant of) the program from Section 7.2 and a scheduler $\mathcal{S}$ that interleaves the commands as follows.

```
GRANT SELECT ON GRADE TO Student                    (1)
x := SELECT ∃x. GRADE(x)                            (2)
REVOKE SELECT ON GRADE FROM Student                 (3)
out(Student, x)                                     (4)
```

The user `Student` is an attacker who wants to learn whether there are any grades in the table `GRADE`. We now show that the program is insecure with respect to Definition 7.3. Abusing notation, we report only `Student`'s memory, and we write m instead of $\langle \texttt{Admin}, m_{\texttt{Admin}}\rangle \cdot \langle \texttt{Student}, m\rangle$.

Consider an initial memory m_0 such that $m_0(x) = \bot$ and an initial database state s_0 such that the table `GRADE` is empty. We write $K_{\texttt{Student}}(\langle m_0, s_0\rangle, C, \mathcal{S}, \tau_i)$ and $A_{\texttt{Student},sec_i}(m_0, s_0)$ to denote, respectively, the attacker's knowledge and the allowed knowledge after executing the commands up to line (i), where $0 \le i \le 4$. Let r be the program execution. Initially, since the `Student` has made no observations, we have $K_{\texttt{Student}}(\langle m_0, s_0\rangle, C, \mathcal{S}, \tau_0) = \texttt{All}$, where `All` is the set of all possible initial database states. Similarly, assuming that the `Student` is not allowed to access the database initially, we have $A_{\texttt{Student},sec_0}(m_0, s_0) = \texttt{All}$. The attacker's knowledge does not change until the output statement is executed, since the observations produced by the policy changes are independent of s_0. As a result, the inclusion relation from Definition 7.3 holds trivially for the execution r^3, since the allowed knowledge set can only make the left-hand side smaller. In particular, we have $K_{\texttt{Student}}(\langle m_0, s_0\rangle, C, \mathcal{S}, \tau_3) = \texttt{All}$ and $A_{\texttt{Student},sec_3}(m_0, s_0) = \texttt{All}$ due to the `REVOKE` statement in line (3). However, after executing the output statement, the attacker learns that the database was initially empty, i.e., $K_{\texttt{Student}}(\langle m_0, s_0\rangle, C, \mathcal{S}, \tau_4) = \{(m_0, s_0)\}$. Thus, $K_{\texttt{Student}}(\langle m_0, s_0\rangle, C, \mathcal{S}, trace(r^3)) \cap A_{\texttt{Student},sec_3}(m_0, s_0) \not\subseteq K_{\texttt{Student}}(\langle m_0, s_0\rangle, C, \mathcal{S}, trace(r^4))$. Hence, the program is rejected as insecure. ■

7.4.4 Discussion

The security conditions for internal and external attackers are incomparable. Namely, there are programs that are secure for one condition and insecure for the other one. We now illustrate these differences using examples.

Consider a simple variation of Example 7.1, where each SQL statement is immediately followed by an output statement printing the query's result. This program is insecure for internal attackers for the same motivations outlined in Example 7.1. Despite that, the program is secure for external attackers. The second `SELECT` query would not modify the attacker's knowledge, since the attacker learned the content of the table `MOVIE` when this was allowed by the policy.

Consider now a variation of Example 7.2, where the output statement is replaced by the statement **if** x **then** $\{\mathbf{out}(\texttt{Student}, x); y \leftarrow \texttt{INSERT 1 INTO } T\}$ **else** $\{\mathbf{out}(\texttt{Student}, x); y \leftarrow \texttt{INSERT 1 INTO } R\}$. This program is insecure for external attackers for the same motivations outlined in Example 7.2. The program is, however, secure according to our condition for internal attacker (instantiated with user-based epochs), even though the `INSERT` commands allows the internal attacker to learn the value of x. This follows from the fact that the `SELECT` and the `INSERT` commands are in different epochs.

7.5 From Database Access Control to Information-flow Control

The security model in Section 7.4 focuses on information-flow control. Given programs $c_1, \ldots, c_n$ executed by the users $u_1, \ldots, u_n$ and an attacker *atk*, our security conditions can be used to determine whether the parallel execution of $c_1, \ldots, c_n$ leaks information to *atk*. This approach relies on two key assumptions: (1) the users $u_1, \ldots, u_n$ interacting with the system are known beforehand, and (2) the executed programs $c_1, \ldots, c_n$ are fixed and known to all users. Neither of these assumptions, however, hold for database access control, where any user can potentially interact with the system and issue arbitrary commands. As a result, neither the users $u_1, \ldots, u_n$ nor the code of the programs $c_1, \ldots, c_n$, i.e., the queries, are known beforehand. Furthermore, each user may have only a limited knowledge of other users' commands. To overcome these restrictions, we now present a reduction that constructs a database access control mechanism starting from an information-flow enforcement mechanism for WhileSql programs.

7.5.1 Preliminaries

We first introduce some terminology and notation.

Progress-sensitive security. To properly capture existing DBAC security notions, we focus on the progress-sensitive versions of our security conditions. The *progress-sensitive knowledge* $PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$ *of a user* $u \in \mathcal{U}$ for a global state $\langle M_0, s_0\rangle$, programs C, a scheduler $\mathcal{S}$, and a trace τ is $\{\langle M, s\rangle \mid s \approx_u s_0 \wedge M \approx_u M_0 \wedge \exists ctx', \tau', C', M', s', \mathcal{S}'.\ \langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle \wedge \tau\restriction_{db(u)} = \tau'\restriction_{db(u)}\}$. The *progress-sensitive variant* of Definition 7.2 (respectively Definition 7.3) is obtained by replacing the knowledge K_u with progress-sensitive knowledge PK_u.

Enforcement Mechanisms. An *enforcement mechanism* is a function *ifEnf* that takes as inputs a sequence of programs $C \in Com^*_{UID}$, a sequence of memories $M \in Mem^*_{UID}$, a scheduler $\mathcal{S}$, a user u, and a system state s_0 and returns as output a security decision $d \in \{\top, \bot\}$, where $\top$ stands for "secure" and $\bot$ stands for "insecure". We say that *ifEnf* is *sound* for a security condition iff the condition holds whenever *ifEnf* returns $\top$.

Database Access Control and Auxiliary Functions. Following Chapter 5, we define a *database access control mechanism* as a function *dbEnf* taking as input a system state s, a context *ctx* (representing the past history), an SQL command $q \in \mathcal{Q}$, and a user $u \in \mathcal{U}$ and returning a security decision in $\{\top, \bot\}$, where $\top$ stands for "authorized" and $\bot$ stands for "denied".

According to the operational semantics defined in Chapter 5, a database access control mechanism can be invoked also during the execution of triggers. We assume that there are (1) a function *inTrigger* that inspects the context *ctx* and determines whether the mechanism has been invoked during the processing of a trigger, and (2) a function *sessUser* extracting from *ctx* the user u' in whose session the query q has been executed. These functions can be easily constructed on top of the auxiliary functions given in Chapter 5. Finally, for simplicity, we assume that the database operational semantics always stores in the current context *ctx* the initial system state, and we denote the initial state stored in *ctx* by $init(ctx)$. Observe that the operational semantics in Chapter 5 can be trivially extended to store the initial state in the context, without affecting the results we presented in Chapters 5 and 6.

Simple, query-only programs. A *query-only* program is a program that contains only sequences of database queries that do not contain free program variables, i.e., statements of the form $x \leftarrow q$ such that $vars(q) = \emptyset$, or **skip** statements. Furthermore, a program is *simple* in case it contains exactly one statement. Hence, a simple query-only program is either **skip** or $x \leftarrow q$. Finally, given

a sequence of simple query-only programs C, we denote by $\#_{qry}(C)$ the number of statements in C that are not **skip**, i.e., the number of query statements in C.

Additional definitions on states and configurations. We say that a global configuration $\langle C, M, s, \mathcal{S}\rangle$ is (E, S)-*single-epoch*, for an epoch predicate E and a selection predicate S, iff $[\![E, S]\!](r) = \{r\}$, where r is the longest run obtained by starting from $\langle C, M, s, \mathcal{S}\rangle$.

We say that a global state $\langle M, \langle s, ctx\rangle\rangle$ is *reachable* iff there is a context ctx' such that $\langle s, ctx'\rangle$ is reachable in the LTS defined by the database semantics in Chapter 5, $triggers(ctx) = \epsilon$, and $triggers(ctx') = \epsilon$.

Safe programs. Let C be a sequence of simple, query-only programs, $\langle M, s\rangle$ be a global state, and $\mathcal{S}$ be a scheduler. We say that C is *safe* for $\langle M, s\rangle$ and $\mathcal{S}$ iff $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{*} \langle \epsilon, \epsilon, \langle s', ctx'\rangle, \mathcal{S}'\rangle$. Furthermore, we say that C is *safe* for $\mathcal{S}$ and a set of global states $\mathbb{S}$ iff it is safe for any $\langle M, s\rangle$ in $\mathbb{S}$.

7.5.2 Reduction

We now present our reduction that builds a DBAC mechanism *dbEnf* starting from an enforcement mechanism *ifEnf*. In the following, we represent commands as pairs $\langle u, q\rangle$, where $u \in UID$ is the user executing the command and q is the executed query. Furthermore, we denote by $\langle u, q\rangle$ the query that is given as input to *dbEnf*. In a nutshell, the reduction extracts from the context ctx all commands $\langle u_1, q_1\rangle, \ldots, \langle u_n, q_n\rangle$ that have already been executed and the initial system state s_0. We associate to each command $\langle u_i, q_i\rangle$ a WhileSql program c_i. Furthermore, we construct the program c associated with the current query $\langle u, q\rangle$. Then, *dbEnf* authorizes $\langle u, q\rangle$ iff (1) *ifEnf* says that the programs $c_1 \cdot \ldots \cdot c_n \cdot c$ are secure for the internal user u, the initial state s_0, and the memories $\langle u_1, m_0\rangle \cdot \ldots \cdot \langle u_n, m_0\rangle \cdot \langle u, m_0\rangle$, where $[\![x]\!](m_0) = 0$ for all $x \in Vars$, and (2) executing the programs $c_1 \cdot \ldots \cdot c_n \cdot c$ does not result in producing a security exception.

Reduction 7.1. Let s, ctx, q, and u be respectively the system state, the context, the query, and the user given as input to the DBAC mechanism. From the context ctx, we extract the sequence L of all the commands that have been executed from the start of the interaction. Note that L contains only the commands directly issued by the users, not the commands executed as part of triggers. We represent each command as a pair $\langle u, q\rangle$, where u is the user executing the command and q is the SQL command itself. Let L' be the sequence of commands obtained by conditionally appending to L the current query, i.e., $L' = L \cdot \langle u, q\rangle$ if $\neg inTrigger(ctx)$ and $L' = L$ otherwise. We now construct a WhileSql program $c_i \in Com_{UID}$ and a memory $m_i \in Mem_{UID}$ for each query in L': for each $1 \leq i \leq |L'|$, the program c_i is $\langle u_i, s_i\rangle$, where $L'(i) = \langle u_i, q_i\rangle$, x_i is a fresh variable, and the statement s_i is $x_i \leftarrow q_i$ in case (1) $i = |L'|$ and $\neg inTrigger(ctx)$, or (2) the i-th query in the history has been successfully executed (i.e., without security exceptions occurring when executing the command or the triggers executed in response to the command) and $s_i =$ **skip** otherwise, and the memory m_i is $\langle u_i, \lambda x \in Vars.\ 0\rangle$. The scheduler $\mathcal{S}$ is as follows: $\mathcal{S} = 0^{\infty}$. The access control mechanism $dbEnf(s, ctx, q, u)$ returns $\top$ iff (1) $ifEnf(c_1 \cdot \ldots \cdot c_{|L'|}, m_1 \cdot \ldots \cdot m_{|L'|}, \mathcal{S}, sessUser(ctx), init(ctx)) = \top$, and (2) $\langle c_1 \cdot \ldots \cdot c_{|L'|}, m_1 \cdot \ldots \cdot m_{|L'|}, \langle init(ctx), \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{*} \langle \epsilon, \epsilon, \langle s', ctx'\rangle, \mathcal{S}'\rangle$ and the last observation in τ does not contain security exceptions. ■

To illustrate, consider an history consisting of two queries $\langle u_1, q_1\rangle$ and $\langle u_2, q_2\rangle$, where the former is authorized and the latter is not authorized, a system state s, and a context ctx such that $inTrigger(ctx) = \bot$. Furthermore, assume that the user u issues the query q. Our reduction first constructs the programs $c_1 = \langle u_1, x_1 \leftarrow q_1\rangle$, $c_2 = \langle u_2,$ **skip**$\rangle$, and $c = \langle u, x \leftarrow q\rangle$. Then, it authorizes $\langle u, q\rangle$ iff (1) *ifEnf* says that the sequential execution of the programs $c_1 \cdot c_2 \cdot c$ is secure for u given the memories $\langle u_1, m_0\rangle \cdot \langle u_2, m_0\rangle \cdot \langle u, m_0\rangle$ and the state $init(ctx)$, and (2) the last query (or any of the associated triggers) does not throw security exceptions. The second condition ensures that the WhileSql's semantics, which directly embeds the checks on database integrity, agrees with the database semantics from Chapter 5, which does not contain these checks.

7.5.3 Security Analysis

Here, we connect our security condition for internal attackers to the data confidentiality notion from Chapter 6.

Stability. A key property for enforcement mechanisms is *stability*. Informally, an enforcement mechanism *ifEnf* is u-stable iff *ifEnf*'s security decision depends only on the traces produced by the user-based epochs associated with the user u but it does not depend on the last statement's result. Stability ensures that *ifEnf*'s security decision does not leak information.

Two runs r_1 and r_2 are u-*epoch-equivalent*, where $u \in UID$, iff there is a bijection $\mu : [\![E^{user}, S_u^{user}]\!](r_1) \rightarrow [\![E^{user}, S_u^{user}]\!](r_2)$ such that (1) μ preserves the ordering of epochs, and (2) for all $e_1 \in [\![E^{user}, S_u^{user}]\!](r_1)$, the initial configurations in e_1 and $\mu(e_1)$ are indistinguishable for u and the

traces in e_1 and $\mu(e_1)$ are the same. We say that an enforcement mechanism *ifEnf* is *u-stable* iff $\mathit{ifEnf}(C, M, \mathcal{S}, u, s) = \mathit{ifEnf}(C', M', \mathcal{S}', u, s')$ whenever: (a) C terminates starting from state $\langle M, s\rangle$ and scheduler $\mathcal{S}$ and produces run r_1 iff C' terminates starting from state $\langle M', s'\rangle$ and scheduler $\mathcal{S}'$ and produces run r_2, (b) the last statement executed in r_1 and r_2 is the same, and (c) if r_1 terminates, then r_1' and r_2' are u-epoch-equivalent, where r_1' (respectively r_2') is the run obtained from r_1 (respectively r_2) by replacing the observation produced by the last statement with ϵ. Namely, an enforcement mechanism *ifEnf* is u-stable iff *ifEnf*'s security decision depends only on the traces produced by the epochs associated with the user u (but it does not depend on the last statement's result).

Main Result. The main result of this section is that a sound and stable enforcement mechanism for internal attackers can be used to construct a provably secure DBAC mechanism with respect to the data confidentiality property and the attacker model from Chapter 6. We prove Theorem 7.1 in Appendix D.

Theorem 7.1. *Let ifEnf be an enforcement mechanism for* WhileSql *programs and dbEnf be the mechanism constructed using Reduction 7.1. If for all $u \in$ UID, (1) ifEnf is sound for the progress-sensitive variant of Definition 7.2 given E^{user} and S_u^{user}, and (2) ifEnf is u-stable, then dbEnf provides data confidentiality with respect to the attacker model from Chapter 6.*

We remark that Theorem 7.1 presents a way of constructing a provably secure DBAC mechanism by re-using existing IFC techniques for internal attackers; it is related neither to the security condition for external attackers (Definition 7.3) nor to the security monitor from Section 7.6. Furthermore, the theorem highlights the importance of stability, which prevents leaks through the IFC's security decision, for the resulting DBAC mechanism's security. In Section 7.5.5, we show how this result can be extended to IFC mechanisms for external attackers.

7.5.4 Extending the results to the Truman Model

The operational semantics from Chapter 5 targets the Non-Truman model for DBAC, as do the SQL standard and many existing database systems. Therefore, both Reduction 7.1 and Theorem 7.1 apply only to the Non-Truman model setting.

We now extend our results to the Truman model. To do so, we develop a technique for deriving a Truman model mechanism from one in the Non-Truman model. Since the Truman model semantics has been defined only for `SELECT` queries [131, 165] and Truman and Non-Truman semantics agree for boolean queries (see Chapter 3), we focus only on non-boolean `SELECT` queries.

Reduction 7.2. Let $s = \langle db, U, sec, T, V\rangle$, ctx, q, and u be the system state, the context, the non-boolean `SELECT` query, and the user given as input to the database access control mechanism respectively. Furthermore, let *NTdbEnf* be a database access control mechanism under the Non-Truman model semantics. The Truman model mechanism *TdbEnf* can be constructed as follows: (1) the mechanism executes the query $q = \{\overline{x} \mid \varphi\}$ over the database db and retrieves the set of tuples T produced as result, and (2) *TdbEnf* returns the set of tuples $\{\overline{t} \in T \mid \mathit{NTdbEnf}(s, ctx, \texttt{SELECT}\ \varphi[\overline{x} \mapsto \overline{t}], u) = \top\}$, namely it returns only the authorized tuples in q's result. ■

By combining Reductions 7.1 and 7.2, it follows that stability and soundness are enough to ensure that the resulting DBAC mechanism is secure. We prove this result in Appendix D.

Theorem 7.2. *Let ifEnf be an enforcement mechanism for* WhileSql *programs and dbEnf be the database access control mechanism for the Truman model semantics constructed using Reductions 7.1 and 7.2. If for all user $u \in$ UID,(1) ifEnf is sound with respect to the progress-sensitive variant of Definition 7.2 given E^{user} and S_u^{user}, and (2) ifEnf is u-stable, then dbEnf is secure and sound for the Truman model, i.e., it satisfies Definitions 3.8 and 3.18.*

7.5.5 From internal attackers to external attackers

Reduction 7.1 constructs a provably secure DBAC mechanism starting from a sound IFC mechanism for internal attackers. Despite that, most of the existing IFC mechanisms target external attackers, not internal ones. These IFC mechanisms cannot, therefore, be used with our construction. To address this, we present a way of using IFC mechanisms that target external attackers to determine the security of simple query-only programs with respect to internal attackers. By combining this with Reduction 7.1, we can, therefore, construct provably secure DBAC mechanisms using sound IFC mechanisms for external attackers as a building block.

Security for internal attackers requires that the attacker's knowledge remains constant inside each epoch. Lemma 7.1 states that, for the sequential scheduler, simple query-only programs, and user-based epochs, if the program corresponding to each epoch is secure, then so is the original program.

Lemma 7.1. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs, s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, and $u \in UID$ be a user. Furthermore, let r be the longest run obtained starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$. If for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, (1) the sequence of programs $C_i^{\lceil n/2\rceil}$ is secure with respect to u for $\mathcal{S}$, s_i, $M_i^{\lceil n/2\rceil}$, E^{user}, S_u^{user} for the progress-sensitive variant of Definition 7.2, and (2) $\mathcal{R}(\langle C_i, M_i, \langle s_i, \epsilon\rangle, \mathcal{S}_i\rangle, \lceil n/2\rceil)$ is a (E^{user}, S_u^{user})-single-epoch configuration, then C is secure with respect to u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user} for the progress-sensitive variant of Definition 7.2, where $\mathcal{R}(\langle C, M, s, \mathcal{S}\rangle, m)$ denotes the global configuration $\langle C^m, M^m, s, \mathcal{S}\rangle$.*

Before connecting security for internal and external attackers, we introduce a transformation on global configurations. The transformation ensures that the runs that start from the transformed configuration (1) leak the same information as those starting from the original configuration, and (2) do not modify the access control policy associated with the original tables and views. This guarantees that the allowed knowledge used in the security condition for external attackers does not contribute to a program's security.

Definition 7.4. Let $\langle D, \Gamma\rangle$ be a system configuration and $\langle C, M, \langle\langle db, U, sec, T, V\rangle, ctx\rangle, \mathcal{S}\rangle$ be a global configuration such that $\langle db, U, sec, T, V\rangle$ is a database state in $\Omega_{D,\Gamma}$ and C is a sequence of simple query-only programs that do not contain `ADD USER` commands. Furthermore, let D' be the database schema obtained by extending D with a predicate symbol TMP_R for each predicate symbol R in D. We denote by $\mathcal{T}(\langle C, M, \langle\langle db, U, sec, T, V\rangle, ctx\rangle, \mathcal{S}\rangle)$ the global configuration $\langle C', M', \langle\langle db', U', sec', T', V'\rangle, ctx'\rangle, \mathcal{S}'\rangle$ over the system configuration $\langle D', \Gamma\rangle$ defined as follows:

- the sequence of programs $C' \in Com_{UID}^{|C|+1}$ is as follows:
  - $C'(0) = \langle admin, x_{1,1} \leftarrow$ `REVOKE` $TMP_{R_1^1}$ `FROM` $u_1; \ldots x_{1,n_1} \leftarrow$ `REVOKE` $TMP_{R_{n_1}^1}$ `FROM` $u_1; \ldots x_{m,1} \leftarrow$ `REVOKE` $TMP_{R_1^m}$ `FROM` $u_m; \ldots x_{m,n_m} \leftarrow$ `REVOKE` $TMP_{R_{n_m}^m}$ `FROM` $u_m\rangle$, where $u_1, \ldots, u_m$ are all users in U and for each user u_i, $\{R_1^i, \ldots, R_{n_i}^i\}$ is the set containing all relation schemas R_j^i in D such that there is no `GRANT` $\langle op, u_i, \langle$`SELECT`$, R_j^i\rangle, admin\rangle \in sec$, where $op \in \{\oplus, \oplus^*\}$, i.e., $\{R_1^i, \ldots, R_{n_i}^i\} = \{R_j^i \mid R_j^i \in D \wedge \neg\exists op \in \{\oplus, \oplus^*\}.\ \langle op, u_i, \langle$`SELECT`$, R_j^i\rangle, admin\rangle \in sec\}$,
  - $C'(i) = \langle u_{i-1},$ `CREATE TRIGGER` tr `ON` T `AFTER` ev `IF` φ `DO GRANT SELECT ON` TMP_R `TO` u `AS` $uc\rangle$ if $c_{i-1} =$ `CREATE TRIGGER` tr `ON` T `AFTER` ev `IF` φ `DO GRANT SELECT ON` R `TO` u `AS` uc, where $C(i-1) = \langle u_{i-1}, c_{i-1}\rangle$,
  - $C'(i) = \langle u_{i-1},$ `CREATE TRIGGER` tr `ON` T `AFTER` ev `IF` φ `DO GRANT SELECT ON` TMP_R `TO` u `WITH GRANT OPTION AS` $uc\rangle$ if $c_{i-1} =$ `CREATE TRIGGER` tr `ON` T `AFTER` ev `IF` φ `DO GRANT SELECT ON` R `TO` u `WITH GRANT OPTION AS` uc, where $C(i-1) = \langle u_{i-1}, c_{i-1}\rangle$,
  - $C'(i) = \langle u_{i-1},$ `CREATE TRIGGER` tr `ON` T `AFTER` ev `IF` φ `DO REVOKE SELECT ON` TMP_R `FROM` u `AS` $uc\rangle$ if $c_{i-1} =$ `CREATE TRIGGER` tr `ON` T `AFTER` ev `IF` φ `DO REVOKE SELECT ON` R `FROM` u `AS` uc, where $C(i-1) = \langle u_{i-1}, c_{i-1}\rangle$,
  - $C'(i) = \langle u_{i-1},$ `GRANT SELECT ON` TMP_R `TO` $u\rangle$ if $c_{i-1} =$ `GRANT SELECT ON` R `TO` u, where $C(i-1) = \langle u_{i-1}, c_{i-1}\rangle$,
  - $C'(i) = \langle u_{i-1},$ `GRANT SELECT ON` TMP_R `TO` u `WITH GRANT OPTION`$\rangle$ if $c_{i-1} =$ `GRANT SELECT ON` R `TO` u `WITH GRANT OPTION`, where $C(i-1) = \langle u_{i-1}, c_{i-1}\rangle$,
  - $C'(i) = \langle u_{i-1},$ `REVOKE SELECT ON` TMP_R `FROM` $u\rangle$ if $c_{i-1} =$ `REVOKE SELECT ON` R `FROM` u, where $C(i-1) = \langle u_{i-1}, c_{i-1}\rangle$,
  - $C'(i) = \langle u_{i-1},$ `CREATE VIEW` $V : q$ `AS` $u;$ `CREATE VIEW` $TMP_V : q'$ `AS` $u\rangle$ if $c_{i-1} =$ `CREATE VIEW` $V : q$ `AS` u, where $C(i-1) = \langle u_{i-1}, c_{i-1}\rangle$ and q' is the query obtained from q by replacing each predicate symbol R with TMP_R, and
  - $C'(i) = C(i-1)$ otherwise,
- the sequence of memories $M' \in Com_{UID}^{|M|+1}$ is $\langle admin, m_0\rangle \cdot M$, where m_0 is the memory where all values are set at 0.
- the runtime state is as follows:
  - $db'(R) = db(R)$ and $db'(TMP_R) = \emptyset$ for all R in D,
  - $U' = U$,
  - $sec' = sec \cup \{\langle \oplus, u, \langle$`SELECT`$, TMP_R\rangle, admin\rangle \mid u \in U \wedge R \in D \wedge \neg\exists op \in \{\oplus, \oplus^*\}.\ \langle op, u, \langle$`SELECT`$, R\rangle, admin\rangle\} \cup \{\langle op, u', \langle$`SELECT`$, TMP_R\rangle, u''\rangle \mid \langle op, u', \langle$`SELECT`$, R\rangle, u''\rangle \in sec\}$,
  - $T = \{transfTrigger(t) \mid t \in T\}$, where $transfTrigger(t) = t$ in case t's action is not a `GRANT` or `REVOKE` command involving a `SELECT` privilege and $transfTrigger(\langle id, u, e, R, \phi, \langle op, u, \langle$`SELECT`$, R\rangle\rangle, m\rangle) = \langle id, u, e, R, \phi, \langle op, u, \langle$`SELECT`$, TMP_R\rangle\rangle, m\rangle$ otherwise,
  - $V' = V \cup \{transfView(v) \mid v \in V\}$, where $transfView(\langle id, o, q, m\rangle) = \langle TMP_{id}, o, q', m\rangle$ and q' is the query obtained from q by replacing each predicate symbol R with TMP_R, and
  - $ctx' = ctx$,
- the scheduler $\mathcal{S}'$ is $\mathcal{S}$. □

We also introduce an inverse transformation $\mathcal{T}^{-1}$ that allows one to revert the effects of the transformation on a system state. The inverse transformation is defined as follows: $\mathcal{T}^{-1}(\langle db, U, sec, T, V\rangle) = \langle db', U, sec', T', V'\rangle$, where db' is obtained by dropping all relation schemas of the form TMP_R, sec' is obtained by (1) removing from sec all `GRANT`s associated with `SELECT` privileges involving predicate symbols not of the form TMP_R, and (2) replacing all occurrences of predicate symbols of the form TMP_R with R, T' is obtained by replacing all predicate symbols of the form TMP_R with R in the triggers in T, and V is obtained by dropping all views that refer to predicate symbols of the form TMP_R.

Lemma 7.2 connects the security conditions for external and internal attackers for a single epoch. Observe that the lemma works for a slightly modified version of the security condition for external attackers, which supports also database-level users.

Lemma 7.2. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs (without `ADD USER` commands), s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, $u \in UID$ be a user, and $\langle C', M', \langle s', \epsilon\rangle, \mathcal{S}'\rangle$ be the configuration $\mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$. If $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$ is (E^{user}, S_u^{user})-single-epoch and the sequence of programs C' is secure with respect to the user $db(u)$ for $\mathcal{S}'$, s', and M' according to the progress-sensitive variant of Definition 7.3 (extended to handle users in $UID \cup \{db(u) \mid u \in UID\}$), then the sequence of programs C is secure with respect to the attacker u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user} according to the progress-sensitive variant of Definition 7.2.*

We are now ready to introduce Reduction 7.3, which shows how to use an IFC mechanism for external attackers to construct an IFC mechanism for internal attackers when restricted to the sequential scheduler and simple query-only programs.

Reduction 7.3. Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs (without `ADD USER` commands), s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, $u \in UID$ be a user, and *ifEnf* be an IFC mechanism for external attackers. The reduction works as follows. We compute the longest run r that can be obtained by starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$ and the set of epochs $[\![E^{user}, S_u^{user}]\!](r)$. Then, the IFC mechanism *ifEnf'* for internal attackers returns $\top$ iff for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, $ifEnf(C', M', \mathcal{S}, u, s') = \top$, where $\mathcal{T}(\langle C_i^{\lceil n/2\rceil}, M_i^{\lceil n/2\rceil}, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle) = \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$. ■

Finally, Theorem 7.3 states that Reduction 7.3 can be used to construct a sound IFC mechanism for internal attackers using a sound IFC mechanism for external attackers as a building block.

Theorem 7.3. *Let ifEnf be an IFC mechanism that is sound with respect to the progress-sensitive variant of Definition 7.3 (extended to handle users in $UID \cup \{db(u) \mid u \in UID\}$) and ifEnf' be the mechanism constructed in Reduction 7.3. Then, ifEnf' is a sound progress-sensitive IFC mechanism for internal attackers (for user-based epochs and simple query-only programs that do not contain `ADD USER` commands).*

Proof. The soundness of the reduction follows from Lemma 7.1, which establish the soundness of considering the single epochs separately, and Lemma 7.2, which establish the soundness of using a mechanism for external attackers to check security against internal attackers for a single epoch. □

We remark that Theorem 7.3 refers to a slightly extended version of the progress-sensitive variant of Definition 7.3 that supports also users of the form $db(u)$, where $u \in UID$. Our results can be immediately extended also to conditions that supports only users in UID by replacing each query statement $x \leftarrow q$ executed by a user u with $x \leftarrow q; \mathbf{out}(u, x)$. Indeed, this transformation ensures that all database-level observations are turned into program-level observations, thereby preserving the attacker's knowledge.

7.5.6 From progress-sensitive to progress-insensitive security

Theorems 7.1–7.3 refer to the progress-sensitive variants of Definitions 7.2 and 7.3. Our results, however, can be lifted to the progress-insensitive setting.

Theorem 7.4 states that the progress-sensitive and progress-insensitive definitions coincide for external attackers for safe programs.

Theorem 7.4. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com^*_u$ be a sequence of simple query-only programs, and $\langle M_0, s_0\rangle$ be a reachable global state such that C is safe for $\mathcal{S}$ and $[\langle M_0, s_0\rangle]_u$. Then, C is secure with respect to Definition 7.3 for $\mathcal{S}$, M, s, and u iff C is secure with respect to the progress-sensitive variant of Definition 7.3 for $\mathcal{S}$, M, s, and u.*

Theorem 7.5 states that the progress-sensitive and progress-insensitive definitions coincide for internal attackers whenever the programs associated with the single epochs are safe.

Theorem 7.5. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com_u^*$ be a sequence of simple query-only programs, and $\langle M_0, s_0\rangle$ be a reachable global state such that $|C| = |M|$ and for all $1 \leq i \leq |M|$, $M(i) = \langle u_i, m_i\rangle$ and $C(i) = \langle u_i, c_i\rangle$. Furthermore, let r be the longest run obtained starting from $\langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle$. If $C_i^{\lceil n/2\rceil}$ is safe for $\mathcal{S}$ and $[\langle M_i^{\lceil n/2\rceil}, s_i\rangle]_u$ for all $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^n \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then C is secure with respect to Definition 7.2 for $\mathcal{S}$, M, s, u, E^{user}, and S_u^{user} iff C is secure with respect to the progress-sensitive variant of Definition 7.2 for $\mathcal{S}$, M, s, u, E^{user}, and S_u^{user}.*

We remark that the programs produced by Reductions 7.1 and 7.2 satisfy the following conditions: (1) each query statement q_i produces an output according to the WhileSql's semantics, (2) the resulting programs are simple query-only programs, and (3) the programs associated to each epoch are safe with respect to the epoch's initial state. Hence, Theorems 7.4 and 7.5 can be used to extend our results to the progress-insensitive setting.

7.5.7 A note on integrity

In this section, we used a sound IFC mechanism as a building block for constructing a provably secure DBAC mechanism that satisfies the data confidentiality notion from Chapter 6. Our results, however, do not extend to database integrity. Even though variants of non-interference can be used to enforce integrity for programs, the database integrity notion from Chapter 6 directly accounts for the semantics of SQL access control. As a result, database integrity introduces restrictions, such as those associated with `GRANT` and `REVOKE` commands, that have no immediate formalization in terms of non-interference.

7.6 Enforcing end-to-end security

This section presents a monitor that provably secures WhileSql programs with respect to external attackers. To achieve end-to-end security across the database and applications, the monitor must keep track of dependencies at the database level (between queries) and at the program level (between variables) and ensure that the information released by output statements complies with the current security policy. To this end, we combine dynamic information-flow tracking with query determinacy [124] and disclosure lattices [26]. We refer the reader to Chapter 2 for an introduction to these concepts.

We leverage *disclosure lattices* to reason about the security of database queries [26]. Given two sets of queries Q_1 and Q_2, disclosure lattices provide a precise model for answering the following questions: (i) Does Q_1 reveal *more* information than Q_2? (ii) What *combined* information is revealed by Q_1 and Q_2? (iii) What *common* information is revealed by Q_1 alone and Q_2 alone? Observe that a security policy defines a set of database tables and views that a user is authorized to read. Hence, policies can be seen as sets of database queries, that is elements of a disclosure lattice. In the following, we fix a system configuration $\langle D, \Gamma\rangle$, where $D = \langle \Sigma, \mathbf{dom}\rangle$.

7.6.1 Preliminaries

Before presenting our security monitor, we define some preliminary concepts.

Predicate queries. A *predicate query* is a query of the form $T(\overline{v})$ where T is a relation schema in D and $\overline{v} \in \mathbf{dom}^{|T|}$. A predicate query represents a single tuple in the database. We denote by RC^{pred} the set of all predicate queries.

Query Support. Let q be a query and Q be a set of predicate queries. We say that Q *minimally determines* q, denoted $minDet_{D,\Gamma}(Q, q)$, iff $D, \Gamma \vdash Q \twoheadrightarrow q$, and there is no $Q' \subset Q$ such that D, $\Gamma \vdash Q' \twoheadrightarrow q$. The *support of* q, denoted $supp_{D,\Gamma}(q)$, is $\{Q \in 2^{RC^{pred}} \mid minDet_{D,\Gamma}(Q, q)\}$. Informally, the support of q contains all (minimal) sets of tuples that determine q. For instance, the query $T(1) \vee R(2)$ is minimally determined by $\{T(1), R(2)\}$. Hence, its support is $\{\{T(1), R(2)\}\}$. For simplicity, we write $supp(q)$ instead of $supp_{D,\Gamma}(q)$ when the system configuration $\langle D, \Gamma\rangle$ is clear from the context.

Well-formed integrity constraints. We say that a set Γ of integrity constraints is *well-formed* iff for all predicate queries $q \in RC^{pred}$, $supp_{D,\Gamma}(q) = \{\{q\}\}$. In the following, we consider only well-formed system configurations $\langle D, \Gamma\rangle$. Observe that common integrity constraints used in practice, such as primary key and foreign key constraints, are well-formed. An example of integrity constraint that is not well-formed is $T(1) \leftrightarrow R(2)$. Indeed, $supp_{D,\{T(1)\leftrightarrow R(2)\}}(T(1)) = \{\{T(1)\}, \{R(2)\}\}$.

7.6.2 Security monitor

Our enforcement mechanism for external attackers is a dynamic security monitor. It keeps track of how information flows across the programs and the database, and it terminates the execution whenever a statement may leak information. For simplicity, we assume there is a single attacker, represented by the user identifier ATK. Our mechanism can be generalized to support multiple attackers. We also assume that each user is associated with at most one program and that different programs use disjoint sets of variable identifiers. In the following, we denote by $Vars_{ATK}$ the set of variables occurring in ATK's program and by sec_0 the initial security policy.

Security Lattice. Our security monitor uses the determinacy-based disclosure lattice to track information. To handle both queries and memory variables, we extend the database schema D with a propositional symbol MEM_x for each variable $x \in Vars$ occurring in the monitored WhileSql programs. We denote by D^{ext} the extended database schema. Formally, our security lattice is the disclosure lattice $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ defined over D^{ext}, where $\sqsubseteq = \preceq^{\rightarrow}_{D^{ext},\Gamma}$. Note that we use labels of the form MEM_x to abstractly represent the information initially stored in the program memories, which does not come from the database.

Monitor States. A *monitor state* Δ is a function $Vars \cup RC^{pred} \cup \{\mathtt{pc}_u \mid u \in UID\} \to \mathcal{L}$ that associates each variable identifier and predicate query with a security label. The monitor state also keeps track of the security level associated with each program's security context. Since each user u is executing at most one program, we formalize the security contexts using identifiers of the form $\mathtt{pc}_u$, where $u \in UID$ is the user executing the program. For example, $\Delta(\mathtt{pc}_{\mathtt{Bob}})$ captures the security level of the condition if `Bob`'s program is executing one of the branches of an **if** statement. We lift Δ to expressions: $\Delta(e) = \bigsqcup_{x \in vars(e)} \Delta(x)$, where e is an expression and $vars(e)$ are e's free variables.

The monitor's initial state Δ_0 is as follows: (a) for each variable x, if $x \in Vars_{ATK}$, then $\Delta_0(x) = MEM_x$ and $\Delta_0(x) = \top$ otherwise, (b) for all $T(\overline{v}) \in RC^{pred}$, then $\Delta_0(T(\overline{v})) = cl(T(\overline{v}))$, and (c) for all $u \in UID$, $\Delta_0(\mathtt{pc}_u) = \bot$.

Mapping Queries to Labels. Our security monitor keeps track only of the dependencies between predicate queries. Hence, we use the function $L_{\mathcal{Q}}$ to derive the labels associated with relational calculus queries: $L_{\mathcal{Q}}(\Delta, q) = \bigsqcup_{Q \in supp(q)} \bigsqcup_{q' \in Q} \Delta(q')$. The function associates to a query q the join of the labels of all predicate queries in q's support. This ensures that $L_{\mathcal{Q}}(\Delta, q)$ takes into account the labels of all tuples that may influence q's results. For instance, given a monitor state Δ, the query $T(1) \wedge R(2)$ is associated with the label $\Delta(T(1)) \sqcup \Delta(R(2))$, thus capturing the fact that it may reveal information about both $T(1)$ and $R(2)$. As expected, for predicate queries, $L_{\mathcal{Q}}(\Delta, q) = \Delta(q)$.

Mapping Users to Labels. The function $L_{\mathcal{U}}$ maps users to labels in our security lattice. Since we are interested in end-to-end security guarantees, we associate the attacker ATK with the set of tables and views he is authorized to read according to the current security policy and to the initial policy sec_0 along with the labels associated with all the variables in ATK's program. Formally, $L_{\mathcal{U}}(s, u) = \top$ for any $u \neq ATK$ and $L_{\mathcal{U}}(s, ATK) = cl(auth(s, ATK) \cup auth(sec_0, ATK) \cup \bigcup_{x \in Vars_{ATK}} MEM_x)$. To illustrate, given a security policy sec_0 stating that the attacker can read the table T but not the table R and ignoring the labels associated with variables, $L_{\mathcal{U}}(s, ATK) = \bigsqcup_{\overline{v} \in \mathbf{dom}^{|T|}} cl(T(\overline{v}))$, i.e., the attacker is authorized to know anything contained in T.

The mapping of queries and users to labels from the disclosure lattice allows one to reason about information disclosure. For instance, if the above attacker observes the result of the query $q = T(1) \wedge R(2)$, this violates the security policy. In fact, $L_{\mathcal{Q}}(\Delta, q) \not\sqsubseteq L_{\mathcal{U}}(s, ATK)$ since $cl(\{T(1), R(2)\}) \not\sqsubseteq \bigsqcup_{\overline{v} \in Vals^{|T|}} cl(T(\overline{v}))$.

Expansion Process. To correctly handle triggers, our monitor rewrites each SQL command into WhileSql statements encoding the triggers' execution. We do so using the $expand(s, m, u, x \leftarrow q)$ function, which takes as input a runtime state s, a memory m, the user u executing the command, and a command $x \leftarrow q$, and produces as output a sequence of statements modeling the command's execution (together with all the triggers executed in response to q). We formalize the *expand* function in Section 7.9.3.

Additional Queries and Statements. Our monitor extends the WhileSql syntax and semantics with two designated queries $T \oplus \overline{e}$ and $T \ominus \overline{e}$, and five designated statements $\mathbf{asuser}(u', c)$, $\mathbf{dbout}(u, v, o, \tau)$, $\|x \leftarrow q\|$, $[c]$, and $\mathbf{set\ pc\ to}\ l$. The $T \oplus \overline{e}$ (respectively $T \ominus \overline{e}$) query inserts (respectively deletes) tuples from the database without firing triggers. The $\mathbf{asuser}(u', c)$ statement is used to execute the command c as the user u' (inside the session of the user u executing the $\mathbf{asuser}(u', c)$ command). The $\mathbf{dbout}(u, v, o, \tau)$ statement, instead, is used to print database-level events, e.g., policy changes. Finally, we use $\|x \leftarrow q\|$ to denote queries that have already been through the expansion process. All the above queries and statements are used during the expansion process.

The main goal of our work is studying the interplay between programs and databases. Therefore, to avoid timing leaks caused by executing multiple programs in parallel, the monitor's semantics

F-Assign
$$\frac{\mathbf{nsu}(x, \mathtt{pc}_u) \qquad \Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \Delta(e)]}{\langle \Delta, x := e, m, s\rangle \rightsquigarrow_u \langle \Delta', \varepsilon, m[x \mapsto [\![e]\!](m)], s\rangle}$$

F-Out
$$\frac{\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')}{\langle \Delta, \mathbf{out}(u', e), m, s\rangle \overset{\langle u', [\![e]\!](m)\rangle}{\rightsquigarrow}_u \langle \Delta, \varepsilon, m, s\rangle}$$

F-Expand
$$\frac{c_e = \mathit{expand}(s, x, q, u)}{\langle \Delta, x \leftarrow q, m, s\rangle \rightsquigarrow_u \langle \Delta, [c_e], m, s\rangle}$$

F-IfTrue
$$\frac{\begin{array}{c}[\![e]\!](m) = \mathbf{tt} \\ c' = [c_1 \,;\, \mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)] \\ \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]\end{array}}{\langle \Delta, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

F-Select
$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = \mathit{vars}(\varphi) \qquad \varphi' = \varphi[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \\ q = \texttt{SELECT}\ \varphi \qquad [\![q]\!](s, u) = \langle s', r, \epsilon, \epsilon\rangle \qquad \ell_\varphi = L_{\mathcal{Q}}(\Delta, \varphi) \sqcup \bigsqcup_{v \in \mathit{vars}(\varphi)} \Delta(v) \qquad \mathbf{nsu}(x, \mathtt{pc}_u)\end{array}}{\langle \Delta, \|x \leftarrow \texttt{SELECT}\ \varphi\|, m, s\rangle \rightsquigarrow_u \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi], \varepsilon, m[x \mapsto r], s'\rangle}$$

F-UpdateDatabaseOk
$$\frac{\begin{array}{c}\overline{v} = \langle [\![e_1]\!](m), \ldots, [\![e_n]\!](m)\rangle \qquad \otimes \in \{\oplus, \ominus\} \qquad q = T \otimes \overline{v} \\ [\![q]\!](s, u) = \langle s', r, \epsilon, \epsilon\rangle \qquad \ell_e = \bigsqcup_{1 \le i \le n} \Delta(e_i) \qquad \mathbf{nsu}(T(\overline{v}), \mathtt{pc}_u) \qquad \ell_e \sqsubseteq \Delta(T(\overline{v})) \qquad \mathbf{nsu}(x, \mathtt{pc}_u)\end{array}}{\langle \Delta, \|x \leftarrow T \otimes \{e_1, \ldots, e_n\}\|, m, s\rangle \rightsquigarrow_u \langle \Delta[T(\overline{v}) \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e, x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e], \varepsilon, m[x \mapsto r], s'\rangle}$$

F-UpdateConfigurationOk
$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = \mathit{vars}(q) \qquad q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \\ \mathit{isCfgCmd}(q') \qquad [\![q']\!](s, u) = \langle s', r, \epsilon, \epsilon\rangle \qquad \ell_{cmd} = \bigsqcup_{1 \le i \le n} \Delta(v_i) \\ \ell_{cmd} \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{ATK})) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{ATK})) \qquad \mathbf{nsu}(x, \mathtt{pc}_u)\end{array}}{\langle \Delta, \|x \leftarrow q\|, m, s\rangle \rightsquigarrow_u \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_{cmd}], \varepsilon, m[x \mapsto r], s'\rangle}$$

Figure 7.5: Security monitor – Selected Rules.

executes branching statements atomically, i.e., without interleaving the execution of other programs whenever a program is executing a branching statement. To do so, we introduce statements of the form $[c]$, which denote that c should be executed atomically. Finally, we use statements of the form **set pc to** l, where $l \in \mathcal{L}$, to update the label associated with the executing program's context.

Relaxed no-sensitive upgrade checks. Our enforcement mechanism is a dynamic security monitor. A common technique for preventing implicit leaks of sensitive information in this setting is using *no-sensitive upgrade* (NSU) checks [22, 169]. Intuitively, a NSU check restricts the changes only to the variables whose label is at least that of the current execution's context, i.e., only for variables x such that $\Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$. This guarantees that changes to "low" variables never happen in "high" execution contexts. NSU checks, however, are rather restrictive: they may block executions that do not leak information. This is particularly relevant for security lattices with many labels, such as our disclosure lattice, where many labels are simply unrelated and NSU checks often fail.

To address this, we propose a simple relaxation of NSU checks that still prevents leaks of sensitive information. In particular, our relaxed NSU checks exploit the fact that the initial security policy sec_0 can be used to determine which labels can be considered as permanently "low", no matter how the policy is modified during the execution. In more detail, given a variable x, the relaxed NSU check is defined as follows: $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, ATK)) \vee \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$, where sec_0 is the initial security policy. Our relaxed NSU check is satisfied whenever the standard NSU check is. Additionally, our relaxed NSU check allows flows of informations whenever the security context $\mathtt{pc}_u$ is permanently low, i.e., $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, ATK))$. In the following, we denote by $\mathbf{nsu}(x, \mathtt{pc})$ the predicate $\Delta(\mathtt{pc}) \sqsubseteq cl(auth(sec_0, ATK)) \vee \Delta(\mathtt{pc}) \sqsubseteq \Delta(x)$, where x is either a variable identifier or a predicate query, $\mathtt{pc}$ is an identifier of the form $\mathtt{pc}_u$, and u is a user identifier.

To illustrate, our relaxed NSU check allows changes to a variable x whose label is, say, $cl(T(1))$ in an execution context such that $\Delta(\mathtt{pc}_u) = cl(V(2))$ whenever the attacker ATK is authorized to read both the table T and V with respect to the initial policy sec_0. A standard NSU check, instead, would have prevented the assignment since $cl(V(1)) \not\sqsubseteq cl(T(1))$.

Enforcement Rules. Figure 7.5 presents selected rules associated with the execution of queries. We present the full operational semantics of our monitor in Section 7.9.2.

The rule F-Assign updates the monitor state whenever there is an assignment. The rule prevents leaks using relaxed NSU checks. The rule F-Out ensures that the monitor produces only secure output events. In particular, it prints to the user u' the value associated with the expression e only if the security labels associated with the printed information and with the program counter are authorized to flow to u', i.e., $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')$. The rule F-IfTrue, instead, executes the **then** branch c_1 in an **if** statement and updates the labels associated with $\mathtt{pc}_u$ based on the label associated with the **if**'s condition. The rule relies on the **set pc to** l command to reset the label of $\mathtt{pc}_u$ when leaving the **then** branch. Also note that the rule encapsulates both the **then** branch c_1 and the **set pc to** l statement inside an atomic statement $[c_1 \text{ ; } \textbf{set pc to } l]$ to prevent internal timing channels caused by the scheduler. We remark that the above rules implement rather standard dynamic information-flow tracking [134].

The rule F-Expand ensures that triggers as well as constraint checking is de-sugared into normal WhileSql code that is handled by the other rules. The rule uses the *expand* function, which we formalize in Section 7.9.3. The F-Select rule ensures, using relaxed NSU checks, that query results are stored only in variables with the proper security labels. The rule, finally, updates the label associated with the variable storing the query's result to correctly propagate the flow of information.

The rule F-UpdateConfigurationOk handles configuration commands, i.e., `GRANT`, `REVOKE`, `ADD USER`, and `CREATE` commands. Since changes to the configuration are visible to ATK, the rule ensures that such changes are performed only in contexts that are initially low for the attacker, i.e., $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, ATK))$. Furthermore, the rule prevents leaks of sensitive information using the free variables in the commands using the check $\ell_{cmd} \sqsubseteq cl(auth(sec_0, ATK))$. The rule also uses relaxed NSU checks to ensure that query results are stored only in variables with the proper security labels. The rule uses the predicate $isCfgCmd(q)$, which returns $\top$ iff q is a configuration command. Finally, the rule F-UpdateDatabaseOk handles queries that modify the database's content. The rule ensures that there are no changes to security labels based on secret information using relaxed NSU checks. Furthermore, the rule keeps track of the labels associated with the information stored in the database by updating the monitor's state Δ.

Observe that in WhileSql policy changes generate public events. This eliminates leaks through authorization channels [17], and no additional checks for such leaks (cf. *channel context bounds* in [18]) are needed.

Soundness. Theorem 7.6, proven in Appendix D, states that our enforcement mechanism is sound for external attackers, i.e., it satisfies Definition 7.3 with $\rightsquigarrow$ as the underlying evaluation relation.

Theorem 7.6. *For all sequences of programs $C \in Com^*_{UID}$, schedulers $\mathcal{S}$, sequences of memories $M \in Mem^*_{UID}$, and system states s, whenever $r = \langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}^n \langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $K^{\rightsquigarrow}_{ATK}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{ATK,sec}(M, s) \subseteq K^{\rightsquigarrow}_{ATK}(\langle M, s\rangle, C, \mathcal{S}, trace(r^i))$, where $K^{\rightsquigarrow}_{ATK}$ refers to Definition 7.1 with $\rightsquigarrow$ as evaluation relation and the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$.*

Example 7.3. Let T, V, Z be three tables, t be the trigger defined by the following command:

$$\texttt{CREATE TRIGGER } t \texttt{ ON } T \texttt{ AFTER INS IF } V(1) \texttt{ DO INSERT } 1 \texttt{ INTO } Z \texttt{ AS } admin,$$

and s be a system state containing t. The trigger is executed whenever a tuple is inserted in the table T, and t inserts 1 in Z if $V(1)$ holds in the database state. The statement $x \leftarrow$ `INSERT 2 INTO` T is expanded as follows (provided all operations are authorized by the policy and there are no integrity constraints):

$$\begin{array}{l}
y \leftarrow \texttt{SELECT } V(1)\\
\textbf{if } y \textbf{ then}\\
\quad x \leftarrow \|T \oplus 2\|\\
\quad z \leftarrow \|Z \oplus 1\|\\
\quad \textbf{dbout}(db(ATK), \|T \oplus 2\|, x, \epsilon)\\
\textbf{else}\\
\quad x \leftarrow \|T \oplus 2\|\\
\quad \textbf{dbout}(db(ATK), \|T \oplus 2\|, x, \epsilon)
\end{array}$$

Suppose the attacker ATK executes $x \leftarrow$ `INSERT 2 INTO` $T; w \leftarrow$ `SELECT` $Z(1); \textbf{out}(ATK, w)$ from an initial database state s_0 where the tables T and Z are empty and the table V contains a single record with value 1. We illustrate the behavior of the monitor for the security policy where ATK cannot read V but can read and modify T and Z. Intuitively, the program is insecure since the presence of 1 in Z depends (implicitly) on the presence of 1 in V, which ATK cannot read.

Consider the program execution with the initial state s_0 as above, and initial monitor state Δ_0 such that $\Delta_0(x) = MEM_x$, $\Delta_0(w) = MEM_w$, $\Delta_0(y) = \Delta_0(z) = \top$ (since y and z do not occur in ATK's program), and $\Delta_0(\mathtt{pc}_{ATK}) = \bot$. The attacker's label is $L_{\mathcal{U}}(s_0, ATK) = \bigsqcup_{v \in Vals} cl(T(v)) \sqcup \bigsqcup_{v \in Vals} cl(Z(v)) \sqcup cl(\{MEM_x, MEM_w\})$. The monitor would apply the rules F-Expand (explained above), F-Select, F-IfTrue, F-UpdateDatabaseOk (twice), F-DbOut (not shown), F-SetPc (not shown), F-Select, and F-Out. The evaluation of the first `SELECT` statement yields $\Delta' = \Delta_0[y \mapsto \Delta(V(1)) \sqcup \bot]$, i.e., $\Delta'(y) = cl(V(1))$. Observe that the `SELECT` statement is successful ($\textbf{nsu}(x, \mathtt{pc}_{ATK})$ holds given that $\Delta_0(\mathtt{pc}_{ATK}) = \bot$). The evaluation of the boolean condition y yields $\Delta' = \Delta[y \mapsto cl(V(1)), \mathtt{pc}_{ATK} \mapsto cl(V(1))]$. For the subsequent database update, the monitor checks whether $\textbf{nsu}(T(2), \mathtt{pc}_{ATK})$ holds for the monitor state Δ'. Namely, the monitor checks whether $cl(V(1)) \sqsubseteq l_0 \vee cl(V(1)) \sqsubseteq cl(T(2))$ holds, where $l_0 = \bigsqcup_{v \in Vals} cl(T(v)) \sqcup \bigsqcup_{v \in Vals} cl(Z(v))$. Since this is not the case, the monitor stops the execution and correctly prevents the leakage. ■

7.6.3 A note on implementations

While most of the rules defining our monitor's semantics can be easily implemented in practice, there are some points that are worth discussing.

Representing the monitor state. The monitor state Δ associates to each predicate query q a label $l \in \mathcal{L}$. However, the set of predicate queries is infinite. To address this, one can explicitly keep track only of the labels of those queries that have been involved in `INSERT` or `DELETE` operations. This allows one to always store only a finite number of labels, without affecting the monitor's security. Furthermore, each label l may consist of an infinite set of predicate queries. Instead of storing the predicate queries, one can just store the finite set of queries Q associated with the label l.

Determining the relation between labels. Given two labels l and l', determining whether $l \sqsubseteq l'$ requires to determine whether a set of queries Q determines another set of queries Q'. However, query determinacy is undecidable for the relational calculus [124]. To address this, one can use a sound under-approximation of determinacy to decide whether Q determines Q', such as the one we presented in Chapter 6. This defines a sound under-approximation $\sqsubseteq_{ax}$ of the relation $\sqsubseteq$.

Computing the $L_{\mathcal{Q}}$ function. Computing the result of $L_{\mathcal{Q}}(\Delta, q)$ requires to compute the set $supp_{D,\Gamma}(q)$. This is challenging for two reasons: (1) the notion of $supp_{D,\Gamma}(q)$ relies on query determinacy, and (2) $supp_{D,\Gamma}(q)$ may contain an infinite amount of queries. The first problem can be addressed by employing a sound over-approximation of query determinacy. Differently from $\sqsubseteq_{ax}$,

computing $supp_{D,\Gamma}(q)$ requires an over-approximation to preserve the monitor's security. To address the second problem one can represent sets of predicate queries using non-boolean queries.

Approximations and precision. While the above points do not impact the monitor's security, they can influence its precision and performance. In general, better under- and over-approximations of determinacy may lead to more precise monitors but may reduce the monitor's performance. For example, one may employ relatively simple syntactic under- and over-approximations of determinacy, such as the one we presented in Chapter 6. This allows the monitor to efficiently compute all the security checks. However, these approximations often lead to imprecise results. For instance, the under-approximation from Chapter 6 is not able to determine that $cl(T(1)) \sqsubseteq cl((R(1) \vee \neg R(1)) \wedge T(1))$. Alternatively, one may rely on more complex approximations, such as using automated theorem provers, or even exact solutions for fragments of the relational calculus [124, 127]. While these approaches may greatly increase the monitor's precision, they usually have a high computational complexity.

7.7 Related Work

Database Access Control. As we already discussed in Chapters 3 and 6, many security conditions have been proposed for `SELECT`-only attackers [26, 27, 131, 165]. These conditions are usually built on top of well-known database problems, such as determinacy [124], instance-based determinacy [107], and certain answer [9], and they target a setting where both the database's content and the policy are static. Our work, instead, considers a stronger attacker model where malicious users may dynamically modify the policy as well as exploiting advanced database features. In this respect, the attacker model considered in this chapter is similar to the one from Chapter 6. We refer the reader to Section 6.6 for an in-depth discussion of DBAC attacker models.

The security conditions we reviewed above often refer to *secure DBAC mechanisms*. They thus mix security and enforcement concerns. In contrast, our condition characterize the security of a given sequence of commands only in terms of the attacker's knowledge. As a result, our condition provides a purely semantic characterization of database security in dynamic settings, where both the security policy and the database's content may change.

Bender et al. [26, 27] introduce disclosure lattices to reason about the security of `SELECT` queries in the presence of fine-grained security policies. Specifically, they use a disclosure lattice based on *query homomorphisms* limited to a restricted class of queries (filter-project queries) [26, 27]. In contrast, we exploit a disclosure lattice based on query determinacy to track how information flows across programs and databases. Observe also that our security condition for internal attackers may serve as a baseline for justifying soundness of their policy enforcement algorithms.

Since the work we presented in this chapter is based on the operational semantics from Chapter 5 and relies on an attacker model similar to the one from Chapter 6, we refer the reader to Section 6.6 for a detailed comparison with Mandatory Access Control research.

Information-flow Control for database-backed applications. Until recently, little work has been done on IFC for database-backed applications [24, 49, 50, 57, 96, 143, 167]. In contrast to existing studies, our work has the following distinguishing features: (1) a realistic model of database systems, which accounts for advanced database constructs like triggers, views, and dynamic policies, (2) a security monitor combining information-flow tracking with disclosure lattices, and (3) a soundness proof of security for a realistic database model. Existing works either consider simple database models or provide informal soundness arguments.

A recent trend is to integrate database query mechanisms directly inside programming languages, see, for instance, Microsoft's Language Integrated Queries (LINQ) [3]. In response to this, information-flow approaches have been proposed for languages supporting LINQ [24, 141]. Schoepe et al. [141] propose SeLinq, a secure type system for a subset of the F# language extended with language integrated queries. Their work builds on the LINQ's foundations of Cheney et al. [46], which supports only `SELECT` queries. Furthermore, they assume that both the database content and the security policy are static and they support only column-level policies over the database. Balliu et al. [24], instead, present JsLinq, a framework for securing multi-tier applications. Their work too builds on the work of Cheney et al. [46]. Hence, it is subject to similar restrictions as SeLinq, namely it supports only `SELECT` queries and column-level policies. Observe that JsLinq supports declassification through escape hatches, which is similar to our use of `GRANT` commands to delegate read privileges. In contrast to these approaches, our work adopts a realistic database model, which accounts for advanced features, and we support dynamic policies. Additionally, we combine dynamic monitoring with disclosure lattices to track fine-grained tuple-level dependencies, which SeLinq and JsLinq cannot track.

Corcoran et al. [57] present SeLinks, an extension of the Links programming language [56] that allows a database and a web application to collaborative enforce security policies using dependent types. Internally, SeLinks builds on top of the Fable system [155]. In contrast to our work, SeLinks

targets a rather simple database model that does not support security-critical features like integrity constraints and triggers, which can be used to leak sensitive information (see Chapter 6).

Sif, developed by Chong et al. [50], is a software framework that allows developers to build secure web applications. Programmers annotate variables and other data sources with fine-grained labels, and Sif automatically tracks information flows across the web application to guarantee the absence of leaks of sensitive information. Sif, however, only provides a limited database support: it requires the database interface to be annotated with Jif annotations at column-level. Fabric [114], instead, is a Jif extension for building secure distributed systems. While it considers persistent storage, it does not provide direct support for databases and for tracking information through them. Compared to these approaches, our work supports complex dynamic policies defined through `GRANT` and `REVOKE` commands and it tracks dependencies between variables and database tuples.

Schultz and Liskov [143] extend decentralized information-flow control to track information across database boundaries. Their work is based on concepts from multi-level security and it relies on polyinstantiation. In contrast to our work, Schultz and Liskov's approach cannot be directly applied to most SQL databases, which do not support multi-level security approaches (see Section 6.6 for a detailed discussion of this topic).

Chlipala [49] introduces UrFlow, a static analysis information-flow tool for the Ur/Web programming language. Similarly to our security policies, UrFlow support policies specified using SQL queries. Huang et al. [96], instead, present WebSSARI, a tool that combines static and runtime analysis to secure database-backed Php applications. Both UrFlow and WebSSARI consider a simplistic database model. In contrast, our work targets a realistic database model that supports advanced security-critical features, like triggers and dynamic policies. Additionally, our security monitor provide precise security guarantees.

Yang et al. [167] present Jacqueline, a framework for policy-agnostic programming for database-backed applications. The system combines dynamic information-flow control and faceted values to secure programs interacting with databases. The database model considered in [167] lacks many of the advanced features that are supported by our model. Observe also that Jacqueline's goal is executing programs in a secure way, not detecting security violations.

Griffin et al. [79] present Hails, a framework for developing web applications that supports mandatory access control policies. DbTaint [59], instead, enhances database data types with one-bit taint information. Compared to our approach, these works lack formal security justifications and advanced database features.

Security Conditions. Our security conditions from Section 7.4 are inspired by existing knowledge-based conditions for IFC [18, 21, 23]. Our condition for external attackers assumes perfect recall attackers and considers dynamic policies. The condition is similar to existing knowledge-based notions that use escape hatches for declassification, such as [21]. However, while escape hatches are often disclosed permanently [21], we support both `GRANT` and `REVOKE` commands. We refer the reader to Broberg et al. [39] for an extensive survey of IFC and dynamic policies.

Our security condition for internal attackers, instead, targets attackers that reset their knowledge about the system's state whenever the epoch changes. In this respect, the notion of non-interference between updates introduced by Hicks et al. [94], which requires that non-interference holds between policy updates, can be seen as a special case of our security condition instantiated with *permission-based epochs*. Permission-based epochs represent portions of a run where the policy does not change. We formalize them using the predicate E_u^{perm}, where $u \in \mathit{UID}$, defined as follows: $\langle\langle C, M, \langle s, \mathit{ctx}\rangle, \mathcal{S}\rangle, \langle C', M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle\rangle \in E_u^{perm}$ iff $\mathit{auth}(s, u) \neq \mathit{auth}(s', u)$.

Our security conditions in Section 7.4 are progress-insensitive. There are different flavors of progress-insensitivity in the literature. For instance, Hedin and Sabelfeld [93] present a security condition that differentiate between divergence and termination. Our security conditions, instead, follow state-of-the-art conditions for dynamic policies [18, 39] by not differentiating between divergence and termination. As a result, our conditions ignore leaks caused by the output trace's length. Observe, however, that our condition is subject to brute-forcing leaks similar to those for [93], with known information-theoretic bounds [19]. Note, however, that the results in Section 7.5 can be applied to both progress-sensitive and progress-insensitive techniques.

Label Models. The *universal lattice* introduced by Hunt and Sands [98] is similar in spirit to our disclosure lattice. The universal lattice allows one to express dependencies between variables, where the lattice's elements are sets of variables and the order relationship is set containment. In contrast, our disclosure lattice allows one to reason about complex dynamic policies in databases and applications. Observe also that the universal lattice can be embedded inside our disclosure lattice, by using predicate symbols of the form MEM_x for each variable x (see Section 7.6). By directly combining disclosure lattices with dynamic information-flow tracking, we can track fine-grained dependencies between variables and queries. Note that these dependencies would be lost using simpler label models, such as the standard "high" and "low" lattice.

Bridging DBAC and IFC. As we already mentioned, there are several IFC approaches that support databases [24, 49, 50, 57, 96, 143, 167]. Despite that, only little work has been done on bridging access control and information-flow control. To the best of our knowledge, John Rushby's paper [133] has been one of the few to formally characterize non-interference and access control in the same framework. Rushby presented three properties that any system monitored by a secure access control mechanism should satisfy. Informally, these three requirements are: (1) the output produced by a user's action should depend only on the values the user is authorized to read, (2) whenever a user's action changes the system's state, the resulting state must depend only on the values the user is authorized to read, and (3) whenever a user's action changes the system's state, the user should possess the corresponding write privilege. Rushby proved that, under specific conditions, a system that satisfies these requirements also satisfies non-interference. Our work can be seen as concretizing Rushby's ideas for a specific setting, namely database access control, and execution model, i.e., the database operational semantics from Chapter 5. In this respect, we present a reduction for constructing secure DBAC mechanisms using IFC techniques as a building block, and we provide a semantic characterization (based only on the attacker's knowledge) of our data confidentiality condition from Chapter 6 in terms of non-interference.

7.8 Conclusions

In this chapter, we developed common foundations for database access control and information-flow control, two previously unconnected research areas. These foundations rely on WhileSql, a simple imperative language with querying capabilities that builds on top of our database operational semantics from Chapter 5. We used these foundations to improve both DBAC and IFC.

Specifically, with respect to DBAC, we leveraged our security conditions to develop a general technique for constructing provably secure access control mechanisms from sound IFC enforcement mechanisms. This allows the DBAC community to benefit from existing IFC techniques and tools. Additionally, our security condition for internal attackers provides a semantic characterization for our data confidentiality notion from Chapter 6.

With respect to IFC, instead, we have developed a novel enforcement mechanism that monitors how information flows inside applications and across database boundaries. Our monitor combines dynamic information flow tracking with advanced DBAC concepts like disclosure lattices and query determinacy. As a result, it can track complex dependencies that depend on the database's content and it supports advanced features like dynamic policies and triggers.

7.9 Technical Details

Here we present some technical details about WhileSql and our enforcement mechanism.

7.9.1 From WhileSql to the database operational semantics of Chapter 5

The WhileSql's operational semantics from Section 7.3 builds on top of the database operational semantics formalized in Chapter 5. In particular, in the WhileSql semantics, the execution of the database commands is delegated to the function $[\![q]\!](s, u)$, which takes as input an SQL statement q, a runtime state $s \in \Omega_M$, and the user $u \in \mathit{UID}$ executing the command, and it returns a tuple $\langle s', r, em, \tau \rangle$, where $s' \in \Omega_M$ is the new runtime state, r is q's result, em is an error message, and τ is a trace of observation representing the public observations produced by the triggers activated by the query q. The function $[\![q]\!](s, u)$ is defined in Figures 7.7–7.9 and it relies on the transition relation $\rightarrow_f$ from Chapter 5, where f is a PDP. In the following, we instantiate f to be the PDP f_{int} developed in Chapter 5. This PDP ensures the integrity of the database, e.g., by avoiding unauthorized changes, but it does not provide confidentiality guarantees. Furthermore, we re-use various functions from Chapter 5, e.g., we reuse the functions res, Ex, and secEx to extract the outcomes of the command's execution from a runtime state, and we lift the auth function from system states to runtime states, i.e., $\mathit{auth}(\langle s, \mathit{ctx} \rangle, u)$ is simply $\mathit{auth}(s, u)$.

We remark that $[\![q]\!](\langle s, \mathit{ctx} \rangle, u)$ guarantees that the trigger transaction, which has been defined in Chapter 5, in the resulting runtime state $\langle s', \mathit{ctx}' \rangle$ is always ϵ. This, combined with the fact that the context for initial states is ϵ, ensures the correct execution of queries. As a consequence of this, $[\![q]\!](\langle s, \mathit{ctx} \rangle, u) = [\![q]\!](\langle s, \mathit{ctx}' \rangle, u)$ for any two contexts $\mathit{ctx}, \mathit{ctx}'$ such that the trigger transaction is ϵ.

For `SELECT` queries, $[\![q]\!](s, u)$ is defined only for boolean queries because the operational semantics in Chapter 5 supports only boolean queries. We refer the reader to Chapter 5 for a discussion on how to handle non-boolean queries.

For `INSERT` and `DELETE` queries, $[\![q]\!](s, u)$ relies on the function $\mathit{toTrace}$, defined in Figure 7.6, that takes as input an `INSERT` or `DELETE` command q and a trace of the database execution consisting

$$toTrace(q, \epsilon) = \epsilon$$

$$toTrace(q, s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_n}_f s_{n+1}) = toTrace(q, s_1 \xrightarrow{t_1}_f s_2) \cdot toTrace(q, s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_n}_f s_{n+1})$$

$$toTrace(q, s \xrightarrow{t}_f s') = \begin{cases} \langle public, t, \texttt{GRANT}\ p\ \texttt{TO}\ u \rangle & \text{if } res(s') = \top \wedge acC(s') = \top \\ & \wedge action(t) = \langle \oplus, u, p \rangle \wedge \\ & auth(s, u) \neq auth(s', u) \\ \langle public, t, \texttt{GRANT}\ p\ \texttt{TO}\ u\ \texttt{WITH GRANT OPTION} \rangle & \text{if } res(s') = \top \wedge acC(s') = \top \\ & \wedge action(t) = \langle \oplus^*, u, p \rangle \wedge \\ & auth(s, u) \neq auth(s', u) \\ \langle public, t, \texttt{REVOKE}\ p\ \texttt{FROM}\ u \rangle & \text{if } res(s') = \top \wedge acC(s') = \top \\ & \wedge action(t) = \langle \ominus, u, p \rangle \wedge \\ & auth(s, u) \neq auth(s', u) \\ \epsilon & \text{otherwise} \end{cases}$$

FIGURE 7.6:
Definition of the *toTrace* function. The functions *res* and *acC* are defined in Chapter 5, the *action* function takes as input a trigger and returns its action.

only of triggers (throwing neither security nor integrity exceptions) and extracts a trace of public observations $\langle public, q, r \rangle$ associated with the `GRANT` and `REVOKE` statements executed by the triggers in the trace. It also relies on the functions *acC* and *acA* that take as input a runtime state and retrieve the access control decisions associated with the trigger's condition and the trigger's action. We refer the reader to Chapter 5 for a formalization of these functions.

7.9.2 Enforcement Operational Semantics

Here we provide the full operational semantics of our security monitor. In the following, *ATK* denotes the user identifier associated with the attacker and sec_0 denotes the initial security policy. Furthermore, we call *extended* any WhileSql program c that possibly contains the additional commands introduced in Section 7.6.

Preliminaries. The auxiliary function $atomic(c)$ takes as input an extended WhileSql program and returns $\top$ if there are c', c'' such that $c = [c']$ or $c = [c'] \,;\, c''$. The auxiliary function $query(c)$ takes as input an extended WhileSql program and returns $\top$ if there are x, q such that $c = x \leftarrow q$ or $c = \|x \leftarrow q\|$. Finally, Figure 7.10 illustrates how the queries of the form $T \otimes \overline{v}$, where $\otimes \in \{\oplus, \ominus\}$, are handled by the underlying database.

Enforcement Rules. Given a user $u \in UID$, the relation $\rightsquigarrow_u$, shown in Figures 7.11–7.14, formalizes the local operational semantics of our dynamic monitor. Figure 7.15 presents the security monitor rules for the global semantics. The rules F-Eval-Step and F-Eval-End are similar to the WhileSql semantics. Additionally, the monitor uses the F-Atomic-Statement rule to handle the atomic execution of code. Observe that the atomic execution does not consume the scheduler.

7.9.3 Expansion Process

Here we illustrate our expansion process for queries.

Extracting triggers. Our expansion process uses the function $triggers : \Omega_M \times \mathcal{Q} \times \mathcal{U} \rightarrow (\mathcal{TRIGGER} \times \mathcal{U} \times \mathcal{Q} \times \mathcal{Q} \times \mathcal{U} \times \mathcal{U})^*$ that provides an interface to the database and returns the triggers in the form of tuples $\langle t, u, cond, act, invk, owner \rangle$, where t is the trigger's identifier, u specifies the user under which privileges the triggers is to be executed (i.e., u is either the trigger's owner or the trigger's invoker depending on the trigger's definition), *cond* specifies t's `WHEN` condition, *act* is t's action, *invk* is the user that fired the trigger, and *owner* is the trigger's owner. Note that the variables associated with the tuple in the original command have already been replaced in both *cond* and *act*. Therefore, if the original command contains program variables, then *cond* and *act* may both contain program variables. Observe that the *triggers* function can be implemented on top of the functions we defined in Chapter 5. Given a trigger represented as $t = \langle t, u, cond, act, invk, owner \rangle$, we denote by $id(t)$ the identifier *id*, by $user(t)$ the user u, by $cond(t)$ the condition *cond*, by $act(t)$ the action *act*, by $invoker(t)$ the invoker *invk*, and by $owner(t)$ the user *owner*.

Instrumented Commands. An instrumented command is a pair $\langle c, r \rangle$ such that c is an `INSERT` command, a `DELETE` command, or a trigger (represented as specified above using a 4-tuple), and

$$[\![\texttt{SELECT}\ \phi]\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{SELECT},\phi\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{SELECT},\phi\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{ADD\_USER}\ u']\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{ADD\_USER},u'\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{ADD\_USER},u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{CREATE}\ obj]\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{CREATE},obj\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{CREATE},obj\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{GRANT}\ p\ \texttt{TO}\ u']\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{GRANT}^*\ p\ \texttt{TO}\ u']\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus^*,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus^*,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{REVOKE}\ p\ \texttt{FROM}\ u']\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle \ominus,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle \ominus,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

FIGURE 7.7: Definition of the $[\![q]\!](s,u)$ function – part 1.

$$[\![q]\!](s,u) = \begin{cases} \langle s', r, \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s' \wedge triggers(s') = \epsilon \\ \langle s', r, \epsilon, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_n \in \mathcal{TRIGGER}. \\ & \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots s_n \xrightarrow{t_n}_f s' \wedge \\ & \quad r = res(s_1) \wedge Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ & \quad \bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ & \quad triggers(s') = \epsilon \wedge \neg secEx(s') \wedge Ex(s') = \emptyset \wedge \\ & \quad \tau = toTrace(\langle u, \texttt{INSERT}, T, \bar{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots s_n \xrightarrow{t_n}_f s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s' \wedge secEx(s') \wedge triggers(s') = \epsilon \\ \langle s', \dagger, \langle \mathbf{IntEx}, Ex(s')\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s' \wedge Ex(s') \neq \emptyset \wedge triggers(s') = \epsilon \\ \langle s', \dagger, \langle t, \texttt{B}, \mathbf{IntEx}, Ex(s')\rangle, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_{n-1} \in \mathcal{TRIGGER}. \\ & \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\ & \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ & \quad \bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ & \quad acA(s') \wedge acC(s') \wedge Ex(s') \neq \emptyset \wedge triggers(s') = \epsilon \wedge \\ & \quad \tau = toTrace(\langle u, \texttt{INSERT}, T, \bar{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots \xrightarrow{t_{n-1}}_f s_n) \\ \langle s', \dagger, \langle t, \texttt{W}, \mathbf{SecEx}, \emptyset\rangle, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_{n-1} \in \mathcal{TRIGGER}. \\ & \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\ & \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ & \quad \bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ & \quad \neg acC(s') \wedge triggers(s') = \epsilon \wedge \\ & \quad \tau = toTrace(\langle u, \texttt{INSERT}, T, \bar{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots \xrightarrow{t_{n-1}}_f s_n) \\ \langle s', \dagger, \langle t, \texttt{B}, \mathbf{SecEx}, \emptyset\rangle, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_{n-1} \in \mathcal{TRIGGER}. \\ & \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\ & \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ & \quad \bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ & \quad acC(s') \wedge \neg acA(s') \wedge triggers(s') = \epsilon \wedge \\ & \quad \tau = toTrace(\langle u, \texttt{INSERT}, T, \bar{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots \xrightarrow{t_{n-1}}_f s_n) \end{cases}$$

FIGURE 7.8: Definition of the $[\![q]\!](s,u)$ function – part 2. Note that $q = \texttt{INSERT}\ \bar{t}\ \texttt{INTO}\ T$.

$$
[\![q]\!](s,u) = \begin{cases}
\langle s', r, \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s' \wedge secEx(s') \\
\langle s', r, \epsilon, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_n \in \mathcal{TRIGGER}. \\
& \quad s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots s_n \xrightarrow{t_n}_f s' \wedge \\
& \quad r = res(s_1) \wedge Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\
& \quad \textstyle\bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\
& \quad triggers(s') = \epsilon \wedge \neg secEx(s') \wedge Ex(s') = \emptyset \wedge \\
& \quad \tau = toTrace(\langle u, \texttt{DELETE}, T, \overline{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots \xrightarrow{t_n}_f s') \\
\langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s' \wedge secEx(s') \wedge triggers(s') = \epsilon \\
\langle s', \dagger, \langle \mathbf{IntEx}, Ex(s')\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s' \wedge Ex(s') \neq \emptyset \wedge triggers(s') = \epsilon \\
\langle s', \dagger, \langle t, \texttt{B}, \mathbf{IntEx}, Ex(s')\rangle, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_{n-1} \in \mathcal{TRIGGER}. \\
& \quad s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\
& \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\
& \quad \textstyle\bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\
& \quad acA(s') \wedge acC(s') \wedge Ex(s') \neq \emptyset \wedge triggers(s') = \epsilon \wedge \\
& \quad \tau = toTrace(\langle u, \texttt{DELETE}, T, \overline{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots \xrightarrow{t_{n-1}}_f s_n) \\
\langle s', \dagger, \langle t, \texttt{W}, \mathbf{SecEx}, \emptyset\rangle, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_{n-1} \in \mathcal{TRIGGER}. \\
& \quad s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\
& \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\
& \quad \textstyle\bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\
& \quad \neg acC(s') \wedge triggers(s') = \epsilon \wedge \\
& \quad \tau = toTrace(\langle u, \texttt{DELETE}, T, \overline{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots \xrightarrow{t_{n-1}}_f s_n) \\
\langle s', \dagger, \langle t, \texttt{B}, \mathbf{SecEx}, \emptyset\rangle, \tau\rangle & \text{if } \exists s_1,\ldots,s_n \in \Omega_M.\ \exists t_1,\ldots,t_{n-1} \in \mathcal{TRIGGER}. \\
& \quad s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\
& \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\
& \quad \textstyle\bigwedge_{1\le i<n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\
& \quad acC(s') \wedge \neg acA(s') \wedge triggers(s') = \epsilon \wedge \\
& \quad \tau = toTrace(\langle u, \texttt{DELETE}, T, \overline{t}\rangle, s_1 \xrightarrow{t_1}_f \ldots \xrightarrow{t_{n-1}}_f s_n)
\end{cases}
$$

FIGURE 7.9: Definition of the $[\![q]\!](s,u)$ function – part 3. Note that $q = \texttt{DELETE}\ \overline{t}\ \texttt{FROM}\ T$.

$$
[\![T \oplus \overline{t}]\!](s,u) = \begin{cases}
\langle s'', res(s'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\overline{t}\rangle}_f s' \wedge \neg secEx(s') \wedge s = \langle db, U, S, T, V, ctx\rangle \wedge \\
& \quad s'' = \langle db[R \oplus \overline{t}], U, S, T, V, ctx\rangle \\
\langle s, \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\overline{t}\rangle}_f s' \wedge secEx(s')
\end{cases}
$$

$$
[\![T \ominus \overline{t}]\!](s,u) = \begin{cases}
\langle s'', res(rs'), \epsilon, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s' \wedge \neg secEx(s') \wedge s = \langle db, U, S, T, V, ctx\rangle \wedge \\
& \quad s'' = \langle db[R \ominus \overline{t}], U, S, T, V, ctx\rangle \\
\langle s, \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{t}\rangle}_f s' \wedge secEx(s')
\end{cases}
$$

FIGURE 7.10: Definition of the $[\![q]\!](s,u)$ function – part 4.

F-Skip

$$\frac{}{\langle \Delta, \mathbf{skip}, m, s\rangle \rightsquigarrow_u \langle \Delta, \varepsilon, m, s\rangle}$$

F-Assign

$$\frac{\mathbf{nsu}(x, \mathtt{pc}_u) \qquad \Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \Delta(e)]}{\langle \Delta, x := e, m, s\rangle \rightsquigarrow_u \langle \Delta', \varepsilon, m[x \mapsto [\![e]\!](m)], s\rangle}$$

F-Out

$$\frac{\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')}{\langle \Delta, \mathbf{out}(u', e), m, s\rangle \overset{\langle u', [\![e]\!](m)\rangle}{\rightsquigarrow}_u \langle \Delta, \varepsilon, m, s\rangle}$$

F-Seq

$$\frac{\langle \Delta, c_1, m, s\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c_1', m', s'\rangle}{\langle \Delta, c_1 \,;\, c_2, m, s\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c_1' \,;\, c_2, m', s'\rangle}$$

F-SeqEmpty

$$\frac{}{\langle \Delta, \varepsilon \,;\, c, m, s\rangle \rightsquigarrow_u \langle \Delta, c, m, s\rangle}$$

F-IfTrue

$$\frac{\begin{array}{c}[\![e]\!](m) = \mathbf{tt} \\ c' = [c_1 \,;\, \mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)] \\ \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]\end{array}}{\langle \Delta, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

F-IfFalse

$$\frac{\begin{array}{c}[\![e]\!](m) = \mathbf{ff} \\ c' = [c_2 \,;\, \mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)] \\ \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]\end{array}}{\langle \Delta, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

F-WhileTrue

$$\frac{\begin{array}{c}[\![e]\!](m) = \mathbf{tt} \\ c' = [c \,;\, \mathbf{while}\ e\ \mathbf{do}\ c \,;\, \mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)] \\ \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]\end{array}}{\langle \Delta, \mathbf{while}\ e\ \mathbf{do}\ c, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

F-WhileFalse

$$\frac{\begin{array}{c}[\![e]\!](m) = \mathbf{ff} \\ c' = [\mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)] \\ \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]\end{array}}{\langle \Delta, \mathbf{while}\ e\ \mathbf{do}\ c, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

Figure 7.11: Security monitor – local operational semantics for assignments, print, and control flow statements.

F-ExpandedCode

$$\frac{\langle \Delta, c, m, s\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle}{\langle \Delta, [c], m, s\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', [c'], m', s'\rangle}$$

F-RemoveExpandedCode

$$\frac{}{\langle \Delta, [\varepsilon], m, s\rangle \rightsquigarrow_u \langle \Delta, \varepsilon, m, s\rangle}$$

Figure 7.12: Security monitor – local operational semantics for atomic statements.

F-UpdateLabels

$$\frac{}{\langle \Delta, \mathbf{set\ pc\ to}\ l, m, s\rangle \rightsquigarrow_u \langle \Delta[\mathtt{pc}_u \mapsto l], \varepsilon, m, s\rangle}$$

F-AsUser

$$\frac{\mathit{query}(c) \qquad \langle \Delta[\mathtt{pc}_{u'} \mapsto \Delta(\mathtt{pc}_u)], c, m, s\rangle \rightsquigarrow_{u'} \langle \Delta', c', m', s'\rangle}{\langle \Delta, \mathbf{asuser}(u', c), m, s\rangle \rightsquigarrow_u \langle \Delta'[\mathtt{pc}_{u'} \mapsto \Delta(\mathtt{pc}_{u'})], c', m', s'\rangle}$$

F-DbOut

$$\frac{\begin{array}{c}\mathit{vars}(v) = \{v_1, \ldots, v_x\} \quad \mathit{vars}(o) = \{o_1, \ldots, o_y\} \quad \tau = \tau_1 \cdot \ldots \cdot \tau_n \\ \mathit{vars}(\tau(1)) = \{k_1^1, \ldots, k_{m_1}^1\} \ldots \mathit{vars}(\tau(n)) = \{k_1^n, \ldots, k_{m_n}^n\} \\ v' = v[v_1 \mapsto [\![v_1]\!](m), \ldots, v_x \mapsto [\![v_x]\!](m)] \\ o' = o[o_1 \mapsto [\![o_1]\!](m), \ldots, o_y \mapsto [\![o_y]\!](m)] \qquad \tau'(i) = \tau(i)[k_1^i \mapsto [\![k_1^i]\!](m), \ldots, k_{n_i}^i \mapsto [\![k_{m_i}^i]\!](m)] \\ \ell_{obs} = \bigsqcup_{x \in \mathit{vars}(v)} \Delta(x) \sqcup \bigsqcup_{x \in \mathit{vars}(o)} \Delta(x) \sqcup \bigsqcup_{1 \le i \le |\tau|} \bigsqcup_{x \in \mathit{vars}(\tau(i))} \Delta(x) \\ u'' = \begin{cases} u' & \text{if } u' \neq \mathit{ATK} \wedge u' \neq \mathit{public} \wedge \tau\restriction_{\mathit{ATK}} = \epsilon \\ \mathit{ATK} & \text{otherwise} \end{cases} \qquad \Delta(\mathtt{pc}_u) \sqcup \ell_{obs} \sqsubseteq L_{\mathcal{U}}(s, u'')\end{array}}{\langle \Delta, \mathbf{dbout}(u', v, o, \tau), m, s\rangle \overset{\langle u', v', o', \tau'\rangle}{\rightsquigarrow}_u \langle \Delta, \varepsilon, m, s\rangle}$$

Figure 7.13: Security monitor – local operational semantics for the new commands (**set pc**, **asuser**, and **dbout**).

F-Expand

$$\frac{c_e = \mathit{expand}(s, x, q, u)}{\langle \Delta, x \leftarrow q, m, s\rangle \rightsquigarrow_u \langle \Delta, [c_e], m, s\rangle}$$

F-Select

$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = \mathit{vars}(\varphi) \qquad \varphi' = \varphi[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \\ q = \texttt{SELECT}\ \varphi \qquad [\![q]\!](s,u) = \langle s', r, \epsilon, \epsilon\rangle \qquad \ell_\varphi = L_{\mathcal{Q}}(\Delta, \varphi) \sqcup \bigsqcup_{v \in \mathit{vars}(\varphi)} \Delta(v) \qquad \mathbf{nsu}(x, \mathtt{pc}_u)\end{array}}{\langle \Delta, \|x \leftarrow \texttt{SELECT}\ \varphi\|, m, s\rangle \rightsquigarrow_u \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi], \varepsilon, m[x \mapsto r], s'\rangle}$$

F-UpdateDatabaseOk

$$\frac{\begin{array}{c}\overline{v} = \langle [\![e_1]\!](m), \ldots, [\![e_n]\!](m)\rangle \qquad \otimes \in \{\oplus, \ominus\} \qquad q = T \otimes \overline{v} \\ [\![q]\!](s,u) = \langle s', r, \epsilon, \epsilon\rangle \qquad \ell_e = \bigsqcup_{1 \le i \le n} \Delta(e_i) \qquad \mathbf{nsu}(T(\overline{v}), \mathtt{pc}_u) \qquad \ell_e \sqsubseteq \Delta(T(\overline{v})) \qquad \mathbf{nsu}(x, \mathtt{pc}_u)\end{array}}{\langle \Delta, \|x \leftarrow T \otimes \{e_1, \ldots, e_n\}\|, m, s\rangle \rightsquigarrow_u \langle \Delta[T(\overline{v}) \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e, x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e], \varepsilon, m[x \mapsto r], s'\rangle}$$

F-UpdateConfigurationOk

$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = \mathit{vars}(q) \qquad q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \\ \mathit{isCfgCmd}(q') \qquad [\![q']\!](s,u) = \langle s', r, \epsilon, \epsilon\rangle \qquad \ell_{cmd} = \bigsqcup_{1 \le i \le n} \Delta(v_i) \\ \ell_{cmd} \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{ATK})) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{ATK})) \qquad \mathbf{nsu}(x, \mathtt{pc}_u)\end{array}}{\langle \Delta, \|x \leftarrow q\|, m, s\rangle \rightsquigarrow_u \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_{cmd}], \varepsilon, m[x \mapsto r], s'\rangle}$$

Figure 7.14: Security monitor – local operational semantics for database operations.

F-Eval-Step

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, |C|\}, u' \in \mathit{UID}.\ C(n) \neq \langle u', \varepsilon\rangle \\ |C| = |M| \qquad n = 1 + (n'\ \mathbf{mod}\ |C|) \qquad C(n) = \langle u, c\rangle \qquad M(n) = \langle u, m\rangle \\ \langle \Delta, c, m, s\rangle \overset{\tau}{\rightarrow}_u \langle \Delta', c', m', s'\rangle \quad \neg \mathit{atomic}(c') \qquad C' = C(1){\cdot}\ldots{\cdot}C(n-1){\cdot}\langle u, c'\rangle{\cdot}C(n+1){\cdot}\ldots{\cdot}C(|C|) \\ M' = M(1){\cdot}\ldots{\cdot}M(n-1){\cdot}\langle u, m'\rangle{\cdot}M(n+1){\cdot}\ldots{\cdot}M(|C|)\end{array}}{\langle \Delta, C, M, s, n'{\cdot}\mathcal{S}\rangle \overset{\tau}{\rightarrow} \langle \Delta', C', M', s', \mathcal{S}\rangle}$$

F-Atomic-Statement

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, |C|\}, u \in \mathit{UID}.\ C(i) \neq \langle u', \varepsilon\rangle \qquad |C| = |M| \\ n = 1 + (n'\ \mathbf{mod}\ |C|) \qquad C(n) = \langle u, c\rangle \qquad M(n) = \langle u, m\rangle \qquad \langle \Delta, c, m, s\rangle \overset{\tau}{\rightarrow}_u \langle \Delta', c', m', s'\rangle \\ \mathit{atomic}(c') \qquad C' = C(1){\cdot}\ldots{\cdot}C(n-1){\cdot}\langle u, c'\rangle{\cdot}C(n+1){\cdot}\ldots{\cdot}C(|C|) \\ M' = M(1){\cdot}\ldots{\cdot}M(n-1){\cdot}\langle u, m'\rangle{\cdot}M(n+1){\cdot}\ldots{\cdot}M(|C|)\end{array}}{\langle \Delta, C, M, s, n'{\cdot}\mathcal{S}\rangle \overset{\tau}{\rightarrow} \langle \Delta', C', M', s', n'{\cdot}\mathcal{S}\rangle}$$

F-Eval-End

$$\frac{\begin{array}{c}1 \le n \le |C| \qquad \forall n' < n, u' \in \mathit{UID}.\ C(n') \neq \langle u', \varepsilon\rangle \qquad C(n) = \langle u, \varepsilon\rangle \qquad |C| = |M| \\ C' = C(1){\cdot}\ldots{\cdot}C(n-1){\cdot}C(n+1){\cdot}\ldots{\cdot}C(|C|) \qquad M' = M(1){\cdot}\ldots{\cdot}M(n-1){\cdot}M(n+1){\cdot}\ldots{\cdot}M(|C|)\end{array}}{\langle \Delta, C, M, s, \mathcal{S}\rangle \overset{\tau}{\rightarrow} \langle \Delta, C', M', s, \mathcal{S}\rangle}$$

Figure 7.15: Security monitor – global operational semantics.

$r \in \{\texttt{ok}, \texttt{dis}, \texttt{secEx}, \texttt{ex}\}$. Note that both commands and triggers may contain program variables. Given an instrumented command $\langle c, r\rangle$, $first(\langle c, r\rangle) = c$ and $second(\langle c, r\rangle) = r$.

Constructing the execution paths. Let $\bar{t}$ be a sequence of commands and triggers. We denote by $paths(\bar{t})$ the following function:

$$paths(\epsilon) = \emptyset$$
$$paths(c) = \{\langle c, \texttt{ok}\rangle, \langle c, \texttt{secEx}\rangle, \langle c, \texttt{ex}\rangle\} \text{ where } c \text{ is an SQL command}$$
$$paths(t) = \{\langle t, \texttt{ok}\rangle, \langle t, \texttt{dis}\rangle, \langle t, \texttt{secEx}\rangle, \langle t, \texttt{ex}\rangle\} \text{ where } t \text{ is a trigger}$$
$$paths(t{\cdot}\bar{t}) = merge(paths(t), paths(\bar{t}))$$
$$merge(S_1, S_2) = \{s_1{\cdot}s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2 \wedge \neg\exists s_1', o.\ (s_1 = s_1'{\cdot}\langle o, \texttt{ex}\rangle \vee s_1 = s_1'{\cdot}\langle o, \texttt{secEx}\rangle)\}$$

Configuration-consistent Execution Paths. The execution paths computed through the *paths* function may, in general, contain unfeasible paths. For instance, they may contain commands terminating in a security exception even though this may not happen given the current security policy. To take this into account, we define the notion of a configuration-consistent execution path. Note that there is no analogous of configuration-consistent path for integrity exceptions caused by `INSERT` and `DELETE` commands. The reason is that these exceptions depend on the database content and are directly handled by the expansion process.

In the following, let s be a runtime state, m be a memory, u be a user, $\bar{t}$ be a sequence of instrumented commands (i.e., an execution path), c be a database command, and t be a trigger. Furthermore, given a command (or trigger) o containing program variables, we denote by $o(m)$ the command (or trigger) obtained from o by replacing all program variables with the corresponding values in m. The configuration-consistency relation is defined as follows:

1. $s, m, u \models^{cfg} \langle c, \texttt{ok}\rangle$ if $allowed(s, u, c(m))$.
2. $s, m, u \models^{cfg} \langle c, \texttt{ex}\rangle$ if $allowed(s, u, c(m))$ and c is an `INSERT` or `DELETE` command.
3. $s, m, u \models^{cfg} \langle c, \texttt{secEx}\rangle$ if $\neg allowed(s, u, c(m))$.
4. $s, m, u \models^{cfg} \langle t, \texttt{ok}\rangle$ if $allowed(s, u, t(m))$.
5. $s, m, u \models^{cfg} \langle t, \texttt{ex}\rangle$ if $allowed(s, u, t(m))$ and t's action is an `INSERT` or `DELETE` command.
6. $s, m, u \models^{cfg} \langle t, \texttt{secEx}\rangle$ if $\neg allowed(s, u, t(m))$.
7. $s, m, u \models^{cfg} \langle t, \texttt{dis}\rangle$.
8. $s, m, u \models^{cfg} \langle o, r\rangle{\cdot}\bar{t}$ iff $\bar{t} \neq \epsilon$, $s, m, u \models^{cfg} \langle o, r\rangle$, $r \neq \texttt{dis}$, and $apply(s, o(m)), m, u \models^{cfg} \bar{t}$.
9. $s, m, u \models^{cfg} \langle o, \texttt{dis}\rangle{\cdot}\bar{t}$ iff $\bar{t} \neq \epsilon$, $s, m, u \models^{cfg} \langle o, \texttt{dis}\rangle$ and $s, m, u \models^{cfg} \bar{t}$.

The above definition relies on the functions *allowed* and *apply*, which are formalized in Figures 7.16 and 7.17. Let $\bar{t}$ be a sequence of commands and triggers (possibly containing program variables), s be a database state, m be a memory, and u be a user. We denote by $consPaths(\bar{t}, s, u)$ the set of all configuration-consistent paths derivable from $\bar{t}$. Formally, $consPaths(\bar{t}, s, m, u) = \{\bar{t'} \in paths(\bar{t}) \mid s, m, u \models^{cfg} \bar{t'}\}$.

Weakest precondition for database updates. We now introduce weakest preconditions for `INSERT` and `DELETE` operations on databases. In the following, we assume that constants are not used inside predicate symbols. E.g., the formula $T(\overline{v})$ is expressed as $\exists \overline{x}.\ T(\overline{x}) \wedge \overline{x} = \overline{v}$. Let ϕ be a relational calculus sentence that does not refer to views (for formulae that refer to views, one can first replace the views with their definitions and later compute the weakest precondition). Furthermore, we denote by $T \oplus \overline{v}$ (respectively $T \ominus \overline{v}$) an insertion (respectively deletion) operation on the database. Note that $\overline{v}$ may contain program variables. The weakest precondition of ϕ given $T \oplus \overline{v}$, written $wp(\phi, T \oplus \overline{v})$, is obtained by replacing all occurrences of $T(\overline{x})$ with $(T(\overline{x}) \vee \overline{x} = \overline{v})$. Similarly, the weakest precondition of ϕ given $T \ominus \overline{v}$, written $wp(\phi, T \ominus \overline{v})$, is obtained by replacing all occurrences of $T(\overline{x})$ with $(T(\overline{x}) \wedge \overline{x} \neq \overline{v})$.

Weakest precondition for execution paths. In Figure 7.18, we extend weakest preconditions from single `INSERT` and `DELETE` commands to sequences of instrumented commands.

Expansion procedure. Finally, the expansion procedure is shown in Figures 7.19 and 7.20. In the figures, s is a database state, m is a memory, u is a user identifier, and $x \leftarrow q$ is an SQL command. Without loss of generality, we assume that $x \notin free(q)$ (if this is not the case, one can just introduce an additional temporary variable). Furthermore, given a list $c_1{\cdot}\ldots{\cdot}c_n$ of WhileSql statements, we denote by $;(c_1{\cdot}\ldots{\cdot}c_n)$ the statement $c_1; c_2; \ldots; c_n$. Similarly, given a list of WhileSql expressions $e_1{\cdot}\ldots{\cdot}e_n$, we denote by $\wedge(e_1{\cdot}\ldots{\cdot}e_n)$ the expression $e_1 \wedge e_2 \wedge \ldots \wedge e_n$.

$$allowed(s, u, \texttt{SELECT}\ \phi) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u, \texttt{SELECT}, \phi\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u, \texttt{SELECT}, \phi\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s, u, \texttt{GRANT}\ p\ \texttt{TO}\ u') = \begin{cases} \top & \text{if } s \xrightarrow{\langle \oplus, u, p, u'\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle \oplus, u, p, u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s, u, \texttt{GRANT}\ p\ \texttt{TO}\ u'\ \texttt{WITH GRANT OPTION}) = \begin{cases} \top & \text{if } s \xrightarrow{\langle \oplus^*, u, p, u'\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle \oplus^*, u, p, u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s, u, \texttt{REVOKE}\ p\ \texttt{FROM}\ u') = \begin{cases} \top & \text{if } s \xrightarrow{\langle \ominus, u, p, u'\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle \ominus, u, p, u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s, u, \texttt{INSERT}\ \overline{v}\ \texttt{INTO}\ T) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u, \texttt{INSERT}, T, \overline{v}\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u, \texttt{INSERT}, T, \overline{v}\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s, u, \texttt{DELETE}\ \overline{v}\ \texttt{FROM}\ T) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u, \texttt{DELETE}, T, \overline{v}\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u, \texttt{DELETE}, T, \overline{v}\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s, u, \texttt{CREATE}\ obj) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u, \texttt{CREATE}, obj\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u, \texttt{CREATE}, obj\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s, u, \texttt{ADD USER}\ u') = \begin{cases} \top & \text{if } s \xrightarrow{\langle u, \texttt{ADD\_USER}, u'\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u, \texttt{ADD\_USER}, u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$allowed(s, u, t) = allowed(s, owner(t), act(t))$ if t is a trigger with owner's privileges

$allowed(s, u, t) = allowed(s, invoker(t), act(t)) \wedge allowed(s, owner(t), act(t))$
if t is a trigger with activator's privileges

FIGURE 7.16: *allowed* function for the expansion process.

$apply(s, u, \texttt{SELECT}\ \phi) = s$
$apply(s, u, \texttt{GRANT}\ p\ \texttt{TO}\ u') = s'$ where $s \xrightarrow{\langle \oplus, u', p, u\rangle}_f s'$
$apply(s, u, \texttt{GRANT}\ p\ \texttt{TO}\ u'\ \texttt{WITH GRANT OPTION}) = s'$ where $s \xrightarrow{\langle \oplus^*, u', p, u\rangle}_f s'$
$apply(s, u, \texttt{REVOKE}\ p\ \texttt{FROM}\ u') = s'$ where $s \xrightarrow{\langle \ominus, u', p, u\rangle}_f s'$
$apply(s, u, \texttt{INSERT}\ \overline{v}\ \texttt{INTO}\ T) = s$
$apply(s, u, \texttt{DELETE}\ \overline{v}\ \texttt{FROM}\ T) = s$
$apply(s, u, \texttt{CREATE}\ obj) = s'$ where $s \xrightarrow{\langle u, \texttt{CREATE}, obj\rangle}_f s'$
$apply(s, u, \texttt{ADD USER}\ u') = s$ where $s \xrightarrow{\langle u, \texttt{ADD\_USER}, u'\rangle}_f s'$
$apply(s, u, t) = apply(s, user(t), act(t))$ if t is a trigger

FIGURE 7.17: *apply* function for the expansion process. Note that we are interested only in changes to the database configuration, not to the database state. Therefore, the function does not update the database on `INSERT` and `DELETE` commands.

$wp(\phi, \epsilon) = \phi$

$wp(\phi, \overline{t'} \cdot ic) = wp(wp(\phi, ic), \overline{t'})$

where ic is an instrumented command and $\overline{t'}$ is a sequence of instrumented commands

$wp(\phi, \langle \texttt{INSERT } T \texttt{ INTO } \overline{e}, \texttt{ok} \rangle) = wp(\phi, T \oplus \overline{e})$

$wp(\phi, \langle \texttt{INSERT } T \texttt{ INTO } \overline{e}, \texttt{ex} \rangle) = wp(\phi, T \oplus \overline{e})$

$wp(\phi, \langle \texttt{INSERT } T \texttt{ INTO } \overline{e}, \texttt{secEx} \rangle) = \phi$

$wp(\phi, \langle \texttt{DELETE } T \texttt{ FROM } \overline{e}, \texttt{ok} \rangle) = wp(\phi, T \ominus \overline{e})$

$wp(\phi, \langle \texttt{DELETE } T \texttt{ FROM } \overline{e}, \texttt{ex} \rangle) = wp(\phi, T \ominus \overline{e})$

$wp(\phi, \langle \texttt{DELETE } T \texttt{ FROM } \overline{e}, \texttt{secEx} \rangle) = \phi$

$wp(\phi, \langle c, r \rangle) = \phi$ where c is a `SELECT`, `GRANT`, `REVOKE`, `CREATE`, or `ADD USER` command

$wp(\phi, \langle t, \texttt{ok} \rangle) = wp(\phi, act(t))$ where t is a trigger

$wp(\phi, \langle t, \texttt{ex} \rangle) = wp(\phi, act(t))$ where t is a trigger

$wp(\phi, \langle t, \texttt{secEx} \rangle) = \phi$ where t is a trigger

$wp(\phi, \langle t, \texttt{dis} \rangle) = \phi$ where t is a trigger

FIGURE 7.18: Weakest precondition for sequences of instrumented commands.

Expansion Procedure.

$$expand(s, m, u, x \leftarrow q) = decls(s, m, u, x \leftarrow q)\ ;\ body(s, m, u, x \leftarrow q)$$

Shorthands.

$\overline{t}_q = triggers(s, u, q)$ $\qquad$ $first(\langle a, b \rangle) = a$ $\qquad$ $\overline{EP}_{s,m,u,q} = toList(consPaths(q \cdot \overline{t}_q, s, u))$

$\Gamma = \{\gamma_1, \ldots, \gamma_n\}$ are the integrity constraints $\qquad$ $\overline{L_\Gamma} = toList(2^{\{\gamma_1, \ldots, \gamma_n\}})$

$throwsEx(\overline{t}) = \exists o.\ (\overline{t}(|\overline{t}|) = \langle o, \texttt{ex} \rangle \vee \overline{t}(|\overline{t}|) = \langle o, \texttt{secEx} \rangle)$ $\qquad$ $secEx(\overline{t}) = \exists o.\ \overline{t}(|\overline{t}|) = \langle o, \texttt{secEx} \rangle$

$isCfgCmd(q) = \top$ iff q is an `GRANT`, `REVOKE`, `ADD USER`, or `CREATE` command

$isInsDel(q) = \top$ iff q is an `INSERT` or `DELETE` command

$isSelect(q) = \top$ iff q is a `SELECT` command

$isTrigger(o) = \top$ iff o is a trigger

Computing the expansion's auxiliary declarations.

$decls(s, m, u, x \leftarrow q) = \texttt{;}(map(g_{s,m,u,q}, \overline{EP}_{s,m,u,q}))$

$g_{s,m,u,q}(\overline{t}) = h_{s,m,u,q,\overline{t}}(1); \ldots; h_{s,m,u,q,\overline{t}}(|\overline{t}|)$

$h_{s,m,u,q,\overline{t}}(i) = \|x^{\overline{t}}_{i,\gamma_1} \leftarrow \texttt{SELECT } wp(\gamma_1, \overline{t}^{\,i})\|; \ldots; \|x^{\overline{t}}_{i,\gamma_n} \leftarrow \texttt{SELECT } wp(\gamma_n, \overline{t}^{\,i})\|$

where $\overline{t}(i)$ is not a trigger and $x^{\overline{t}}_{i,\gamma_1}, \ldots, x^{\overline{t}}_{i,\gamma_n}$ are fresh variables

$h_{s,m,u,q,\overline{t}}(i) = \|x^{\overline{t}}_{i,cond} \leftarrow \texttt{SELECT } wp(\varphi, \overline{t}^{\,i-1})\|; \|x^{\overline{t}}_{i,\gamma_1} \leftarrow \texttt{SELECT } wp(\gamma_1, \overline{t}^{\,i})\|; \ldots; \|x^{\overline{t}}_{i,\gamma_n} \leftarrow \texttt{SELECT } wp(\gamma_n, \overline{t}^{\,i})\|$

where $\overline{t}(i)$ is a trigger, φ is $\overline{t}(i)$'s condition, and $x^{\overline{t}}_{i,cond}, x^{\overline{t}}_{i,\gamma_1}, \ldots, x^{\overline{t}}_{i,\gamma_n}$ are fresh variables

FIGURE 7.19: Expansion process – 1.

Computing the expansion's body.

$body(s, m, u, x \leftarrow q) = \mathbf{;}(map(d_{s,m,u,x,q}, \overline{EP}_{s,m,u,x,q}))$

$d_{s,m,u,x,q}(\overline{t}) = \mathbf{if}\ cond_{s,m,u,x,q}(\overline{t})\ \mathbf{then}\ body_{s,m,u,x,q}(\overline{t})\ \mathbf{else\ skip}$

$cond_{s,m,u,x,q}(\overline{t}) = \wedge(map(c_{s,m,u,x,q,\overline{t}}, 1 \cdot \ldots \cdot |\overline{t}|))$

$c_{s,m,u,x,q,\overline{t}}(i) = x^{\overline{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{i,\gamma_n}$ where $\overline{t}(i) = \langle c, \texttt{ok}\rangle$ and c is not a trigger

$c_{s,m,u,x,q,\overline{t}}(i) = \neg(x^{\overline{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{i,\gamma_n})$ where $\overline{t}(i) = \langle c, \texttt{ex}\rangle$ and c is not a trigger

$c_{s,m,u,x,q,\overline{t}}(i) = \top$ where $\overline{t}(i) = \langle c, \texttt{secEx}\rangle$ and c is not a trigger

$c_{s,m,u,x,q,\overline{t}}(i) = x^{\overline{t}}_{i,cond} \wedge x^{\overline{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{i,\gamma_n}$ where $\overline{t}(i) = \langle t, \texttt{ok}\rangle$ and t is a trigger

$c_{s,m,u,x,q,\overline{t}}(i) = x^{\overline{t}}_{i,cond} \wedge \neg(x^{\overline{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{i,\gamma_n})$ where $\overline{t}(i) = \langle t, \texttt{ex}\rangle$ and t is a trigger

$c_{s,m,u,x,q,\overline{t}}(i) = x^{\overline{t}}_{i,cond}$ where $\overline{t}(i) = \langle t, \texttt{secEx}\rangle$ and t is a trigger

$c_{s,m,u,x,q,\overline{t}}(i) = \neg x^{\overline{t}}_{i,cond}$ where $\overline{t}(i) = \langle t, \texttt{dis}\rangle$ and t is a trigger

$$body_{s,m,u,x,q}(\overline{t}) = \begin{cases} x = \langle \mathbf{SecEx}, \emptyset\rangle; o_{s,m,u,x,q}(\overline{t}) & \text{if } \overline{t}(1) = \langle o, \texttt{secEx}\rangle \text{ and } o \text{ is not a trigger} \\ x = \langle id(first(\overline{t}(|\overline{t}|))), \mathbf{SecEx}, \emptyset\rangle\ ;\ o_{s,m,u,x,q}(\overline{t}) & \text{if } \overline{t}(|\overline{t}|) = \langle o, \texttt{secEx}\rangle \text{ and } o \text{ is a trigger} \\ \mathbf{;}(map(e_{s,m,u,x,q,\overline{t}}, \overline{L_\Gamma})) & \text{if } \overline{t}(|\overline{t}|) = \langle o, \texttt{ex}\rangle \\ \mathbf{;}\,(map(b_{s,m,u,x,q,\overline{t}}, 1 \cdot \ldots \cdot |\overline{t}|))\ ;\ o_{s,m,u,x,q}(\overline{t}) & \text{otherwise} \end{cases}$$

$b_{s,m,u,x,q,\overline{t}}(i) = \|x \leftarrow T \oplus \overline{e}\|$ where $\overline{t}(i) = \langle \texttt{INSERT}\ T\ \texttt{INTO}\ \overline{e}, \texttt{ok}\rangle$

$b_{s,m,u,x,q,\overline{t}}(i) = \|x \leftarrow T \ominus \overline{e}\|$ where $\overline{t}(i) = \langle \texttt{DELETE}\ T\ \texttt{FROM}\ \overline{e}, \texttt{ok}\rangle$

$b_{s,m,u,x,q,\overline{t}}(i) = \|x \leftarrow q'\|$ where $\overline{t}(i) = \langle q', \texttt{ok}\rangle$, $isTrigger(q') = \bot$, and $isInsDel(q') = \bot$

$b_{s,m,u,x,q,\overline{t}}(i) = \mathbf{asuser}(user(t), \|y \leftarrow T \oplus \overline{e}\|)$ where $\overline{t}(i) = \langle t, \texttt{ok}\rangle$, t is a trigger, $act(t) = \texttt{INSERT}\ T\ \texttt{INTO}\ \overline{e}$, and y is a fresh variable

$b_{s,m,u,x,q,\overline{t}}(i) = \mathbf{asuser}(user(t), \|y \leftarrow T \ominus \overline{e}\|)$ where $\overline{t}(i) = \langle t, \texttt{ok}\rangle$, t is a trigger, $act(t) = \texttt{DELETE}\ T\ \texttt{FROM}\ \overline{e}$, and y is a fresh variable

$b_{s,m,u,x,q,\overline{t}}(i) = \mathbf{asuser}(user(t), \|y \leftarrow act(t)\|)$ where $\overline{t}(i) = \langle t, \texttt{ok}\rangle$, t is a trigger, $isInsDel(act(t)) = \bot$, and y is a fresh variable

$b_{s,m,u,x,q,\overline{t}}(i) = \mathbf{skip}$ where $\overline{t}(i) = \langle t, \texttt{dis}\rangle$

$$e_{s,m,u,x,q,\overline{t}}(\Theta) = \mathbf{if} \bigwedge_{\gamma \in \Theta} \neg x^{\overline{t}}_{|\overline{t}|,\gamma} \wedge \bigwedge_{\gamma \in \Gamma \setminus \Theta} x^{\overline{t}}_{|\overline{t}|,\gamma}\ \mathbf{then}\ e'_{s,m,u,x,q,\overline{t}}(\Theta)\ \mathbf{else\ skip}$$

$$e'_{s,m,u,x,q,\overline{t}}(\Theta) = \begin{cases} \mathbf{dbout}(db(u), q, \langle \mathbf{IntEx}, \Theta\rangle, extr(\overline{t})); x = \langle \mathbf{IntEx}, \Theta\rangle & \text{if } |\overline{t}| = 1 \\ \mathbf{dbout}(db(u), q, \langle id(first(\overline{t}(|\overline{t}|))), \mathbf{IntEx}, \Theta\rangle, extr(\overline{t})); \\ \quad x = \langle id(first(\overline{t}(|\overline{t}|))), \mathbf{IntEx}, \Theta\rangle & \text{if } |\overline{t}| > 1 \end{cases}$$

$$o_{s,m,u,x,q}(\overline{t}) = \begin{cases} \mathbf{dbout}(db(u), q, \top, extr(\overline{t})) & \text{if } isInsDel(q) \wedge \neg throwsEx(\overline{t}) \\ \mathbf{dbout}(db(u), q, \langle \mathbf{SecEx}, \emptyset\rangle, extr(\overline{t})) & \text{if } isInsDel(q) \wedge secEx(\overline{t}) \wedge |\overline{t}| = 1 \\ \mathbf{dbout}(db(u), q, \langle id(first(\overline{t}(|\overline{t}|))), \mathbf{SecEx}, \emptyset\rangle, extr(\overline{t})) & \text{if } isInsDel(q) \wedge secEx(\overline{t}) \wedge |\overline{t}| > 1 \\ \mathbf{dbout}(public, q, x, \epsilon) & \text{if } isCfgCmd(q) \wedge \neg throwsEx(\overline{t}) \\ \mathbf{dbout}(db(u), q, x, \epsilon) & \text{if } isCfgCmd(q) \wedge throwsEx(\overline{t}) \\ \mathbf{dbout}(db(u), q, x, \epsilon) & \text{if } isSelect(q) \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$extr(\epsilon) = \epsilon$

$$extr(\langle t, r\rangle \cdot \overline{t}) = \begin{cases} \langle public, id(t), act(t)\rangle \cdot extr(\overline{t}) & \text{if } isTrigger(t) \wedge r = \texttt{ok} \wedge isCfgCmd(act(t)) \\ extr(\overline{t}) & \text{otherwise} \end{cases}$$

FIGURE 7.20: Expansion process – 2.

Part IV

Conclusions

Chapter 8

Conclusion

I hate endings. Just detest them. Beginnings are definitely the most exciting, middles are perplexing and endings are a disaster. ... The temptation towards resolution, towards wrapping up the package, seems to me a terrible trap. Why not be more honest with the moment? The most authentic endings are the ones which are already revolving towards another beginning. That's genius.

Sam Shepard

Preventing disclosure of sensitive data is an extremely challenging task. It often requires a careful configuration of multiple enforcement mechanisms. It may also involve reasoning about security at different levels, from restricting access to the database's content to enforcing end-to-end security requirements on the applications interacting with the database. Therefore, insecure systems may result from mistakes introduced during several phases of the security engineering process. For instance, security vulnerabilities may be introduced by inaccurate security requirements, by misconfigured enforcement mechanisms, or by mechanisms that fail at providing the desired security guarantees.

In this context, provably secure enforcement mechanisms represent an indispensable component for database security. By providing precise security guarantees and provably preventing attacks, they significantly simplify the process of securing databases. Indeed, provably secure mechanisms free the security engineers from the difficult task of verifying whether a given mechanism, which is often an off-the-shelf component, correctly provides the desired security guarantees. Hence, the security engineers can focus on their main duties, namely eliciting the security requirements and formalizing them using security policies.

In this thesis, we developed solid foundations for access control and inference control for database systems. We used these foundations to build and verify practical and provably secure enforcement mechanisms. Specifically, we investigated DBAC and DBIC with respect to three different classes of attackers of increasing strength. First, we provided theoretical results clarifying the exact limits of DBAC with respect to `SELECT`-only attackers. Second, we designed a practical and provably secure enforcement mechanism that secures databases against `SELECT`-only attackers with probabilistic reasoning capabilities. Finally, we developed a framework for reasoning about database security against attackers that execute `SELECT` queries, modify the database's content and configuration, and exploit advanced database features like triggers and views. We used this framework to build and verify a provably secure enforcement mechanism for this class of attackers. We also used it as a starting point for bridging DBAC with application-level end-to-end security in the form of IFC, and we showed that each area of research can benefit from reusing concepts and techniques developed by the other.

The work in this thesis opens up a number of future research directions, which we outline below.

Efficient algorithms for probabilistic inference. As shown by our work in Chapter 4, dedicated inference engines are an essential component for building practical enforcement mechanisms that effectively secure databases against selected classes of attackers, as the runtime complexity of general-purpose inference is intractable. In this respect, there are two main open research directions:

1. Tractable inference procedures are needed for other fragments of ProbLog. This would allow security engineers to model larger classes of attacker models. We remark that there are various classes of probabilistic models with tractable inference other than poly-tree BNs. For instance, inference runs in Ptime in the network's size also for Bayesian Networks with bounded junction-trees [108]. While bounded junction-trees Bayesian Networks cannot be encoded as acyclic ProbLog programs, an approach similar to the one we adopted in Chapter 4, i.e., dedicated syntactic criteria together with a compilation to Bayesian Networks, may be used to derive other classes of ProbLog programs where inference is tractable.

2. Approximate inference algorithms are needed to handle programs that do not belong to fragments with efficient inference procedures. To be useful for security purposes, an approximate algorithm should return an interval $[a, b]$ containing the actual probability, as this would allow an enforcement mechanism to check both lower and upper bounds on an attacker's belief. While naive approximate algorithms often have a very low precision, further research on the topic may lead to approximate algorithms with a reasonable efficiency and precision. A particularly interesting research direction here is investigating how approximate algorithms may benefit from exact algorithms, e.g., by reusing exact algorithms as sub-routines while analyzing portions of the overall program.

From a security standpoint, another open challenge is extending our framework to dynamic settings where the database and the policy change over time. In this respect, one would need to (1) lift attacker beliefs from databases states to traces, and (2) specify how operations modifying the database's content and configuration affect these beliefs.

Attacker models as first-class citizens. We developed the idea that attacker models should be studied and formalized for databases. Rather than being implicit, the relevant models must be made explicit, just like when analyzing security in other domains. The `SELECT`-only attacker model, the class of attackers expressible using AtkLog, and the attacker model we developed in Chapter 6 are just a few examples of attackers. Reasoning about database security requires, however, a large library of attacker models, each one capturing an attacker with different capabilities (attackers that can interact only through applications, attackers with direct database access, attackers with external knowledge, and so on). Studying and formalizing realistic classes of attackers is an important research direction for database security, which unfortunately has received only little attention so far. We remark that this library of attackers should be complemented with a family of provably secure enforcement mechanisms. In this way, a security engineer could just select an enforcement mechanism that provides the desired guarantees against a given attacker model.

Integration between DBAC and IFC. Our work in Chapter 7 opens research opportunities for both access control and information-flow control. For DBAC, our reduction allows the development of provably secure DBAC mechanisms starting from IFC solutions. From a theoretical standpoint, an interesting open problem is extending our reduction to work for larger fragments of SQL. On a more practical side, it would be interesting to use our reduction to construct practical DBAC mechanisms using state-of-the-art IFC techniques. For IFC, instead, it would be interesting to study how other common IFC techniques, such as secure type systems or secure multi-executions, could be extended and benefit from a closer integration with databases.

Part V

Appendices

Appendix A

Proofs for Chapter 3

Proof of Theorem 3.3. We prove the decidability of $AGREE^{ERC}$ using Lemma 3.1, where the fragments are $F = ERC$ and $F' = BSRRC$. We divide the proof in four steps. First, we define some preliminary notation. Next, we provide a formula $\phi'_{INDIST(S,db)}$ not in the $BSRRC$ fragment that encodes the indistinguishability relation. Afterwards, we show that $\phi'_{INDIST(S,db)}$ can be equivalently rewritten as a $BSRRC$-formula $\phi_{INDIST(S,db)}$. Finally, we prove that the conditions 2 and 3 of Lemma 3.1 are met, and therefore we derive the decidability result.

Step 1. Let D be a database schema, S be an ERC-security policy over D, and db be a state in Ω_D. Let $\bar{t}$ be a tuple, and $\overline{y} = \langle y_1, \ldots, y_n \rangle$ be a tuple of free variables such that $|\bar{t}| = |\overline{y}|$. The formula $\theta_{q,S}(\overline{y}, \bar{t})$, where S is an ERC-policy, q is a non-boolean ERC-query such that there is a constraint for q in S, $\overline{x}$ is a tuple of variables, and $\bar{t}$ is a tuple of values in **dom** such that $|\overline{y}| = |\bar{t}|$, is defined as follows:

$$\theta_{\{\overline{x}|\psi\},S}(\overline{y},\bar{t}) \;:=\; \bigwedge_{\substack{i\in\{1,\ldots,|\bar{t}|\}\\ \wedge \bar{t}(i)\neq\dagger}} (\psi^S_{q,i}[\overline{x} \;\mapsto\; \overline{y}] \;\wedge\; \overline{y}(i) \;=\; \bar{t}(i)) \;\wedge \bigwedge_{\substack{i\in\{1,\ldots,|\bar{t}|\}\\ \wedge \bar{t}(i)=\dagger}} \neg\psi^S_{q,i}[\overline{x} \;\mapsto\; \overline{y}].$$

The formula $\theta_{q,S}(\overline{y}, \bar{t})$ enforces that the tuple $\overline{v}$ associated with the values of the variables $\overline{x}$ is such that $mask_{S,s,q}(\overline{v}) = \bar{t}$, where q is a non-boolean ERC-query.

Given a tuple of variables $\overline{x}$, we denote by $\overline{y}_{\overline{x}}$ the tuple of variables $y_1, \ldots, y_{|\overline{x}|}$.

Step 2. The formula $\phi'_{INDIST(S,db)}$ encoding the indistinguishability relation is as follows:

$$\begin{aligned}
\psi_{S,\{\overline{x}|\psi\},db} &:= \exists^{=|Auth_{S,\{\overline{x}|\psi\}}(db)|}\overline{y}_{\overline{x}}.\ (\psi[\overline{x}\mapsto\overline{y}_{\overline{x}}] \wedge \psi^S_{\{\overline{x}|\psi\}}[\overline{x}\mapsto\overline{y}_{\overline{x}}])\\
\gamma_{\{\overline{x}|\psi\},S,db,\bar{t}} &:= \exists^{\geq card_{S,db,\{\overline{x}|\psi\}}(\bar{t})}\overline{y}_{\overline{x}}.\ (\psi[\overline{x}\mapsto\overline{y}_{\overline{x}}] \wedge \psi^S_{\{\overline{x}|\psi\}}[\overline{x}\mapsto\overline{y}_{\overline{x}}] \wedge \theta_{\{\overline{x}|\psi\},S}(\overline{y}_{\overline{x}},\bar{t}))\\
\phi'_{INDIST(S,db)} &:= \bigwedge_{\langle q,\phi\rangle\in ROW} (\psi_{S,q,db} \wedge \bigwedge_{\bar{t}\in Ind_{S,q}(db)} \gamma_{q,S,db,\bar{t}}).
\end{aligned}$$

Observe that $|Auth_{S,q}(db)| = \Sigma_{\bar{t}\in Ind_{S,q}(db)} card_{S,db,q}(\bar{t})$ for any security policy S and any non-boolean query q such that S contains a constraint for q. Therefore, although in $\gamma_{q,S,db,\bar{t}}$ we used counting quantifiers with the $\geq$ operator, the whole formula $\phi'_{INDIST(S,db)}$ can be satisfied iff all the counting quantifiers are satisfied exactly with equality. Furthermore, note also that if $S = \langle\emptyset,\emptyset\rangle$, then $\phi'_{INDIST(S,db)} := \top$. The formula $\phi'_{INDIST(S,db)}$ encodes the indistinguishability relation. The encoding, however, is not in the $BSRRC$ fragment.

Correctness: We now prove the correctness of the encoding $\phi'_{INDIST(S,db)}$. Let D be a database schema, $S = \langle ROW, COL\rangle$ be a security policy, and db, db' be two states in Ω_D. We now show that $[\phi'_{INDIST(S,db)}]^{db'} = \top$ iff $db \cong_S db'$. Without loss of generality, we assume that $S \neq \langle\emptyset,\emptyset\rangle$. If this is not the case, the claim trivially holds.

($\Rightarrow$). We show that $[\phi'_{INDIST(S,db)}]^{db'} = \top$ implies $db \cong_S db'$. Let $\langle q,\phi\rangle \in ROW$ be a constraint. From $[\phi'_{INDIST(S,db)}]^{db'} = \top$, it follows that $[\psi_{S,q,db} \wedge \bigwedge_{\bar{t}\in Ind_{S,q}(db)} \gamma_{q,S,db,\bar{t}}]^{db'} = \top$. From $[\psi_{S,q,db}]^{db'} = \top$, it follows that $|Auth_{S,q}(db)| = |Auth_{S,q}(db')|$. Let $\bar{t}$ be a tuple in $Ind_{S,q}(db)$. From $[\gamma_{q,S,db,\bar{t}}]^{db'} = \top$, it follows that there are at least $card_{S,db,q}(\bar{t})$ tuples $\overline{v}$ in $Auth_{S,q}(db')$ such that $mask_{S,db',q}(\overline{v}) = \bar{t}$. Note also that, as said before, $|Auth_{S,q}(db)| = \Sigma_{\bar{t}\in Ind_{S,q}(db)} card_{S,db,q}(\bar{t})$. Therefore, for each $\bar{t} \in Ind_{S,q}(db)$, there are exactly $card_{S,db,q}(\bar{t})$ tuples $\overline{v}$ in $Auth_{S,q}(db')$ such that $mask_{S,db',q}(\overline{v}) = \bar{t}$. From this, it follows that there is a bijection f from $Auth_{S,q}(db)$ to $Auth_{S,q}(db')$ such that $mask_{S,db,q}(\bar{t}) = mask_{S,db',q}(f(\bar{t}))$ for all $\bar{t} \in Auth_{S,q}(db)$. Hence, $[\phi_{INDIST(S,db)}]^{db'} = \top$ implies $db \cong_S db'$.

($\Leftarrow$). We show that $db \cong_S db'$ implies $[\phi'_{INDIST(S,db)}]^{db'} = \top$. Let $\langle q,\phi\rangle \in ROW$ be a constraint. From $db \cong_S db'$, it follows that $|Auth_{S,q}(db)| = |Auth_{S,q}(db')|$, and therefore $[\psi_{S,q,db}]^{db'} = \top$. From $db \cong_S db'$, it also follows that for each $\bar{t} \in Ind_{S,q}(db)$, there are exactly $card_{S,db,q}(\bar{t})$ tuples $\overline{v}$ in

$Auth_{S,q}(db')$ such that $mask_{S,db,q}(\overline{v}) = \overline{t}$, and thus $[\bigwedge_{\overline{t} \in Ind_{S,q}(db)} \gamma_{q,S,db,\overline{t}}]^{db'} = \top$. From $[\psi_{S,q,db}]^{db'} = \top$ and $[\bigwedge_{\overline{t} \in Ind_{S,q}(db)} \gamma_{q,S,db,\overline{t}}]^{db'} = \top$, it follows that $[\phi'_{INDIST(S,db)}]^{db'} = \top$. Hence, $db \cong_S db'$ implies $[\phi'_{INDIST(S,db)}]^{db'} = \top$.

Step 3. We now show that there is a formula $\phi_{INDIST(S,db)} \in BSRRC$ that is equivalent to $\phi'_{INDIST(S,db')}$. Let D be a database schema, S be a security policy over D, s be a state, q be a non-boolean query such that there is a constraint on q in S, and $\overline{t}$ be a tuple. We first show that there is a $BSRRC$ formula equivalent to $\psi_{S,q,db}$. Afterwards, we show that there is a $BSRRC$ formula equivalent to $\gamma_{q,S,db,\overline{t}}$. Finally, we show that, under some weak assumptions, $\phi \wedge \psi \in BSRRC$ if $\phi, \psi \in BSRRC$. In the following, we denote logical equivalence by $\equiv$. Our proof mainly exploits the equivalence $(Q\ x.\ \phi)\ op\ \psi \equiv Q\ x.\ (\phi\ op\ \psi)$, where $Q \in \{\forall, \exists\}$ and $op \in \{\wedge, \vee\}$, if x is not a free variable in ψ. Without loss of generality, we assume that the quantified variables used in the authorization constraints and in the queries have unique variable identifiers.

We now show that all the formulae of the form of $\psi_{S,q,db}$ can be equivalently rewritten as $BSRRC$ formulae if S is an ERC-policy. Let m be a natural number and $\exists \overline{z}.\ \psi(\overline{z}, \overline{x})$ and $\exists \overline{y}.\ \phi(\overline{y}, \overline{x})$ be two ERC formulae. Figure A.1 shows that all the formulae of the form of $\psi_{S,q,s}$ can be equivalently rewritten as $BSRRC$ formulae if S is an ERC-policy.[1] Note that the last formula in Figure A.1 is in the $BSRRC$ fragment. Since $\psi_{S,q,s}$ has the same form as the formula ψ used in the derivation, it follows that $\psi_{S,q,db}$ can be equivalently rewritten as a $BSRRC$-formula.

Let ψ and ϕ be two formulae in the $BSRRC$ fragment such that the free variables of ψ are not used as quantified variables in ϕ and vice versa. Then, $\psi \wedge \phi$ is in $BSRRC$ as shown by the following derivation.

$$\begin{aligned}
\psi \wedge \phi &= \exists \overline{x}.\forall \overline{y}.\psi'(\overline{x}, \overline{y}, \overline{z}) \wedge \exists \overline{r}.\forall \overline{s}.\phi'(\overline{r}, \overline{s}, \overline{z}) \\
&\equiv \exists \overline{x}.(\forall \overline{y}.\psi'(\overline{x}, \overline{y}, \overline{z}) \wedge \exists \overline{r}.\forall \overline{s}.\phi'(\overline{r}, \overline{s}, \overline{z})) \\
&\equiv \exists \overline{x}.\exists \overline{r}.(\forall \overline{y}.\psi'(\overline{x}, \overline{y}, \overline{z}) \wedge \forall \overline{s}.\phi'(\overline{r}, \overline{s}, \overline{z})) \\
&\equiv \exists \overline{x}.\exists \overline{r}.\forall \overline{y}.(\psi'(\overline{x}, \overline{y}, \overline{z}) \wedge \forall \overline{s}.\phi'(\overline{r}, \overline{s}, \overline{z})) \\
&\equiv \exists \overline{x}.\exists \overline{r}.\forall \overline{y}.\forall \overline{s}.(\psi'(\overline{x}, \overline{y}, \overline{z}) \wedge \phi'(\overline{r}, \overline{s}, \overline{z})) \\
&\equiv \exists \overline{x}, \overline{r}.\forall \overline{y}, \overline{s}.(\psi'(\overline{x}, \overline{y}, \overline{z}) \wedge \phi'(\overline{r}, \overline{s}, \overline{z}))
\end{aligned}$$

In a similar way, we can prove that $\psi \vee \phi$ is in the $BSRRC$ fragment for all $\psi, \phi \in BSRRC$ such that the free variables of ψ are not used as quantified variables in ϕ and vice versa.

To prove that $\gamma_{q,S,s,\overline{t}}$ can be rewritten as an equivalent $BSRRC$ formula, we first show, in Figure A.2, that given an ERC-security policy S, all formulae of the form of $\theta_{q,S}(\overline{y}, \overline{t})$ can be equivalently rewritten as $BSRRC$ formulae with free variables $\overline{y}$. The last formula in Figure A.2 is in the $BSRRC$ fragment. Observe that ψ, ψ_q^S, and $\theta_{q,S}(\overline{y}, \overline{t})$ are $BSRRC$ formulae, and that the quantified variables in one of the formulae are not free in the others (and vice versa). The conjunction of them is again a $BSRRC$ formula $\delta(\overline{y})$, and therefore

$$\begin{aligned}
\gamma_{q,S,db,\overline{t}} &= \exists^{\geq m} \overline{y}.\delta(\overline{y}) \\
&\equiv \exists \overline{y}_1, \ldots, \overline{y}_m.(\bigwedge_{i \in \{1,\ldots,m\}} \delta(\overline{y}_i) \wedge \bigwedge_{i,j \in \{1,\ldots,m\} \wedge i \neq j} \overline{y}_i \neq \overline{y}_j)
\end{aligned}$$

Both $\bigwedge_{i \in \{1,\ldots,m\}} \delta(t_i)$ and $\bigwedge_{i,j \in \{1,\ldots,m\} \wedge i \neq j} \overline{y}_i \neq \overline{y}_j$ are $BSRRC$ formulae (and quantified variables in one of the formulae are not free variables in the other one), and therefore also their conjunction is a $BSRRC$ formula. Moreover, since a $BSRRC$ formula is of the form $\exists \overline{x}.\forall \overline{y}.\phi(\overline{x}, \overline{y}, \overline{z})$, then also the formula $\exists \overline{z}_1, \ldots, \overline{z}_m.\exists \overline{x}.\forall \overline{y}.\bigwedge_i \phi(\overline{x}, \overline{y}, \overline{z}_i)$ is a $BSRRC$ formula, and therefore $\gamma_{q,S,db,\overline{t}} \in BSRRC$.

Since $\psi_{S,q,db}$ and $\gamma_{q,S,db,\overline{t}}$ can be equivalently rewritten as $BSRRC$ formulae, and since $\phi'_{INDIST(S,db)}$ is obtained from $\psi_{S,q,db}$ and $\gamma_{q,S,db,\overline{t}}$ only by conjunctions, there is a $BSRRC$-formula $\phi_{INDIST(S,db)}$ that is equivalent to $\phi'_{INDIST(S,db)}$ in the $BSRRC$ fragment.

Step 4. Finally, we prove that for any ERC formula ψ and any ERC-policy S then:

1. $\phi_{INDIST(S,db)} \wedge \psi \in BSRRC$, and
2. $\phi_{INDIST(S,db)} \wedge \neg\psi \in BSRRC$.

The first case is simple since both $\phi_{INDIST(S,db)}$ and ψ are $BSRRC$ sentences, and therefore also their conjunction can be rewritten as a $BSRRC$ sentence. In the second case we can apply the following

[1] In case $m = 0$, the resulting formula $\exists^{=0} \overline{x}.\ (\exists \overline{y}.\phi(\overline{y}, \overline{x}))$ can be rewritten as an equivalent formula of the form $\forall \overline{x}, \overline{y}.\ \neg\phi(\overline{y}, \overline{x})$ which is in the $BSRRC$ fragment.

$$
\begin{aligned}
\psi &= \exists^{=m}\overline{x}.(\exists\overline{z}.\psi(\overline{z},\overline{x}) \wedge \exists\overline{y}.\phi(\overline{y},\overline{x})) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\exists\overline{z}.\psi(\overline{z},\overline{x}_i) \wedge \exists\overline{y}.\phi(\overline{y},\overline{x}_i)) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad \forall\overline{x}_{m+1}.(\neg(\exists\overline{z}.\psi(\overline{z},\overline{x}_{m+1}) \wedge \exists\overline{y}.\phi(\overline{y},\overline{x}_{m+1})) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1})) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m.(\exists\overline{y}_1,\ldots,\overline{y}_m,\overline{z}_1,\ldots,\overline{z}_m. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\psi(\overline{z}_i,\overline{x}_i) \wedge \phi(\overline{y}_i,\overline{x}_i))) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad \forall\overline{x}_{m+1}.(\neg(\exists\overline{z}.\psi(\overline{z},\overline{x}_{m+1}) \wedge \exists\overline{y}.\phi(\overline{y},\overline{x}_{m+1})) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1})) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m.(\exists\overline{y}_1,\ldots,\overline{y}_m,\overline{z}_1,\ldots,\overline{z}_m. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\psi(\overline{z}_i,\overline{x}_i) \wedge \phi(\overline{y}_i,\overline{x}_i))) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad \forall\overline{x}_{m+1}.(\forall\overline{z}.\neg\psi(\overline{z},\overline{x}_{m+1}) \vee \forall\overline{y}.\neg(\phi(\overline{y},\overline{x}_{m+1})) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1})) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m.(\exists\overline{y}_1,\ldots,\overline{y}_m,\overline{z}_1,\ldots,\overline{z}_m. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\psi(\overline{z}_i,\overline{x}_i) \wedge \phi(\overline{y}_i,\overline{x}_i))) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad \forall\overline{x}_{m+1}.(\forall\overline{y},\overline{z}.(\neg\psi(\overline{z},\overline{x}_{m+1}) \vee \neg\phi(\overline{y},\overline{x}_{m+1}) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1}))) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m.(\exists\overline{y}_1,\ldots,\overline{y}_m,\overline{z}_1,\ldots,\overline{z}_m. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\psi(\overline{z}_i,\overline{x}_i) \wedge \phi(\overline{y}_i,\overline{x}_i))) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad \forall\overline{x}_{m+1},\overline{y},\overline{z}.(\neg\psi(\overline{z},\overline{x}_{m+1}) \vee \neg\phi(\overline{y},\overline{x}_{m+1}) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1})) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m.\exists\overline{y}_1,\ldots,\overline{y}_m,\overline{z}_1,\ldots,\overline{z}_m. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\psi(\overline{z}_i,\overline{x}_i) \wedge \phi(\overline{y}_i,\overline{x}_i)) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad \forall\overline{x}_{m+1},\overline{y},\overline{z}.(\neg\psi(\overline{z},\overline{x}_{m+1}) \vee \neg\phi(\overline{y},\overline{x}_{m+1}) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1})) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m.\exists\overline{y}_1,\ldots,\overline{y}_m,\overline{z}_1,\ldots,\overline{z}_m.\forall\overline{x}_{m+1},\overline{y},\overline{z}. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\psi(\overline{z}_i,\overline{x}_i) \wedge \phi(\overline{y}_i,\overline{x}_i)) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad \wedge (\neg\psi(\overline{z},\overline{x}_{m+1}) \vee \neg\phi(\overline{y},\overline{x}_{m+1}) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1})) \\
&\equiv \exists\overline{x}_1,\ldots,\overline{x}_m,\overline{y}_1,\ldots,\overline{y}_m,\overline{z}_1,\ldots,\overline{z}_m.\forall\overline{x}_{m+1},\overline{y},\overline{z}. \\
&\quad (\bigwedge_{i\in\{1,\ldots,m\}} (\psi(\overline{z}_i,\overline{x}_i) \wedge \phi(\overline{y}_i,\overline{x}_i)) \wedge \bigwedge_{i,j\in\{1,\ldots,m\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j \wedge \\
&\quad (\neg\psi(\overline{z},\overline{x}_{m+1}) \vee \neg\phi(\overline{y},\overline{x}_{m+1}) \vee \bigvee_{i\in\{1,\ldots,m\}} \overline{x}_i = \overline{x}_{m+1}))
\end{aligned}
$$

FIGURE A.1: Derivation for the $\psi_{S,q,s}$ formula.

$$\begin{aligned}
\psi(\overline{y},\overline{t}) &= \bigwedge_{i\in\{1,\ldots,n\}} \exists \overline{j}.(\phi_i(\overline{j},\overline{y}) \wedge \overline{y}(i) = \overline{t}(i)) \wedge \bigwedge_{j\in\{1,\ldots,m\}} \neg\exists \overline{z}.\phi_j(\overline{z},\overline{y}) \\
&\equiv \exists \overline{j}_1,\ldots,\overline{j}_n.(\bigwedge_{i\in\{1,\ldots,n\}} \phi_i(\overline{j}_i,\overline{y}) \wedge \overline{y}(i) = \overline{t}(i)) \wedge \bigwedge_{j\in\{1,\ldots,m\}} \forall \overline{z}.\neg\phi_j(\overline{z},\overline{y}) \\
&\equiv \exists \overline{j}_1,\ldots,\overline{j}_n.(\bigwedge_{i\in\{1,\ldots,n\}} \phi_i(\overline{j}_i,\overline{y}) \wedge \overline{y}(i) = \overline{t}(i)) \wedge \forall \overline{z}_1,\ldots,\overline{z}_m. \bigwedge_{j\in\{1,\ldots,m\}} \neg\phi_j(\overline{z}_j,\overline{y}) \\
&\equiv \exists \overline{j}_1,\ldots,\overline{j}_n.(\bigwedge_{i\in\{1,\ldots,n\}} \phi_i(\overline{j}_i,\overline{y}) \wedge \overline{y}(i) = \overline{t}(i) \wedge \forall \overline{z}_1,\ldots,\overline{z}_m. \bigwedge_{j\in\{1,\ldots,m\}} \neg\phi_j(\overline{z}_j,\overline{y})) \\
&\equiv \exists \overline{j}_1,\ldots,\overline{j}_n.\forall \overline{z}_1,\ldots,\overline{z}_m.(\bigwedge_{i\in\{1,\ldots,n\}} \phi_i(\overline{j}_i,\overline{y}) \wedge \overline{y}(i) = \overline{t}(i) \wedge \bigwedge_{j\in\{1,\ldots,m\}} \neg\phi_j(\overline{z}_j,\overline{y}))
\end{aligned}$$

FIGURE A.2: Derivation for the $\theta_{q,S}(\overline{y},\overline{t})$ formula.

derivation:

$$\begin{aligned}
\gamma &= \exists \overline{x}.\forall \overline{y}.\phi(\overline{x},\overline{y}) \wedge \neg(\exists \overline{z}.\psi(\overline{z})) \\
&\equiv \exists \overline{x}.\forall \overline{y}.\phi(\overline{x},\overline{y}) \wedge \forall \overline{z}.\neg\psi(\overline{z}) \\
&\equiv \exists \overline{x}.(\forall \overline{y}.\phi(\overline{x},\overline{y}) \wedge \forall \overline{z}.\neg\psi(\overline{z})) \\
&\equiv \exists \overline{x}.\forall \overline{y},\overline{z}.(\phi(\overline{x},\overline{y}) \wedge \neg\psi(\overline{z}))
\end{aligned}$$

Therefore we can rewrite $\phi_{INDIST(S,db)} \wedge \neg\psi$ to an equivalent *BSRRC* formula.

Given a non-boolean query q, a state db, and a security policy S, the sets $Auth_{S,q}(db)$ and $Ind_{S,q}(db)$ are finite and can be computed just by evaluating the constraints in S for all the tuples in $[q]^{db}$. Similarly, we can trivially compute the value $card_{S,db,q}(\overline{t})$ for any $\overline{t} \in Ind_{S,q}(db)$. Note that given a state s and a security policy S, we can implement a computable algorithm that produces the formula $\phi'_{INDIST(S,db)}$. It is easy to see that the rewriting from $\phi'_{INDIST(S,db)}$ to the *BSRRC*-formula $\phi_{INDIST(S,db)}$ can be done in a systematic way in a finite number of operations. Therefore, we can implement a computable algorithm that, given a state db, an *ERC*-policy S, and an *ERC*-formula ψ, produces the *BSRRC*-formulae $\phi'_{INDIST(S,db)}$, $\phi_{INDIST(S,db)} \wedge \psi$, and $\phi_{INDIST(S,db)} \wedge \neg\psi$.

Since all the conditions of Lemma 3.1 are satisfied, then $AGREE^{ERC}$ is decidable. □

Proof of Theorem 3.7. We apply Lemma 3.3 to the *ERC* fragment, i.e., $F = F' = ERC$. From Theorem 3.3, it follows that $AGREE^{ERC}$ is decidable. We therefore need only an encoding of $\phi_{T,\psi(\overline{x})}$ for any finite multi-set T and any $\psi(\overline{x}) \in ERC$. In the following, given a multi-set of tuples T in M and a tuple $\overline{t} \in T$, let $K_{\overline{t},T}$ be the multi-set $\{\overline{t}' \mid \overline{t}' \in T \wedge \overline{t} \sqsubseteq \overline{t}'\}$.

Let T be a multi-set of masked and unmasked tuples, and let $\psi(\overline{x}) \in ERC$ be a formula with free variables $\overline{x}$ such that $|\overline{x}| = |\overline{t}|$ for all $\overline{t} \in T$. The formula $\phi_{T,\psi(\overline{x})}$ is given by:

$$\phi_{T,\psi(\overline{x})} := \bigwedge_{\overline{t}\in T} \exists^{\geq |K_{\overline{t},T}|}\overline{x}.\ (\psi(\overline{x}) \wedge \bigwedge_{i\in\{1,\ldots,|\overline{x}|\}\wedge\overline{t}(i)\neq\dagger} \overline{x}(i) = \overline{t}(i)).$$

Note that $\phi_{T,\psi(\overline{x})}$ can be equivalently rewritten as an *ERC*-sentence as follows:

$$\begin{aligned}
\phi_{T,\psi(\overline{x})} := \bigwedge_{\overline{t}\in T} \exists \overline{x}_1,\ldots,\overline{x}_{|K_{\overline{t},T}|}.(&\bigwedge_{i\in\{1,\ldots,|K_{\overline{t},T}|\}} (\bigwedge_{l\in\{1,\ldots,|\overline{x}|\}\wedge\overline{t}(l)\neq\dagger} \overline{x}_i(l) = \overline{t}(l)) \wedge \\
&\bigwedge_{i\in\{1,\ldots,|K_{\overline{t},T}|\}} \psi(\overline{x}_i) \wedge \bigwedge_{i,j\in\{1,\ldots,|K_{\overline{t},T}|\}\wedge i\neq j} \overline{x}_i \neq \overline{x}_j).
\end{aligned}$$

In the formula above, $|\overline{x}_i| = |\overline{x}|$ for all $i \in \{1,\ldots,|K_{\overline{t},T}|\}$. Note that if $T = \emptyset$, then the encoding is $\top$ since $\emptyset \preceq K$ for any $K \in N$.

Given a finite multi-set T, and a tuple $\overline{t} \in T$, the multi-set $K_{\overline{t},T}$ is finite and can be computed trivially. Therefore, given a multi-set T and an *ERC*-formula $\psi(\overline{x})$, we can implement a computable algorithm that produces the *ERC*-formula $\phi_{T,\psi(\overline{x})}$. Since all the conditions of Lemma 3.3 are satisfied, then $SUBSUME^{ERC}$ is decidable.

Correctness: Here we prove the correctness of the encoding $\phi_{T,\psi(\overline{x})}$ given above. We now show that given a state db in Ω_D, $[\phi_{T,\psi(\overline{x})}]^{db} = \top$ iff $T \preceq [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$.

(⇒). From $[\phi_{T,\psi(\overline{x})}]^{db} = \top$, it follows that for each $\overline{t} \in T$ there are at least $|K_{\overline{t},T}|$ tuples $\overline{v} \in [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$ such that $\overline{t} \sqsubseteq \overline{v}$. Therefore, we can define a mapping f from T to $[\{\overline{x} \mid \psi(\overline{x})\}]^{db}$ such that $\overline{t} \sqsubseteq f(\overline{t})$ for all $\overline{t} \in T$. Moreover, from the definition of $K_{\overline{t},T}$, it follows that for each $\overline{t} \in T$, T contains exactly $|K_{\overline{t},T}|$ tuples $\overline{t}'$ such that $\overline{t} \sqsubseteq \overline{t}'$. Therefore, it is easy to see that there is a mapping f' from T to $[\{\overline{x} \mid \psi(\overline{x})\}]^{db}$ such that $\overline{t} \sqsubseteq f'(\overline{t})$ for all $\overline{t} \in T$ that is also injective. Therefore, $T \preceq [\{\overline{x}|\psi(\overline{x})\}]^{s}$.
(⇐). From $T \preceq [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$, it follows that for each $\overline{t} \in T$ there are at least $|K_{\overline{t},T}|$ tuples $\overline{v} \in [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$ such that $\overline{t} \sqsubseteq \overline{v}$. Therefore, $[\phi_{T,\psi(\overline{x})}]^{db} = \top$. □

Proof of Theorem 3.10. We first introduce some additional results that we use in the proof of Theorem 3.10. There is no encoding of $\phi_{T,\psi(\overline{x})}$ in the *ERC* fragment, but there is an encoding in an extension of the *ERC* fragment called ERC^{+}. We first introduce this extension, and afterwards we show that $AGREE^{ERC^{+}}$ is decidable for all *ERC* policies, i.e., the problem is decidable for all database schemas, all *ERC*-policies, and all ERC^{+}-queries.

We first introduce the $\overline{ERC}$ fragment, the dual of the *ERC*-fragment.

Definition A.1. $\overline{\text{ERC}}$: Let D be a database schema. The formula $\phi(\overline{z}) := \forall \overline{x}.\, \psi(\overline{x}, \overline{y})$ is an $\overline{ERC}$-formula over D iff ψ is a quantifier-free *RC* formula over D. □

Then, the ERC^{+} fragment is defined as follows:

Definition A.2. ERC^{+}: Let D be a database schema. An ERC^{+} formula over D is inductively defined as follows.

1. ϕ is an ERC^{+} formula over D, where ϕ is an *ERC*-formula over D.
2. ϕ is an ERC^{+} formula over D, where ϕ is an $\overline{ERC}$-formula over D.
3. $\phi \; op \; \psi$ is an ERC^{+} formula over D, where ϕ is an *ERC*-formula over D, ψ is an $\overline{ERC}$-formula over D, and $op \in \{\wedge, \vee\}$.

□

Observe that $ERC \subset ERC^{+} \subset BSRRC$. As a result, the encoding presented in Theorem 3.3 can be used to prove the decidability of the *AGREE* problem for the ERC^{+} fragment. Note that this result holds just for *ERC*-policies, not for all ERC^{+}-policies.

Theorem A.1. *$AGREE^{ERC^{+}}$ is decidable for all ERC-policies.*

Proof. The steps from 1 to 3 are the same as in the proof of Theorem 3.3 because we consider only *ERC*-policies. We just change slightly the fourth step.
Step 4: We prove that for any ERC^{+}-sentence γ and any *ERC*-policy S then:

1. $\phi_{INDIST(S,db)} \wedge \gamma \in BSRRC$, and
2. $\phi_{INDIST(S,db)} \wedge \neg\gamma \in BSRRC$.

From the fact that γ is of the form $\phi \; op \; \psi$, where $\phi \in ERC$ and $\psi \in \overline{ERC}$, it follows that both γ and $\neg\gamma$ are in *BSRRC* because ϕ is a sentence of the form $\exists \overline{x}.\phi(\overline{x})$ and ψ is a sentence of the form $\forall \overline{x}.\phi(\overline{x})$. From this and $\phi_{INDIST(S,db)} \in BSRRC$, it follows that $\phi_{INDIST(S,db)} \wedge \gamma \in BSRRC$, and $\phi_{INDIST(S,db)} \wedge \neg\gamma \in BSRRC$.

It is easy to see that we can implement a computable algorithm that, given a state db, an *ERC*-security policy S, and an ERC^{+}-formula ψ, produces the *BSRRC*-formulae $\phi'_{INDIST(S,db)}$, $\phi_{INDIST(S,db)} \wedge \psi$, and $\phi_{INDIST(S,db)} \wedge \neg\psi$.

Since all the conditions of Lemma 3.1 are met, $AGREE^{ERC^{+}}$ is decidable for all *ERC*-policies. □

We are now ready to prove that $EQUAL^{ERC}$ is decidable (i.e., Theorem 3.10). In this proof, we apply Lemma 3.4 to the *ERC* and ERC^{+} fragments, i.e., $F = ERC$ and $F' = ERC^{+}$. Let $\psi(\overline{x})$ be an *ERC*-formula with free variables $\overline{x}$ and T be a set of tuples such that $|\overline{x}| = |\overline{t}|$ for all $\overline{t} \in T$. For $EQUAL^{ERC}$ the encoding is as follows:

$$AT\_LEAST_{T,\psi(\overline{x})} := \bigwedge_{\overline{t} \in T} \exists \overline{x}.(\psi(\overline{x}) \wedge \overline{x} = \overline{t})$$

$$AT\_MOST_{T,\psi(\overline{x})} := \forall \overline{x}.(\psi(\overline{x}) \Rightarrow \bigvee_{\overline{t} \in T} \overline{x} = \overline{t})$$

$$\phi_{T,\psi(\overline{x})} := AT\_LEAST_{T,\psi(\overline{x})} \wedge AT\_MOST_{T,\psi(\overline{x})}$$

Note that if $T = \emptyset$, then $\phi_{T,\psi(\overline{x})} := \forall \overline{x}.\neg\psi(\overline{x}) \in \overline{ERC}$.

Observe that $AT\_LEAST_{T,\psi(\overline{x})}$ can be equivalently rewritten as an *ERC*-formula, whereas $AT\_MOST_{T,\psi(\overline{x})}$ can be equivalently rewritten as an $\overline{ERC}$-formula, and therefore $\phi_{T,\psi(\overline{x})}$ can be equivalently rewritten as an ERC^{+}-sentence. Moreover, it is easy to see that given an *ERC* formula

$\psi(\overline{x})$ and a set of tuples T we can compute the encoding $\phi_{T,\psi(\overline{x})}$. Therefore from Lemma 3.4 and Theorem A.1, it follows that $EQUAL^{ERC}$ is decidable.

Correctness: We now prove the correctness of the encoding $\phi_{T,\psi(\overline{x})}$ given above.

(⇒). We prove that $[\phi_{T,\psi(\overline{x})}]^{db} = \top$ implies $T = [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$. From $[\phi_{T,\psi(\overline{x})}]^{s} = \top$, it follows that $[AT\_LEAST_{T,\psi(\overline{x})}]^{db} = \top$ and $[AT\_MOST_{T,\psi(\overline{x})}]^{db} = \top$. From the former, it follows that $T \subseteq [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$, and from the latter, it follows that $T \supseteq [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$. Hence, $T = [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$.

(⇐). We prove that $T = [\{\overline{x} \mid \psi(\overline{x})\}]^{db}$ implies $[\phi_{T,\psi(\overline{x})}]^{db} = \top$. From $T = [\{\overline{x} \mid \psi(\overline{x})\}]^{s}$, it follows that $[AT\_LEAST_{T,\psi(\overline{x})}]^{s} = \top$ and $[AT\_MOST_{T,\psi(\overline{x})}]^{s} = \top$. Hence, $[\phi_{T,\psi(\overline{x})}]^{s} = \top$. □

Appendix B

Proofs for Chapter 4

B.1 Acyclicity of the ground graph

In this section we show that the ground graph of an acyclic program is a forest of poly-trees.

B.1.1 Proofs about Annotations

Here we prove that the ordering, disjointness, and uniqueness annotations capture the desired semantic properties.

Lemma B.1. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain and p be a $(\Sigma, \mathbf{dom})$-*ProbLog *program. Furthermore, let $\prec$ be the following relation over pairs of predicate symbols in Σ: $a \prec b$ iff $b \in reach(a)$ and $a \notin reach(b)$. The relation $\prec$ is a strict partial order and it is well-founded.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain and p be a $(\Sigma, \mathbf{dom})$-ProbLog program. Furthermore, let $\prec$ be the following relation over pairs of predicate symbols in Σ: $a \prec b$ iff $b \in reach(a)$ and $a \notin reach(b)$.

$\prec$ is a strict partial order. To show that $\prec$ is a strict partial order we must show that $\prec$ is irreflexive and transitive. For irreflexivity, let a be a predicate symbol and assume, for contradiction's sake, that $a \prec a$ holds. From this, $a \in reach(a)$ and $a \notin reach(a)$, leading to a contradiction. For transitivity, let a, b, c be three predicate symbols such that $a \prec b$ and $b \prec c$. From $a \prec b$, $b \in reach(a)$ and $a \notin reach(b)$. From $b \prec c$, $c \in reach(b)$ and $b \notin reach(c)$. From $b \in reach(a)$ and $c \in reach(b)$, it follows that $c \in reach(a)$ (since $reach(b) \subseteq reach(a)$). From $b \prec c$, $reach(c) \subset reach(b)$. From this and $a \notin reach(b)$, it follows that $a \notin reach(c)$. From $c \in reach(a)$ and $a \notin reach(c)$, $a \prec c$.

$\prec$ is well-founded. To show that $\prec$ is well-founded, it is enough to show that there is no infinite descending sequence of elements. Assume, for contradiction's sake, that this is not the case. Namely, there is an infinite sequence of predicate symbols $a_0, a_1, \ldots$ such that for all $i \in \mathbb{N}$, $a_{i+1} \prec a_i$. We now consider the first $n+1$ elements, where n is the number of predicate symbols in Σ. Then, we have the descending chain $a_{n+1} \prec a_n \prec \ldots \prec a_1 \prec a_0$. Observe that all $n+1$ elements have to be distinct (due to the irreflexivity and transitivity of $\prec$). This however contradicts the fact that we have at most n different predicate symbols. □

Proposition B.1. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain and p be a $(\Sigma, \mathbf{dom})$-*ProbLog *program. If we can derive the annotation $ORD(A)$ from p, then if $pr_1(\overline{a}_1, \overline{a}_2)$, $pr_2(\overline{a}_2, \overline{a}_3), \ldots, pr_{n-1}(\overline{a}_{n-1}, \overline{a}_n)$ are in $ground(p)$ and $\{pr_1, \ldots, pr_{n-1}\} \subseteq A$, then $\overline{a}_1 \neq \overline{a}_n$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain and p be a $(\Sigma, \mathbf{dom})$-ProbLog program. Furthermore, we assume that (1) we can derive $ORD(A)$ from p, (2) $pr_1(\overline{a}_1, \overline{a}_2), \ldots, pr_{n-1}(\overline{a}_{n-1}, \overline{a}_n)$ are in $ground(p)$, and (3) $\{pr_1, \ldots, pr_{n-1}\} \subseteq A$. Let $\prec$ be the strict partial order over predicates in Σ defined in Lemma B.1. We lift $\prec$ over sets of predicate symbols as follows: $A \prec B$ iff there is a bijection μ from A to B such that (1) there exists $a \in A$ such that $a \prec \mu(a)$, (2) $a \prec \mu(a)$ or $a = \mu(a)$ for all $a \in A$, and (3) $A \neq B$. We now prove, by induction over $\prec$, that the transitive closure of the unions of the relations induced by $pr_1, \ldots, pr_{n-1}$ over $ground(p)$ is a strict partial order over $\mathbf{dom}^{|pr|/2}$. From this, it follows that if $pr_1(\overline{a}_1, \overline{a}_2), pr_2(\overline{a}_2, \overline{a}_3), \ldots, pr_{n-1}(\overline{a}_{n-1}, \overline{a}_n)$ are in $ground(p)$, then $\overline{a}_1 \neq \overline{a}_n$.

Base Case. For the base case, assume that for all $a \in A$, there is no predicate symbol a' (different from a) such that $a' \prec a$. From this and $ORD(A)$ can be derived from p, it follows that (1) for all $pr \in A$, there is no rule r in p such that $pred(head(r)) = pr$ and $body(r) \neq \emptyset$, and (2) the transitive closure of the relation $R = \bigcup_{pr \in A} \{(\overline{c}, \overline{v}) \mid \exists r \in p.\, ((head(r) = pr(\overline{c}, \overline{v}) \vee \exists v'.\, head(r) = v'{::}pr(\overline{c}, \overline{v})) \wedge body(r) = \emptyset) \wedge |\overline{c}| = |\overline{v}| = {}^{|pr|}/_2\}$ is strict partial order. From this and $\{(\overline{c}, \overline{v}) \mid \exists pr \in A.\, pr(\overline{c}, \overline{v}) \in ground(p) \wedge |\overline{c}| = |\overline{v}| = {}^{|pr|}/_2\} \subseteq R$, it follows that the transitive closure of the union of the relations induced by pr over $ground(p)$ is a strict partial order over $\mathbf{dom}^{|pr|/2}$.

Induction Step. Assume that the claim holds for all $A' \prec A$. We now prove that it holds also for A. From the fact that $ORD(A)$ can be derived from p, it follows that there is an annotation $ORD(A')$ and two distinct predicate symbols $pr \in \Sigma$ and $pr' \in A'$ such that

1. $ORD(A')$ can be derived from p,
2. $|pr| = |pr'|$,
3. $A = (A' \setminus \{pr'\}) \cup \{pr\}$,
4. $pr' \notin \bigcup_{a \in A} reach(a)$, and
5. for all rules r in p such that $pred(head(r)) = pr$, there are sequences of variables $\overline{x}$ and $\overline{y}$ such that $head(r) = pr(\overline{x}, \overline{y})$, $pr'(\overline{x}, \overline{y}) \in body(r)$, and $|\overline{x}| = |\overline{y}|$.

There are two cases:

- There are no rules in p such that $pred(head(r)) = pr$. In this case, the claim trivially holds from (1) and the induction hypothesis.
- There exists at least one rule in p such that $pred(head(r)) = pr$. From this and (5), $head(r) = pr(\overline{x}, \overline{y})$ and $pr'(\overline{x}, \overline{y}) \in body(r)$. Hence, $pr' \prec pr$ (since $pr' \notin reach(pr)$ and $pr \in reach(pr')$). From this, (3), and (4), it follows that $A' \prec A$. We now prove that also the relation induced by A over $ground(p)$ is a strict partial order over $\mathbf{dom}^{|pr|/2}$. Since $pr \in A$, $A = (A' \setminus \{pr_1\}) \cup \{pr\}$, $pr_1 \in A'$, and $ORD(A)$ can be derived from p, it follows that for all rules r in p such that $pred(head(r)) = pr$, there are sequences of variables $\overline{x}$ and $\overline{y}$ and an $i \in \mathbb{N}$ such that $head(r) = pr(\overline{x}, \overline{y})$, $body(r, i) = pr_1(\overline{x}, \overline{y})$, and $|\overline{x}| = |\overline{y}|$. From this, it follows that the $\{(\overline{v}, \overline{w}) \mid pr(\overline{v}, \overline{w}) \in ground(p)\} \subseteq \{(\overline{v}, \overline{w}) \mid pr_1(\overline{v}, \overline{w}) \in ground(p)\}$. From $A' \prec A$ and the induction hypothesis, it follows that the transitive closure of $\bigcup_{pr' \in A'} \{(\overline{v}, \overline{w}) \mid pr'(\overline{v}, \overline{w}) \in ground(p)\}$ is a strict partial order over $\mathbf{dom}^{|pr'|/2}$. From this, $pr_1 \in A'$, $A = (A' \setminus \{pr_1\}) \cup \{pr\}$, $pr_1 \in A'$, and $\{(\overline{v}, \overline{w}) \mid pr(\overline{v}, \overline{w}) \in ground(p)\} \subseteq \{(\overline{v}, \overline{w}) \mid pr_1(\overline{v}, \overline{w}) \in ground(p)\}$, it follows that the transitive closure of $\bigcup_{pr' \in A} \{(\overline{v}, \overline{w}) \mid pr'(\overline{v}, \overline{w}) \in ground(p)\}$ is a strict partial order over $\mathbf{dom}^{|pr'|/2}$. This completes the proof of our claim.

This completes the proof. □

Proposition B.2. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain and p be a $(\Sigma, \mathbf{dom})$-*ProbLog *program. If $DIS(pr, pr')$ can be derived from p, then there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain and p be a $(\Sigma, \mathbf{dom})$-ProbLog program. Furthermore, we assume that $DIS(pr, pr')$ can be derived from p. Furthermore, let $\prec$ be the strict partial order defined in Lemma B.1. We lift $\prec$ to pairs of predicate symbols as follows: $(a, b) \prec (a', b')$ iff (1) $a = a'$ and $b \prec b'$, (2) $a \prec a'$ and $b = b'$, or (3) $a \prec a'$ and $b \prec b'$. We prove, by induction over $\prec$, that whenever we can derive $DIS(pr, pr')$ from p, there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$.

Base Case. Let (pr, pr') be a pair in Σ^2 such that there is no $(pr_1, pr_1') \prec (pr, pr')$ and $DIS(pr, pr')$ can be derived from p. From this, it follows that (1) there are no rules $r \in p$ such that $body(r) \neq \emptyset$ and $pred(head(r)) = pr$ or $pred(head(r)) = pr'$, and (2) the sets $A = \{\overline{c} \mid pr(\overline{c}) \in p \vee \exists v'.\ v'::pr(\overline{c}) \in p\}$ and $A' = \{\overline{c} \mid \exists r \in p.\, head(r) = pr'(\overline{c}) \vee \exists v'.\ v'::pr'(\overline{c}) \in p\}$ are disjoint. From this, it follows that there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$.

Induction Step. Assume that there is no tuple $\overline{v}$ such that $pr_1(\overline{v}) \in ground(p)$ and $pr_1'(\overline{v}) \in ground(p)$ for any $(pr_1, pr_1') \prec (pr, pr')$ such that $DIS(pr_1, pr_1')$ can be derived from p. We now prove that there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$. There are three cases:

- There are no rules $r \in p$ such that $body(r) \neq \emptyset$ and $pred(head(r)) = pr$, the annotation $DIS(pr, pr_1')$ can be derived from p, $pr_1' \notin reach(pr')$, $pr' \neq pr_1'$, and for all rules $r \in p$ such that $head(r) = pr'(\overline{x})$, $pr_1'(\overline{x}) \in body(r)$. From this, $(pr, pr_1') \prec (pr, pr')$. From $(pr, pr_1') \prec (pr, pr')$, $DIS(pr, pr')$ can be derived from p, and the induction's hypothesis, it follows that there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr_1'(\overline{v}) \in ground(p)$. Furthermore, the relation associated with pr' is a subset of the relation associated to pr_1'. Therefore, there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$.
- There are no rules $r \in p$ such that $body(r) \neq \emptyset$ and $pred(head(r)) = a'$, the annotation $DIS(pr_1, pr')$ can be derived from p, $pr_1 \notin reach(pr)$, $pr \neq pr_1$, and for all rules $r \in p$ such that $head(r) = pr(\overline{x})$, $pr_1(\overline{x}) \in body(r)$. From this, $(pr_1, pr') \prec (pr, pr')$. From $(pr_1, pr') \prec (pr, pr')$, $DIS(pr_1, pr')$ can be derived from p, and the induction's hypothesis, it follows that there is no tuple $\overline{v}$ such that $pr_1(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$. Furthermore, the relation associated with pr is a subset of the relation associated to pr_1. Therefore, there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$.
- The annotation $DIS(pr_1, pr_1')$ can be derived from p, $\{pr_1, pr_1'\} \cap (reach(pr) \cup reach(pr')) = \emptyset$, $pr_1 \neq pr$, $pr_1' \neq pr'$, and for all rules $r \in p$, if $head(r) = pr(\overline{x})$, $pr_1(\overline{x}) \in body(r)$ and if

$head(r) = pr'(\overline{x})$, $pr'_1(\overline{x}) \in body(r)$. From this, $(pr_1, pr'_1) \prec (pr, pr')$. From $(pr_1, pr'_1) \prec (pr, pr')$, $DIS(pr_1, pr'_1)$ can be derived from p, and the induction's hypothesis, it follows that there is no tuple $\overline{v}$ such that $pr_1(\overline{v}) \in ground(p)$ and $pr'_1(\overline{v}) \in ground(p)$. Furthermore, the relation associated with pr is a subset of the relation associated with pr_1 and the relation associated with pr' is a subset of the relation associated with pr'_1. Therefore, there is no tuple $\overline{v}$ such that $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$.

This completes the proof of our claim. □

Proposition B.3. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain and p be a $(\Sigma, \mathbf{dom})$-*ProbLog *program. If $UNQ(pr, K)$ can be derived from p, then for all tuples $pr(\overline{v})$, $pr(\overline{w}) \in ground(p)$, if $\overline{v}(i) = \overline{w}(i)$ for all $i \in K$, then $\overline{v} = \overline{w}$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain and p be a $(\Sigma, \mathbf{dom})$-ProbLog program. Furthermore, we assume that $UNQ(pr, K)$ can be derived from p. Finally, let $\prec$ be the strict partial order defined in Lemma B.1. We prove, by induction over $\prec$, that for all annotations $UNQ(pr, K)$ that can be derived from p, and all tuples $pr(\overline{v}), pr(\overline{w}) \in ground(p)$, if $\overline{v}(i) = \overline{w}(i)$ for all $i \in K$, then $\overline{v} = \overline{w}$. Without loss of generality, in the following we assume that $K \neq \{1, \ldots, |pr|\}$. If this is not the case, then our claim holds trivially.

Base Case. Let pr be a predicate such that there is no $pr' \prec pr$. From this and $UNQ(pr, K)$ can be derived from p, it follows that there are no rules r in p such that $body(r) \neq \emptyset$ and $pred(head(r)) = pr$, and for all $\overline{w}, \overline{v}$ in $A = \{\overline{c} \mid pr(\overline{c}) \in p \vee \exists v.v{::}pr(\overline{c}) \in p\}$, if $\overline{v}(i) = \overline{w}(i)$ for all $i \in K$, then $\overline{v} = \overline{w}$. From this and $\{\overline{c} \mid pr(\overline{c}) \in ground(p)\} \subseteq A$, it follows that for all tuples $pr(\overline{v}), pr(\overline{w}) \in ground(p)$, if $\overline{v}(i) = \overline{w}(i)$ for all $i \in K$, then $\overline{v} = \overline{w}$.

Induction Step. Assume that the claim holds for any $pr' \prec pr$ and $K \subseteq \{1, \ldots, |pr|\}$ such that $UNQ(pr, K)$ can be derived from p (we denote this induction hypothesis as (♣)). We now prove that the claim holds for pr as well. Without loss of generality, we assume that there is at least one rule r such that $body(r) \neq \emptyset$ and $pred(head(r)) = pr$ (otherwise the proof is trivial). Assume, for contradiction's sake, that there are two ground atoms $pr(\overline{v}), pr(\overline{w}) \in ground(p)$ such that $\overline{v}(i) = \overline{w}(i)$ for all $i \in K$ but $\overline{v} \neq \overline{w}$. There are two cases:

- $pr(\overline{v})$ and $pr(\overline{w})$ are generated by two ground instances of the same rule r. From $\overline{v} \neq \overline{w}$, it follows that there is $j \notin K$ such that $\overline{v}(j) \neq \overline{w}(j)$. From this and the fact that both atoms are generated by the same rule r, it follows that there is a variable x such that $x \in \{\overline{x}(i) \mid i \notin K\}$ and $x = \overline{x}(j)$, where $\overline{x} = args(head(r))$. From this, it follows that $x \in \bigcup_{l \in bound(head(r), K, body^+(r), \mu')} vars(l)$. We claim that the value of x is determined by the values of the variables whose positions are in K. From this and the fact that $\overline{v}$ and $\overline{w}$ agree on all values whose positions are in K, it follows that $\overline{v} = \overline{w}$, leading to a contradiction.
  We now prove our claim that the value of x is determined by the values of the variables whose positions are in K. The value of the variable x is determined by the grounding of one of the atoms in $bound(head(r), K, body^+(r), \mu')$. We first slightly modify the definition of $bound$. We denote by $bd^0(h, K, L, \mu')$ the function $\bigcup_{\substack{b(\overline{y}) \in L \wedge b \notin reach(pr) \wedge b \neq pr \wedge \\ u(\overline{y}, \mu'(b)) \subseteq u(args(h), K)}} \{b(\overline{y})\}$ and by $bd^i(h, K, L, \mu')$, where $i > 0$, the function $bd^{i-1}(h, K, L, \mu') \cup \bigcup_{\substack{b(\overline{y}) \in L \wedge b \notin reach(pr) \wedge b \neq pr \wedge \\ \exists l' \in bound^{i-1}(h, K, L, \mu').(u(\overline{y}, \mu'(b)) \subseteq vars(l'))}} \{b(\overline{y})\}$.
  It is easy to see that (1) $bound(head(r), K, body^+(r), \mu') = \bigcup_{i \in \mathbb{N}} bd^i(h, K, L, \mu')$, and (2) the fixpoint is always reached in a finite number of steps (bounded by $|body^+(r)|$). We now prove by induction on i that the groundings of the literals in $bd^i(h, K, L, \mu')$, where $L = body^+(r)$, is always determined by the values of the variables in $u(args(head(r)), K)$. From this, our claim immediately follows. For the base case, let $i = 0$. Then, for any literal $b(\overline{y}) \in bd^0(h, K, L, \mu')$, it follows that (a) $b \notin reach(pr)$, (b) $b \neq pr$, and (c) $u(\overline{y}, \mu'(b)) \subseteq u(args(head(r)), K)$. From this, $L = body^+(r)$, and $pred(head(r)) = pr$, it follows that $b \prec pr$. From this, $UNQ(b, \mu'(b))$ can be derived from p, and the induction's hypothesis (♣), it follows that the variables in $UNQ(b, \mu'(b))$ uniquely determine the grounding of $b(\overline{y})$. From this and $u(\overline{y}, \mu'(b)) \subseteq u(args(head(r)), K)$, it follows that the variables in $u(args(head(r)), K)$ uniquely determine the grounding of $b(\overline{y})$. For the induction's step, assume that the claim hold for all $j < i$ (we denote this induction hypothesis as (♠)). Let $b(\overline{y}) \in bd^i(h, K, L, \mu')$ (without loss of generality, $b(\overline{y}) \notin bd^{i-1}(h, K, L, \mu')$). From this, it follows that $b \prec pr$ (since $b \notin reach(pr)$, $b \neq pr$, and $pr \in reach(b)$) and there is an $l' \in bound^{i-1}(h, K, L, \mu')$ such that $u(\overline{y}, \mu'(b)) \subseteq vars(l')$. From the induction hypothesis (♠), it follows that the grounding of l' is directly determined by the grounding of the values of the variables in $u(args(head(r)), K)$. Furthermore, from $b \prec pr$, $UNQ(b, \mu'(b))$ can be derived from p, and the induction hypothesis (♣), it follows that the values of the variables in $u(\overline{y}, \mu'(b))$ uniquely determine the grounding of $b(\overline{y})$. Therefore, the values of the variables in $u(args(head(r)), K)$ uniquely determine the grounding of $b(\overline{y})$. This completes the proof of the claim.

- $pr(\overline{v})$ and $pr(\overline{w})$ are generated by ground instances of two different rules r_1 and r_2. From this, $UNQ(pr, K)$ can be derived from p, and $K \neq \{1, \ldots, |pr|\}$, it follows that there is a value $i \in K$ such that $args(head(r_1))(i) \in \mathbf{dom}$, $args(head(r_2))(i) \in \mathbf{dom}$, and $args(head(r_1))(i) \neq args(head(r_2))(i)$. Therefore, there is an $i \in K$ such that $\overline{v}(i) \neq \overline{w}(i)$. This contradicts the fact that $\overline{v}(i) = \overline{w}(i)$ for all $i \in K$.

This completes the proof of our claim. □

B.1.2 Proofs about Propagation Maps

Here we prove some results about propagation maps.

Lemma B.2. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain, p be a $(\Sigma,$* **dom***)-ProbLog program, r be a rule in p, and l be the i-th literal in $body(r)$. Furthermore, let μ be the (r, l)-vertical map. Given a rule $r' \in ground(p, r)$, then $\overline{b}(j) = \overline{h}(\mu(j))$ for any j such that $\mu(j)$ is defined, where $\overline{h} = args(head(r'))$ and $\overline{b} = args(body(r', i))$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, r be a rule in p, and l be the i-th literal in $body(r)$. Furthermore, let μ be the (r, l)-vertical map and $r' \in ground(p, r)$. From the definition of $ground(p, r)$, it follows that there is an assignment Θ from variables to elements in **dom** such that $r' = r\Theta$. From this, $\overline{b}(j) = \Theta(args(l)(j))$ and $\overline{h}(\mu(j)) = \Theta(args(head(r))(\mu(j)))$. Note that from the definition of (r, l)-vertical map it follows that $args(l)(j) = args(head(r))(\mu(j))$. From this, $\overline{b}(j) = \overline{h}(\mu(j))$ whenever $\mu(j)$ is defined. □

Lemma B.3. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, r be a rule in p, l be the i-th literal in $body(r)$, and l' be the j-th literal in $body(r)$. Furthermore, let μ be the (r, l, l')-horizontal map. Given a rule $r' \in ground(p, r)$, then $\overline{b_1}(k) = \overline{b_2}(\mu(k))$ for any k such that $\mu(k)$ is defined, where $\overline{b_1} = args(body(r', i))$ and $\overline{b_2} = args(body(r', j))$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, r be a rule in p, l be the i-th literal in $body(r)$, and l' be the j-th literal in $body(r)$. Furthermore, let μ be the (r, l, l')-horizontal map and $r' \in ground(p, r)$. From the definition of $ground(p, r)$, it follows that there is an assignment Θ from variables to elements in **dom** such that $r' = r\Theta$. From this, $\overline{b_1}(j) = \Theta(args(l)(j))$ and $\overline{b_2}(\mu(j)) = \Theta(args(l')(\mu(j)))$. Note that from the definition of (r, l, l')-vertical map it follows that $args(l)(j) = args(l')(\mu(j))$. From this, $\overline{b_1}(j) = \overline{b_2}(\mu(j))$ whenever $\mu(j)$ is defined. □

Proposition B.4. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $P = pr_1 \xrightarrow{r_1, i_1} \ldots \xrightarrow{r_{n-1}, i_{n-1}} pr_n$ be a directed path in $graph(p)$, and $\nu : \mathbb{N} \to \mathbb{N}$ be a mapping. Furthermore, let $P' = a_1 \xrightarrow{r_1, s_1, i_1} a_2 \xrightarrow{r_2, s_2, i_2} \ldots \xrightarrow{r_{n-1}, s_{n-1}, i_{n-1}} a_n$ be a directed path in $gg(p)$ corresponding to P. If P ν-downward links to l, where l is the k-th literal in r_j, then $\overline{b_1}(m) = \overline{v_2}(\nu(m))$ whenever $\nu(m)$ is defined, where $\overline{b_1} = args(body(s_1, i_1))$ and $\overline{v_2} = args(body(s_j, k))$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $P = pr_1 \xrightarrow{r_1, i_1} \ldots \xrightarrow{r_{n-1}, i_{n-1}} pr_n$ be a directed path in $graph(p)$, and $\nu : \mathbb{N} \to \mathbb{N}$ be a mapping. Furthermore, let $P' = a_1 \xrightarrow{r_1, s_1, i_1} a_2 \xrightarrow{r_2, s_2, i_2} \ldots \xrightarrow{r_{n-1}, s_{n-1}, i_{n-1}} a_n$ be a directed path in $gg(p)$ corresponding to P. Assume that P ν-downward links to l, where l is the k-th literal in r_j. From this, it follows that the function $\mu := \mu' \circ \mu_j \circ \ldots \circ \mu_1$ satisfies $\mu(m) = \nu(m)$ for all m for which $\nu(k)$ is defined, where for $1 \leq h \leq j$, μ_h is the vertical map connecting $body(r_h, i_h)$ and r_h, and μ' is the horizontal map connecting $body(r_{j+1}, i_{j+1})$ with l. By repeatedly applying Lemma B.2 to the rules in P', we have that $\overline{b_1}(m) = \overline{b_j}(\phi(m))$ whenever $\phi(m)$ is defined, where $\overline{b_1} = args(body(s_1, i_1))$, $\overline{b_j} = args(body(s_j, i_j))$, and $\phi = \mu_j \circ \ldots \circ \mu_1$. Moreover, by applying Lemma B.3, we have that $\overline{b_j}(m) = \overline{v_2}(\mu'(m))$ whenever $\mu'(m)$ is defined, where $\overline{b_j} = args(body(s_j, i_j))$ and $\overline{v_2} = args(body(s_j, k))$. Therefore, we have that $\overline{b_1}(m) = \overline{v_2}(\mu(m))$ whenever $\mu'(m)$ is defined (by composing the previous results). From this and $\mu(m) = \nu(m)$ for all m for which $\nu(m)$ is defined, it follows that $\overline{b_1}(m) = \overline{v_2}(\nu(m))$ whenever $\mu'(m)$ is defined. □

Proposition B.5. *Let* **dom** *be a finite domain, $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $P = pr_1 \xrightarrow{r_1, i_1} \ldots \xrightarrow{r_{n-1}, i_{n-1}} pr_n$ be a directed path in $graph(p)$, and $\nu : \mathbb{N} \to \mathbb{N}$ be a mapping. Furthermore, let $P' = a_1 \xrightarrow{r_1, s_1, i_1} a_2 \xrightarrow{r_2, s_2, i_2} \ldots \xrightarrow{r_{n-1}, s_{n-1}, i_{n-1}} a_n$ be a directed path in $gg(p)$ corresponding to P. If P ν-upward links to l, where l is the k-th literal in r_j, then $\overline{b_n}(m) = \overline{v_2}(\nu(m))$ whenever $\nu(m)$ is defined, where $\overline{b_n} = args(head(r_{n-1}))$ and $\overline{v_2} = args(body(s_j, k))$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $P = pr_1 \xrightarrow{r_1,i_1} \dots \xrightarrow{r_{n-1},i_{n-1}} pr_n$ be a directed path in $graph(p)$, and $\nu : \mathbb{N} \to \mathbb{N}$ be a mapping. Furthermore, let $P' = a_1 \xrightarrow{r_1,s_1,i_1} a_2 \xrightarrow{r_2,s_2,i_2} \dots \xrightarrow{r_{n-1},s_{n-1},i_{n-1}} a_n$ be a directed path in $gg(p)$ corresponding to P. Assume that P ν-upward links to l, where l is the k-th literal in r_j. From this, it follows that the function $\mu := \mu'^{-1} \circ \mu_{j+1}^{-1} \circ \dots \circ \mu_{n-1}^{-1}$ satisfies $\mu(k) = \nu(k)$ for all k for which $\nu(k)$ is defined, where μ_h is the $(r_h, body(r_h, i_h))$-vertical map, for $j < h \leq n-1$, and μ' is the (r_j, l)-vertical map. By repeatedly applying Lemma B.2 to the rules in P', we obtain that $\overline{b_{j+1}}(\phi^{-1}(m)) = \overline{b_n}(m)$ whenever $\phi^{-1}(m)$ is defined, where $\overline{b_{j+1}} = args(body(s_{j+1}, i_{j+1}))$, $\overline{b_n} = args(head(s_{n-1}))$, and $\phi^{-1} = \mu_{j+1}^{-1} \circ \dots \circ \mu_{n-1}^{-1}$. Furthermore, by applying Lemma B.2 to the (r_j, l)-vertical map, we have that $\overline{v_2}(\mu'^{-1}(m)) = \overline{b_{j+1}}(m)$ whenever $\mu'^{-1}(m)$ is defined, where $\overline{b_{j+1}} = args(head(s_j))$ and $\overline{v_2} = args(body(s_j, k))$. From this and $\mu(m) = \nu(m)$ for all m for which $\nu(m)$ is defined, it follows that $\overline{b_n}(m) = \overline{v_2}(\nu(m))$ whenever $\nu(m)$ is defined. □

B.1.3 Proofs about Connected Rules

We now prove that, for strongly connected rules, the grounding of a rule's head uniquely determines the grounding of the rule's body (Proposition B.6), whereas for weakly connected rules the grounding of one of the atoms in the body determines the rule's grounding (Proposition B.7).

Proposition B.6. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, r be a rule in p, and $\mathcal{T}$ be the template containing all annotations that can be derived from p. If r is strongly connected for $\mathcal{T}$, then for all $r_1, r_2 \in ground(p, r)$, if $head(r_1) = head(r_2)$, then $r_1 = r_2$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, r be a rule in p, and $\mathcal{T}$ be the template containing all annotations that can be derived from p. Furthermore, we assume that there are join trees $J_1, \dots, J_n$ such that (a) the trees cover all literals in $body(r)$, and (b) for each $1 \leq i \leq n$, J_i is strongly connected for $\mathcal{T}$. Finally, let $r_1, r_2 \in ground(p, r)$ be two ground rules such that $head(r_1) = head(r_2)$. Assume, for contradiction's sake, that $r_1 \neq r_2$. Therefore, there is a position $1 \leq i \leq |body(r)|$ such that $body(r_1, i) \neq body(r_2, i)$. Let $J = \langle N, E, root, \lambda \rangle$ be one of the join trees that cover $l = body(r, i)$. We claim that the grounding of $head(r)$ determines the grounding of all literals in J. From this, it follows that $body(r_1, i) = body(r_2, i)$ leading to a contradiction.

We now prove our claim that the grounding of $head(r)$ determines the grounding of all literals in J. Let $V(J, i)$ be the set of all the nodes in J at distance at most i from the root *root*. Furthermore, we denote by $ground(r, r', J, i)$ the set $\{body(r', j) \mid body(r, j) \in V(J, i)\}$. We prove, by induction on i, that for all i and all ground rules r_1 and r_2 instances of r, if $head(r_1) = head(r_2)$, then $ground(r, r_1, J, i) = ground(r, r_2, J, i)$. From this, it follows that the grounding of $head(r)$ determines the grounding of all literals in J.

Base case. For $i = 0$, there is a j such that $V(J, 0) = \{body(r, j)\}$. From this, it follows that $ground(r, r_1, J, 0) = \{body(r_1, j)\}$ and $ground(r, r_2, J, 0) = \{body(r_2, j)\}$. Furthermore, $anc(J, body(r, j)) = \emptyset$. There are two cases depending on whether $body(r, j)$ is a positive literal or not:

1. If $body(r, j)$ is a positive literal of the form $a(\overline{x})$, then $body(r_1, j) = a(\overline{c}_1)$ and $body(r_2, j) = a(\overline{c}_2)$. From the fact that J is strongly connected for $\mathcal{T}$, it follows that there is a set of variables $K \subseteq \{i \mid \overline{x}(i) \in support(a(\overline{x}))\}$ such that $UNQ(a, K) \in \mathcal{T}$. From $support(a(\overline{x})) = vars(head(r)) \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\}$ and $head(r_1) = head(r_2)$, it follows that the values assigned to the variables associated to the indexes in K are the same in r_1 and r_2. From this, $UNQ(a, K) \in \mathcal{T}$, $\mathcal{T}$ has been derived from p, $\{a(\overline{c}_1), a(\overline{c}_2)\} \subseteq ground(p)$, and Proposition B.3, it follows $\overline{c}_1 = \overline{c}_2$ and $ground(r, r_1, J, 0) = ground(r, r_2, J, 0)$.
2. If $body(r, j)$ is a negative literal of the form $\neg a(\overline{x})$, then $body(r_1, j) = \neg a(\overline{c}_1)$ and $body(r_2, j) = \neg a(\overline{c}_2)$. From the fact that J is strongly connected for $\mathcal{T}$, it follows that $vars(\neg a(\overline{x})) \subseteq support(\neg a(\overline{x}))$. From this, $support(\neg a(\overline{x})) = vars(head(r)) \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\}$, and $head(r_1) = head(r_2)$, it follows that the values of the variables in $support(\neg a(\overline{x}))$ are the same in r_1 and r_2. From this, it follows that $\overline{c}_1 = \overline{c}_2$ and $ground(r, r_1, J, 0) = ground(r, r_2, J, 0)$.

Induction Step. Assume that for all $j < i$ and all ground rules $r_1, r_2 \in ground(p, r)$, if $head(r_1) = head(r_2)$, then $ground(r, r_1, J, j) = ground(r, r_2, J, j)$. We now show that $ground(r, r_1, J, i) = ground(r, r_2, J, i)$. Assume, for contradiction's sake, that this is not the case, namely $ground(r, r_1, J, i) \neq ground(r, r_2, J, i)$. From the definition of $ground(r, r', J, i)$, it follows that $ground(r, r_1, J, i) = ground(r, r_1, J, i-1) \cup \{body(r_1, j) \mid body(r, j) \in V(J, i) \setminus V(J, i-1)\}$ and $ground(r, r_2, J, i) = ground(r, r_2, J, i-1) \cup \{body(r_2, j) \mid body(r, j) \in V(J, i) \setminus V(J, i-1)\}$. From this, the induction's hypothesis,

and $ground(r, r_1, J, i) \neq ground(r, r_2, J, i)$, it follows that $\{body(r_1, j) \mid body(r, j) \in V(J, i) \setminus V(J, i-1)\} \neq \{body(r_2, j) \mid body(r, j) \in V(J, i) \setminus V(J, i-1)\}$. Therefore, there is a j such that $body(r, j) \in V(J, i) \setminus V(J, i-1)$ and $body(r_1, j) \neq body(r_2, j)$. There are two cases, depending on whether $body(r, j)$ is a positive literal:

1. If $body(r, j) = a(\overline{x})$, then $body(r_1, j) = a(\overline{c}_1)$, $body(r_2, j) = a(\overline{c}_2)$, and $\overline{c}_1 \neq \overline{c}_2$. From the fact that J is strongly connected for $\mathcal{T}$, it follows that there is a set of variables $K \subseteq \{i \mid \overline{x}(i) \in support(a(\overline{x}))\}$ such that $UNQ(a, K) \in \mathcal{U}$. From $support(a(\overline{x})) = vars(head(r)) \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\} \cup \bigcup_{l \in V(J, i-1)} vars(l)$, $ground(r, r_1, J, i-1) = ground(r, r_2, J, i-1)$ (from the induction's hypothesis), and $head(r_1) = head(r_2)$, it follows that the values assigned to the variables associated to the indexes in K are the same in r_1 and r_2. From this, $UNQ(a, K) \in \mathcal{T}$, $\mathcal{T}$ has been derived from p, $\{a(\overline{c}_1), a(\overline{c}_2)\} \subseteq ground(p)$, and Proposition B.3, it follows that $\overline{c}_1 = \overline{c}_2$ leading to a contradiction.
2. If $body(r, j) = \neg a(\overline{x})$, then $body(r_1, j) = \neg a(\overline{c}_1)$, $body(r_2, j) = \neg a(\overline{c}_2)$, and $\overline{c}_1 \neq \overline{c}_2$. From the fact that J is strongly connected for $\mathcal{T}$, it follows that $vars(\neg a(\overline{x})) \subseteq support(a(\overline{x}))$. From $support(a(\overline{x})) = vars(head(r)) \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\} \cup \bigcup_{l \in V(J, i-1)} vars(l)$, $ground(r, r_1, J, i-1) = ground(r, r_2, J, i-1)$ (from the induction's hypothesis), it follows that the values assigned to the variables in $support(a(\overline{x}))$ are the same in r_1 and r_2. From this, it follows that $\overline{c}_1 = \overline{c}_2$ leading to a contradiction.

Since both cases lead to a contradiction, $ground(r, r_1, J, i) = ground(r, r_2, J, i)$. □

Proposition B.7. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that $\mathbf{dom}$ is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, r be a rule in p, and $\mathcal{T}$ be the template containing all annotations derived from p. If r is weakly connected for $\mathcal{T}$, then for all $r_1, r_2 \in ground(p, r)$, if there is an $1 \leq i \leq |body(r)|$ such that $body(r_1, i) = body(r_2, i)$, then $r_1 = r_2$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that $\mathbf{dom}$ is a finite domain, p be a $(\Sigma, \mathbf{dom})$-ProbLog program, r be a rule in p, and $\mathcal{T}$ be the template containing all annotations derived from p. Furthermore, we assume that r is weakly connected for $\mathcal{T}$, namely there is a join tree $J = \langle N, E, root, \lambda \rangle$ such that (a) J is weakly connected for $\mathcal{T}$, (b) $N \subseteq body^+(r)$, and (c) all literals in $body(r) \setminus N$ are $(r, \mathcal{U}, N)$-strictly guarded. Let $r_1, r_2 \in ground(p, r)$ be two ground rules such that there is an $1 \leq i \leq |body(r)|$ such that $body(r_1, i) = body(r_2, i)$, and let l be the literal $body(r, i)$. There are two cases:

1. $body(r, i) \in N$. Given a node n in a join tree J, we denote by $adjacent(n, i)$, where $i \in \mathbb{N}$, the sub-tree obtained by considering only the nodes reachable from n using at most i edges. We claim that for all j and all nodes in $adjacent(l, j)$, the corresponding ground atoms in r_1 and r_2 are the same. From this and the fact that there is a j such that $adjacent(l, j) = J$, it follows that for all $i \in \{j \mid body(r, j) \in N\}$, $body(r_1, i) = body(r_2, i)$. From this and the fact that the literals in $body(r) \setminus N$ are $(r, \mathcal{T}, N)$-strictly guarded, it follows that for all $i \in \{j \mid body(r, j) \in body(r) \setminus N\}$, $body(r_1, i) = body(r_2, i)$ (since $vars(l) \subseteq \bigcup_{l' \in N \cap body^+(r)} vars(l') \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\}$ for any literal l in $body(r) \setminus N$). Therefore, $body(r_1) = body(r_2)$.
2. $body(r, i) \notin N$. From this and the fact that r is weakly connected, it follows that there a j such that $body(r, j) = a(\overline{x})$, $body(r, j) \in N$, and a $UNQ(a, K) \in \mathcal{T}$ such that $\{\overline{x}(i) \mid i \in K\} \subseteq vars(body(r, i))$. From this, $\mathcal{T}$ has been derived from p, $body(r_1, i) = body(r_2, i)$, $\{body(r_1, j), body(r_2, j)\} \subseteq ground(p)$, and Proposition B.3, it follows that $body(r_1, j) = body(r_2, j)$. We proved above that if $body(r_1, j) = body(r_2, j)$ and $body(r, j) \in N$, then $body(r_1) = body(r_2)$. Therefore, $body(r_1) = body(r_2)$.

From $vars(head(r)) \subseteq \bigcup_{l \in body^+(r)} vars(l)$ and $body(r_1) = body(r_2)$, it follows that $head(r_1) = head(r_2)$. Therefore, $r_1 = r_2$.

We now prove, by induction on j, that for all j and all nodes in $adjacent(l, j)$, the corresponding ground atoms in r_1 and r_2 are the same.

Base Case. The sub-tree $adjacent(l, 0)$ contains only the node $l = body(r, i)$. Since $body(r_1, i) = body(r_2, i)$, the claim holds for the base case.

Induction step. Assume now that the claim holds for all $j' < j$. We now prove that the claim holds also for j. The sub-tree $adjacent(l, j)$ is obtained by extending $adjacent(l, j-1)$ with either edges of the form $n_1 \xrightarrow{L_1} n_1'$, where $n_1' \in adjacent(l, j-1)$, or $n_2' \xrightarrow{L_2} n_2$, where $n_2' \in adjacent(l, j-1)$. In both cases, from the induction hypothesis, the ground literals corresponding to n_1' and n_2' are l_1' and l_2' and they are the same in r_1 and r_2. From the definition of weakly connected join tree, there are variables $K_1 \subseteq L_1$ and $K_2 \subseteq L_2$ such that $UNQ(pred(n_1), K_1) \in \mathcal{T}$ and $UNQ(pred(n_2), K_2) \in \mathcal{T}$. From this, $\mathcal{T}$ has been derived from p, the fact that the value of K_1 and K_2 are fixed by l_1' and l_2', and Proposition B.3, it follows that the ground atoms corresponding to n_1 and n_2 are the same in r_1 and r_2 (because the values in K_1 and K_2 determines all values in n_1 and n_2). □

B.1.4 Acyclicity Proof

We now prove our first key result, namely that the ground graph associated to an acyclic ProbLog program is a forest of *poly-trees*, i.e., its undirected version does not contain simple cycles (which are cycles without repetitions of edges and vertices other than the starting and ending vertices).

Theorem B.1. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain, and p be a $(\Sigma, \mathbf{dom})$-acyclic* ProbLog *program. The graph $gg(p)$ is a forest of poly-trees, i.e., its undirected version does not contain simple cycles.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, and p be a $(\Sigma, \mathbf{dom})$-acyclic ProbLog program. We prove that the undirected version of $gg(p)$ is acyclic. Assume, for contradiction's sake, that this is not the case, namely there is a simple cycle $C := n_1 \to n_2 \to \ldots \to n_k \to n_1$ in the undirected version of $gg(p)$. There are two cases: (1) there is a directed simple cycle in $gg(p)$ that directly corresponds to C, or (2) there are n directed and reversed paths $P_1, \ldots, P_n$ in $gg(p)$ that induce a simple cycle C in the undirected version of $gg(p)$ (and $P_1 \cdot \ldots \cdot P_n$ does not correspond to any directed simple cycle in $gg(p)$). From the first case, it follows that there is a directed cycle in $graph(p)$, whereas from the second case, it follows that there is an undirected cycle in $graph(p)$.

Directed Cycle. Assume that $C := n_1 \to n_2 \to \ldots \to n_k \to n_1$ directly corresponds to a directed cycle in $gg(p)$. Then, the cycle has form $a_1(\overline{c}_1) \xrightarrow{r_1,s_1,i_1} a_2(\overline{c}_2) \ldots a_n(\overline{c}_n) \xrightarrow{r_n,s_n,i_n} a_{n+1}(\overline{c}_{n+1})$, where $n = k$ and $a_{n+1}(\overline{c}_{n+1}) = a_1(\overline{c}_1)$. From this, it follows that there are rules $r_1, \ldots, r_n$ and ground rules $s_1, \ldots, s_n$ such that: (1) there is a directed cycle $a_1 \xrightarrow{r_1,i_1} a_2 \xrightarrow{r_2,i_2} \ldots \xrightarrow{r_{n-1},i_{n-1}} a_n \xrightarrow{r_n,i_n} a_1$ in p's dependency graph $graph(p)$, and (2) for all $1 \leq i \leq n$, $s_i \in ground(p, r_i)$ and $head(s_i) = a_{i+1}(c_{i+1})$. From this, it follows that p's dependency graph $graph(p)$ contains a directed cycle $C' = a_1 \xrightarrow{r_1,i_1} a_2 \xrightarrow{r_2,i_2} \ldots \xrightarrow{r_{n-1},i_{n-1}} a_n \xrightarrow{r_n,i_n} a_1$. Note that C' may contain loops (i.e., it is not simple). Since p is acyclic and C' is a directed cycle in $graph(p)$, it follows that there is a directed unsafe structure S that covers C' and is $\mathcal{T}$-guarded, where $\mathcal{T}$ is the template containing all annotations that can be derived from p. Without loss of generality, we assume that $S = C'$. From S covers C' and S is $\mathcal{T}$-guarded, it follows that there is an ordering annotation $ORD(O) \in \mathcal{T}$ such that there are integers $1 \leq y_1 < y_2 < \ldots < y_e = n$, literals $o_1(\overline{x}_1), \ldots, o_e(\overline{x}_e)$ (where $o_j \in O$ and $|\overline{x}_j| = |o_j|$), a non-empty set $K \subseteq \{1, \ldots, |pr_1|\}$, and a bijection $\nu : K \to \{1, \ldots, {}^{|o_1|}/_2\}$ such that for each $0 \leq k < e$, (1) $pr_{y_k} \xrightarrow{r_{y_k},i_{y_k}} \ldots \xrightarrow{r_{y_{k+1}-1},i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν-downward connects to $o_{k+1}(\overline{x}_{k+1})$, and (2) $pr_{y_{k+1}-1} \xrightarrow{r_{y_{k+1}-1},i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν'-upward connects to $o_{k+1}(\overline{x}_{k+1})$, where $\nu'(i) = \nu(x) + {}^{|o_1|}/_2$ for all $1 \leq i \leq {}^{|o_1|}/_2$, and $y_0 = 1$. By applying Proposition B.4 and Proposition B.5 to the paths $pr_{y_k} \xrightarrow{r_{y_k},i_{y_k}} \ldots \xrightarrow{r_{y_{k+1}-1},i_{y_{k+1}-1}} pr_{y_{k+1}}$, which ν-downward connects to $o_{k+1}(\overline{x}_{k+1})$, for each $0 \leq k < e$, it follows that (1) ground atoms $o_1(\overline{b}_1, \overline{b}_2), \ldots, o_e(\overline{b}_{|K|}, \overline{b}_{|K|+1})$ are in $ground(p)$, and (2) for all $1 \leq w \leq {}^{|pr|}/_2$, both $\overline{b}_1(w) = args(body(s_1, i_1))(\nu(w))$ and $\overline{b}_{|K|+1}(w) = args(head(s_n))(\nu'(w))$ hold. From this, $ORD(O) \in \mathcal{T}$, $\mathcal{T}$ has been derived from p, and Proposition B.1, it follows that $\overline{b}_1 \neq \overline{b}_{|K|+1}$. From this and $\overline{b}_1(w) = args(body(s_1, i_1))(\nu(w))$ and $\overline{b}_{|K|+1}(w) = args(head(s_n))(\nu'(w))$ for all $1 \leq w \leq {}^{|pr|}/_2$, it follows that $body(s_1, i_1) \neq head(s_n)$. This contradicts $head(s_n) = a(\overline{c}_1)$ and $body(s_1, i_1) = a(\overline{c}_1)$ (which directly follows from the existence of the cycle C).

Undirected Cycle. Assume that C does not directly correspond to any directed simple cycle in $gg(p)$. From this, it follows that there are n directed and reversed paths $P_1 \cdot \ldots \cdot P_n$ in $gg(p)$ such that $P_1, \ldots, P_n$ correspond to the simple cycle C in the undirected version of $gg(p)$. From this, it follows that $P_1 \cdot \ldots \cdot P_n$ form an undirected cycle in $gg(p)$. Therefore:

1. For $1 \leq j \leq n$ such that P_j is a directed path, there is a directed path $D_j := a_1 \xrightarrow{r_1,i_1} a_2 \xrightarrow{r_2,i_2} \ldots \xrightarrow{r_{n_j-1},i_{n_j-1}} a_{n_j}$ in p's dependency graph $graph(p)$, where P_j is $a_1(\overline{c}_1) \xrightarrow{r_1,s_1,i_1} a_2(\overline{c}_2) \ldots a_{n_j-1}(\overline{c}_{n_j-1}) \xrightarrow{r_{n_j-1},s_{n_j-1},i_{n_j-1}} a_{n_j}(\overline{c}_{n_j})$.
2. For $1 \leq j \leq n$ such that P_j is a reversed path, there is a reversed path $D_j := a_1 \xleftarrow{r_1,i_1} a_2 \xleftarrow{r_2,i_2} \ldots \xleftarrow{r_{n_j-1},i_{n_j-1}} a_{n_j}$ in p's dependency graph $graph(p)$, where P_j is $a_1(\overline{c}_1) \xleftarrow{r_1,s_1,i_1} a_2(\overline{c}_2) \ldots a_{n_j-1}(\overline{c}_{n_j-1}) \xleftarrow{r_{n_j-1},s_{n_j-1},i_{n_j-1}} a_{n_j}(\overline{c}_{n_j})$.
3. $D_1 \cdot \ldots \cdot D_n$ form an undirected cycle in $graph(p)$.
4. $D_1 \cdot \ldots \cdot D_n$ is not a directed cycle. Indeed, if $D_1 \cdot \ldots \cdot D_n$ is a directed cycle, then either C would correspond to a directed simple cycle in $gg(p)$ (contradicting our assumption that C does not directly correspond to any directed simple cycle in $gg(p)$) or C would contain loops (contradicting our assumption that C is a simple cycle).

Since p is acyclic and $D_1 \cdot \ldots \cdot D_n$ forms an undirected cycle in $graph(p)$ that is not a directed cycle, it follows that there is an unsafe structure S that covers $D_1 \cdot \ldots \cdot D_n$ and is $\mathcal{T}$-guarded, where $\mathcal{T}$ is the set of all annotations derived from p. We assume that there are values i, a, b, c such that:

1. $S = \langle D_a, D_b, D_c, U \rangle$,
2. $a = i$, $b = (i+1)\%n$, and $c = (i+2)\%n$ (i.e., P_a, P_b, and P_c are adjacent in the cycle), and
3. U is the undirected path containing all D_i's that are different from D_a, D_b, and D_c.

Note that the previous assumption is without loss of generality. Since S covers C, we can always pick the paths in $\mathtt{gg}(p)$ inducing C in such a way that they match the guarded structure S.

Let P_a be $a_1(\overline{a}_1) \xrightarrow{r_1,s_1,i_1} a_2(\overline{a}_2) \xrightarrow{r_2,s_2,i_2} \ldots \xrightarrow{r_{n_a-1},s_{n_a-1},i_{n_a-1}} a_{n_a}(\overline{a}_{n_a})$, P_b be $b_1(\overline{b}_1) \xrightarrow{r'_1,s'_1,i'_1} b_2(\overline{b}_2) \xrightarrow{r'_2,s'_2,i'_2} \ldots \xrightarrow{r'_{n_b-1},s'_{n_b-1},i'_{n_b-1}} b_{n_b}(\overline{b}_{n_b})$, and P_c be the path $c_1(\overline{c}_1) \xrightarrow{r''_1,s''_1,i''_1} c_2(\overline{c}_2) \xrightarrow{r''_2,s''_2,i''_2} \ldots \xrightarrow{r''_{n_c-1},s''_{n_c-1},i''_{n_c-1}} c_{n_c}(\overline{c}_{n_c})$. From $S = \langle D_a, D_b, D_c, U \rangle$ and S covers C, it follows that $a_1(\overline{a}_1) = b_1(\overline{b}_1)$ and $b_{n_b}(\overline{b}_{n_b}) = c_{n_c}(\overline{c}_{n_c})$. From $S = \langle D_a, D_b, D_c, U \rangle$, S covers C, and S is $\mathcal{T}$-guarded, there are two cases:

1. (D_a, D_b) is $\mathcal{T}$-head guarded. Therefore, D_a and D_b are non-empty. There are two cases:
   (a) $D_a = D_b$. From this, it follows that (1) P_a and P_b are directed paths in $gg(p)$, (2) $a_1(\overline{a}_1) = b_1(\overline{b}_1)$, (3) $n_a = n_b$, and (4) D_a and D_b are head-connected. From this and (D_a, D_b) is head-guarded, it follows that the rules in D_a and D_b are weakly connected for $\mathcal{T}$. We claim that $s_h = s'_h$ for any $1 \leq h \leq n_a$. From this, it follows that $P_a = P_b$. This contradicts the fact that $P_1, \ldots, P_n$ induces a simple cycle in the undirected version of $gg(p)$ (because the cycle is not simple).
   We now prove, by induction on d, that $s_d = s'_d$ and $a_d(\overline{a}_d) = b_d(\overline{b}_d)$.

   Base Case. Assume that $d = 1$. From $r_1 = r'_1$, r_1 is weakly connected for $\mathcal{T}$, $body(s_1, i_1) = body(s'_1, i'_1) = a_1(\overline{c}_1)$, $s_d \in ground(p, r_d)$, $s'_d \in ground(p, r'_d)$, and Proposition B.7, it follows that $s_1 = s'_1$. From this, $a_1(\overline{a}_1) = head(s_1)$, and $b_1(\overline{b}_1) = head(s'_1)$, it follows that $a_1(\overline{a}_1) = b_1(\overline{b}_1)$.

   Induction Step. Assume that the claim holds for all $d' < d$. We now prove that $s_d = s'_d$ and $a_d(\overline{a}_d) = b_d(\overline{b}_d)$ hold as well. From the induction's hypothesis, it follows that $head(s_{d-1}) = head(s'_{d-1})$. From this and gg's definition, it follows that $body(s_d, i_d) = body(s'_d, i'_d)$. From this, $r_d = r'_d$, $i_d = i'_d$, r_d is weakly connected for $\mathcal{T}$, $s_d \in ground(p, r_d)$, $s'_d \in ground(p, s_d)$, and Proposition B.7, it follows that $s_d = s'_d$. From this, $a_d(\overline{a}_d) = head(s_d)$, and $b_d(\overline{b}_d) = head(s'_d)$, it follows that $a_d(\overline{a}_d) = b_d(\overline{b}_d)$.
   (b) $D_a \neq D_b$. From this and (D_a, D_b) are head-guarded, it follows that there is an annotation $DIS(pr, pr') \in \mathcal{T}$ a set $K \subseteq \{1, \ldots, |a|\}$, and a bijection $\nu : K \to \{1, \ldots, |pr|\}$ such that D_a ν-downward links to pr and D_a ν-downward links to pr'. From D_a ν-downward links to pr, P_a is a ground instance of D_a, and Proposition B.4, it follows that there is a positive literal $pr(\overline{v})$ in the body of one of the ground rules such that $a_1(\overline{a}_1)(k) = \overline{v}(\nu(K))$ for any $k \in K$. From this and the definition of $ground$, it follows that $pr(\overline{v}) \in ground(p)$. From D_b ν-downward links to pr', P_b is a ground instance of D_b, and Proposition B.4, it follows that there is a positive literal $pr'(\overline{v}')$ in the body of one of the ground rules such that $b_1(\overline{b}_1)(k) = \overline{v}'(\nu(K))$ for any $k \in K$. From this and the definition of $ground$, it follows that $pr'(\overline{v}') \in ground(p)$. Finally, from the fact that P_a and P_b are head-connected, it follows that $a_1(\overline{a}_1) = b_1(\overline{b}_1)$. From this, $a_1(\overline{a}_1)(k) = \overline{v}(\nu(K))$ for any $k \in K$, and $b_1(\overline{b}_1)(k) = \overline{v}'(\nu(K))$ for any $k \in K$, it follows that $\overline{v} = \overline{v}'$. From this, $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$. This contradicts Proposition B.2, since $DIS(pr, pr') \in \mathcal{T}$ and $\mathcal{T}$ has been derived from p.
2. (D_b, D_c) is $\mathcal{T}$-tail guarded. Therefore, D_b and D_c are non-empty. There are two cases:
   (a) $D_b = D_c$. From this, it follows that (1) P_c and P_b are directed paths in $gg(p)$, (2) $c_{n_c}(\overline{c}_{n_c}) = b_{n_b}(\overline{b}_{n_b})$, (3) $n_b = n_c$, and (4) D_b and D_c are tail-connected. From this and (D_b, D_c) is tail-guarded, it follows that (1) D_c and D_b are tail-connected, and (2) the rules in D_c and D_b are strongly connected for $\mathcal{T}$. We claim that $s''_h = s'_h$ for any $1 \leq h \leq n_c$. From this, it follows that $P_c = P_b$. This contradicts the fact that $P_1, \ldots, P_n$ induces a simple cycle in the undirected version of $gg(p)$ (because the cycle is not simple).
   We now prove, by induction on d, that $s''_{n_b-d} = s'_{n_b-d}$ and $head(s''_{n_b-d}) = head(s'_{n_b-d})$.

   Base Case. Assume $d = 0$. From $r''_{n_b} = r'_{n_b}$, r'_{n_b} is strongly connected for $\mathcal{T}$, $head(s''_{n_b}) = head(s'_{n_b}) = b_{n_b}(\overline{b}_{n_b})$, $s''_{n_b} \in ground(p, r''_d)$, $s'_d \in ground(p, r'_d)$, and Proposition B.6, it follows that $s_1 = s'_1$. From this, $a_1(\overline{a}_1) = head(s_1)$, and $b_1(\overline{b}_1) = head(s'_1)$, it follows that $b_{n_b}(\overline{b}_{n_b}) = c_{n_b}(\overline{c}_{n_b})$.

   Induction Step. Assume that the claim holds for all $d' < d$. We now prove that $s''_{n_b-d} = s'_{n_b-d}$ and $c_{n_b-d}(\overline{c}_{n_b-d}) = b_{n_b-d}(\overline{b}_{n_b-d})$ hold as well. From the induction's hypothesis, it follows that $s''_{n_b-d+1} = s'_{n_b-d+1}$. From this and gg's definition, it follows that

$head(s''_{n_b-d}, i''_{n_b-d}) = head(s'_{n_b-d}, i'_{n_b-d})$. From this, $r''_{n_b-d} = r'_{n_b-d}$, $i''_{n_b-d} = i'_{n_b-d}$, r''_{n_b-d} is strongly connected for $\mathcal{T}$, $s''_{n_b-d} \in ground(p, r''_{n_b-d})$, $s'_{n_b-d} \in ground(p, s_{n_b-d})$, and Proposition B.6, it follows that $s''_{n_b-d} = s'_{n_b-d}$. From this, $c_{n_b-d}(\overline{c}_{n_b-d}) = head(s''_{n_b-d})$, and $b_{n_b-d}(\overline{b}_{n_b-d}) = head(s'_{n_b-d})$, it follows that $c_{n_b-d}(\overline{c}_{n_b-d}) = b_{n_b-d}(\overline{b}_{n_b-d})$.

(b) $D_b \neq D_c$. From this and (D_b, D_c) is tail-guarded, it follows that there is an annotation $DIS(pr, pr') \in \mathcal{T}$, a set $K \subseteq \{1, \ldots, |a|\}$, and a bijection $\nu : K \to \{1, \ldots, |pr|\}$, such that D_b ν-upward links to pr and D_c ν-upward links to pr'. From D_b ν-upward links to pr, P_b is a path in the ground graph corresponding to D_b, and Proposition B.5, it follows that a positive literal $pr(\overline{v})$ in the body of one of the ground rules such that $b_{n_b}(\overline{b}_{n_b})(k) = \overline{v}(\nu(K))$ for any $k \in K$. From this and the definition of *ground*, it follows that $pr(\overline{v}) \in ground(p)$. From D_c ν-upward links to pr', P_c is a path in the ground graph corresponding to D_c, and Proposition B.5, it follows that a positive literal $pr'(\overline{v}')$ in the body of one of the ground rules such that $c_{n_c}(\overline{c}_{n_c})(k) = \overline{v}'(\nu(K))$ for any $k \in K$. From this and the definition of *ground*, it follows that $pr'(\overline{v}') \in ground(p)$. Finally, from the fact that P_b and P_c are tail-connected, it follows that $b_{n_b}(\overline{b}_{n_b}) = c_{n_c}(\overline{c}_{n_c})$. From this, $b_{n_b}(\overline{b}_{n_b})(k) = \overline{v}(\nu(K))$ for any $k \in K$, and $c_{n_c}(\overline{c}_{n_c})(k) = \overline{v}'(\nu(K))$ for any $k \in K$, it follows that $\overline{v} = \overline{v}'$. Therefore, $pr(\overline{v}) \in ground(p)$ and $pr'(\overline{v}) \in ground(p)$. This contradicts Proposition B.2 since $DIS(pr, pr') \in \mathcal{T}$ and $\mathcal{T}$ has been derived from p.

This completes the proof of our claim. □

B.1.5 Auxiliary Results

Here we prove two auxiliary results that help in establishing that programs are acyclic. In particular, Proposition B.8 states pre-conditions that allows reducing the guardedness of a complex undirected structure to the guardedness of simpler structures. Similarly, Proposition B.9 states pre-conditions that allows reducing the guardedness of a complex directed structure to the guardedness of a sequence of simpler structures. Note that Proposition B.9 can be easily extended to support (1) different forms of cycle combination, and (2) combinations of non-self-loop cycles.

Proposition B.8. *Let p be a* ProbLog *program, $\mathcal{T}$ be the template containing all annotations that can be derived from p, C be an undirected cycle in $graph(p)$, $S = \langle D_1, D_2, D_3, U\rangle$ be an undirected unsafe structure, and U'_1 and U'_2 be undirected cycles in $graph(p)$. If (1) C is equivalent to $D_1{\cdot}U'_1{\cdot}U{\cdot}U'_2{\cdot}D_3{\cdot}D_2$, (2) S is $\mathcal{T}$-guarded, then there is an undirected unsafe structure that covers C and is $\mathcal{T}$-guarded.*

Proof. Let p be a ProbLog program, $\mathcal{T}$ be the template containing all annotations that can be derived from p, C be an undirected cycle in $graph(p)$, and $S = \langle D_1, D_2, D_3, U\rangle$ be an undirected structure, and U'_1 and U'_2 be undirected cycles in $graph(p)$. We assume that (1) C is equivalent to $D_1{\cdot}U'_1{\cdot}U{\cdot}U'_2{\cdot}D_3{\cdot}D_2$, (2) S is $\mathcal{T}$-guarded. We define the undirected unsafe structure $S' = \langle D_1, D_2, D_3, (U'_1{\cdot}U{\cdot}U'_2)\rangle$. From our assumption C is equivalent to $D_1{\cdot}U'_1{\cdot}U{\cdot}U'_2{\cdot}D_3{\cdot}D_2$. Thus, S' covers C. Furthermore, since S and S' agree on all directed paths, which are the only ones that determine whether an undirected structure is guarded, it follows that S' is $\mathcal{T}$-guarded □

In Proposition B.9, use the notion of a directed cycle C guarded for a set of predicates O and a mapping ν. This notion is similar to the notion of guarded directed unsafe structure restricted O and ν.

Proposition B.9. *Let p be a* ProbLog *program, $C_1 = pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_{n-1},i_{n-1}} pr_n \xrightarrow{r_n,i_n} pr_1$ be a directed cycle in $graph(p)$, O be a set of predicate symbols, $pr_1 \xrightarrow{r_{n+1},i_{n+1}} pr_1$, and o be a predicate symbol such that for all $o' \in O$, $|o| = |o'|$, K be a non-empty set $K \subseteq \{1, \ldots, |pr_1|\}$, and a bijection $\nu : K \to \{1, \ldots, {}^{|o|}/_2\}$. If (1) C_1 is guarded for O and ν, and (2) $pr_1 \xrightarrow{r_{n+1},i_{n+1}} pr_1$ is guarded for $\{o\}$ and ν, then $pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_{n-1},i_{n-1}} pr_n \xrightarrow{r_n,i_n} pr_1 \xrightarrow{r_{n+1},i_{n+1}} pr_1$ is guarded for $O \cup \{o\}$ and ν.*

Proof. Let p be a ProbLog program, $C_1 = pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_{n-1},i_{n-1}} pr_n \xrightarrow{r_n,i_n} pr_1$ be a directed cycle in $graph(p)$, O be a set of predicate symbols, $pr_1 \xrightarrow{r_{n+1},i_{n+1}} pr_1$, and o be a predicate symbol such that for all $o' \in O$, $|o| = |o'|$, K be a non-empty set $K \subseteq \{1, \ldots, |pr_1|\}$, ν be a bijection $\nu : K \to \{1, \ldots, {}^{|o|}/_2\}$ and ν' be the bijection $\nu'(i) = \nu(x) + {}^{|o|}/_2$ for all $1 \leq i \leq {}^{|o|}/_2$. Furthermore, we assume that (1) C_1 is guarded for O and ν, and (2) $pr_1 \xrightarrow{r_{n+1},i_{n+1}} pr_1$ is guarded for $\{o\}$ and ν.

First, we rewrite $C_1, pr_1 \xrightarrow{r_{n+1},i_{n+1}} pr_1$ as $pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_{n-1},i_{n-1}} pr_n \xrightarrow{r_n,i_n} pr_{n+1} \xrightarrow{r_{n+1},i_{n+1}} pr_1$. From C_1 is guarded for O_1, it follows that there are integers $1 \leq y_1 < y_2 < \ldots < y_e \leq n$ such that $y_e = n$ and literals $o_1(\overline{x}_1), \ldots, o_e(\overline{x}_e)$ (where $o_j \in O$ and $|\overline{x}_j| = |o_j|$), such that for each $0 \leq k < e$, (1) $pr_{y_k} \xrightarrow{r_{y_k},i_{y_k}} \ldots \xrightarrow{r_{y_{k+1}-1},i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν-downward connects to $o_{k+1}(\overline{x}_{k+1})$, and

(2) $pr_{y_{k+1}-1} \xrightarrow{r_{y_{k+1}-1}, i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν'-upward connects to $o_{k+1}(\overline{x}_{k+1})$, where $y_0 = 1$. Furthermore, from $pr_1 \xrightarrow{r_{n+1}, i_{n+1}} pr_1$ is guarded for $\{o\}$ and ν, it follows that there is a literal $o(\overline{x})$ such that (3) $pr_{n+1} \xrightarrow{r_{n+1}, i_{n+1}} pr_1$ ν-downward connects to $o(\overline{x})$, and (4) $pr_{y_{n+1}} \xrightarrow{r_{n+1}, i_{n+1}} pr_{y_1}$ ν'-upward connects to $o_{k+1}(\overline{x}_{k+1})$. From (1)–(4), it therefore follows that $pr_1 \xrightarrow{r_1, i_1} \ldots \xrightarrow{r_{n-1}, i_{n-1}} pr_n \xrightarrow{r_n, i_n} pr_{n+1} \xrightarrow{r_{n+1}, i_{n+1}} pr_1$ is guarded for ν and $O \cup \{o\}$. Indeed, there are integers $1 \leq y_1 < y_2 < \ldots < y_e < y_{e+1} \leq n+1$ such that $y_{e+1} = n+1$ and literals $o_1(\overline{x}_1), \ldots, o_e(\overline{x}_e), o_{e+1}(\overline{x}_{e+1})$ (where $o_j \in O \cup \{o\}$ and $|\overline{x}_j| = |o_j|$) such that for each $0 \leq k < e+1$, (1) $pr_{y_k} \xrightarrow{r_{y_k}, i_{y_k}} \ldots \xrightarrow{r_{y_{k+1}-1}, i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν-downward connects to $o_{k+1}(\overline{x}_{k+1})$, and (2) $pr_{y_{k+1}-1} \xrightarrow{r_{y_{k+1}-1}, i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν'-upward connects to $o_{k+1}(\overline{x}_{k+1})$. □

B.2 Encoding's acyclicity

Here we prove that Algorithm 2 produces a Bayesian Network that is a forest of poly-trees. Note that Algorithm 2 produces a forest of poly-trees and not just a single poly-tree because some predicates symbols may be independent. For instance, the program consisting of the rules $a(x) \leftarrow b(x)$ and $c(x) \leftarrow d(x)$ corresponds to two poly-trees (one per rule).

Proposition B.10. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain, p be a $(\Sigma, \mathbf{dom})$-relaxed acyclic* ProbLog *program, and W be a witness for p's acyclicity. The Bayesian Network $\langle N, E, CPT \rangle$ produced by Algorithm 2 on input p is a forest of poly-trees.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-relaxed acyclic ProbLog program, and W be a witness for p's acyclicity. Furthermore, let $BN = \langle N, E, CPT \rangle$ be the Bayesian Network produced by Algorithm 2 on input p. There is a one-to-one mapping from paths in the ground graph $gg(\alpha(\beta_\mu(p))))$ and paths in the Bayesian Network produced by Algorithm 2. Namely, there is a path from $a(\overline{c})$ to $a'(\overline{c}')$ in $gg(\alpha(\beta_W(p))))$ iff there is a path from $X[a(\overline{c})]$ to $X[a'(\overline{c}')]$ in BN. Assume that there is a cycle in the undirected version of BN. From this, it follows that there is an undirected cycle in $gg(\alpha(\beta_W(p))))$. This, however, contradicts Theorem B.1 (since p is relaxed acyclic, then $\alpha(\beta_W(p)))$ is acyclic and there are no undirected cycles in $gg(\alpha(\beta_W(p)))))$. □

B.3 Encoding's Correctness

Here we prove the correctness of our encoding.

B.3.1 Terminology and Notation

Before presenting our correctness proof, we introduce some notation. Let p be a relaxed acyclic ProbLog program, W be a witness for p, and $BN = \langle N, E, CPT \rangle$ be the Bayesian Network produced by Algorithm 2 having p and W as inputs. We say that the *kernel of BN*, denoted $K(BN)$, is the set of variables $\{X[r, \emptyset, a(\overline{c})] \in N \mid \exists v.\, r = v{::}a(\overline{c})\}$. Note that all nodes $n \in K(BN)$ are boolean random variables by construction, namely $D(n) = \{\top, \bot\}$. Given a BN-total assignment ν, we say that ν is *consistent* iff for all variables $n \in N \setminus K(BN)$, $cpt(n)(\nu(P_1), \ldots, \nu(P_m), \nu(n)) = 1$, where $p(n) = \{P_1, \ldots, P_m\}$. Furthermore, we say that ν is a *model for a state s*, written $\nu \models s$, iff $a(\overline{c}) \in s \Leftrightarrow \nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$.

Let r be a rule whose head is h and whose body consists of literals $l_1, \ldots, l_k$ and $\langle \overline{v_1}, \ldots, \overline{v_n}, \overline{v_{n+1}} \rangle$ be a tuple, where $k \leq n$. We say that s *matches* $\langle \overline{v_1}, \ldots, \overline{v_n}, \overline{v_{n+1}} \rangle$ iff the following conditions hold: (1) $\overline{v_{n+1}} = h\uparrow_p$, (2) for all $1 \leq i \leq k$, if $l_i \in body^+(r)$, then $\overline{v_i} = l_i\uparrow_p$, and (3) for all $1 \leq i \leq k$, if $l_i \in body^-(r)$, then $\overline{v_i} = \bot$ or $\overline{v_i} \neq l_i\uparrow_p$.

B.3.2 Exact Grounding

We now introduce the exact grounding of a ProbLog program p given a p-probabilistic assignment. The exact grounding encodes the ProbLog semantics.

Let p be a ProbLog program and f be a p-probabilistic assignment. Furthermore, let μ be the mapping from predicate symbols in p to $\mathbb{N}$: $\mu(a) = max_{t \in paths(p,a)}(w(t))$, where $paths(p, a)$ is the set

of all directed paths that ends in a in p's dependency graph and the weight of each path is

$$w(t) = \begin{cases} 0 & \text{if } t = \epsilon \\ w(t') & \text{if } t = a_1 \xrightarrow{r_1,i_1} a_2 \xrightarrow{r_2,i_2} \dots \xrightarrow{r_n,i_n} a_{n+1} \\ & \quad \wedge t' = a_2 \xrightarrow{r_2,i_2} \dots \xrightarrow{r_n,i_n} a_{n+1} \\ & \quad \wedge body(r_1,i_1) \in \mathcal{A}^+_{\Sigma,\mathbf{dom}} \\ 1 + w(t') & \text{if } t = a_1 \xrightarrow{r_1,i_1} a_2 \xrightarrow{r_2,i_2} \dots \xrightarrow{r_n,i_n} a_{n+1} \\ & \quad \wedge t' = a_2 \xrightarrow{r_2,i_2} \dots \xrightarrow{r_n,i_n} a_{n+1} \\ & \quad \wedge body(r_1,i_1) \notin \mathcal{A}^+_{\Sigma,\mathbf{dom}} \end{cases}$$

We remark that μ is a valid stratification for the program p. From more details about logic programming with stratified negation we refer the reader to [10].

The functions $g_f(p,j,i)$, $g_f(p,j)$, and $g_f(p)$ are defined as follows:

$$\begin{aligned}
g_f(p,0,0) =& \{a(\overline{c}) \mid a(\overline{c}) \in p \wedge \mu(a) = 0\} \cup \\
& \{a(\overline{c}) \mid \exists v.\ v{::}a(\overline{c}) \in p \wedge f(v{::}a(\overline{c})) = \top \wedge \mu(a) = 0\} \\
g_f(p,0,i) =& g_f(p,0,i-1) \cup \\
& \{head(r)\Theta \mid \Theta \in ASGN(r) \wedge r \in p \wedge \forall l \in body^+(r).\ l\Theta \in g_f(p,0,i-1) \wedge \\
& \quad body^-(r) = \emptyset \wedge \mu(pred(head(r))) = 0\} \\
g_f(p,j,0) =& g_f(p,j-1) \cup \{a(\overline{c}) \mid a(\overline{c}) \in p \wedge \mu(a) = j\} \cup \\
& \{a(\overline{c}) \mid \exists v.\ v{::}a(\overline{c}) \in p \wedge f(v{::}a(\overline{c})) = \top \wedge \mu(a) = j\} \\
g_f(p,j,i) =& g_f(p,j,i-1) \cup \\
& \{head(r)\Theta \mid \Theta \in ASGN(r) \wedge r \in p \wedge \forall l \in body^+(r).\ l\Theta \in g_f(p,j,i-1) \wedge \\
& \quad \forall l \in body^-(r).\ atom(l)\Theta \notin g_f(p,j-1) \wedge \mu(pred(head(r))) = j\} \\
g_f(p,j) =& \bigcup_{i \in \mathbb{N}} g_f(p,j,i) \\
g_f(p) =& \bigcup_{j \in \mathbb{N}} g_f(p,j)
\end{aligned}$$

Furthermore, the functions $g_f(p,r,j,i)$, $g_f(p,r,j)$, and $g_f(p,r)$ are defined as follows:

$$\begin{aligned}
g_f(p,a(\overline{c}),j,i) =& \{a(\overline{c}) \mid a(\overline{c}) \in p \wedge \mu(a) = j\} \\
g_f(p,v{::}a(\overline{c}),j,i) =& \{a(\overline{c}) \mid v{::}a(\overline{c}) \in p \wedge f(v{::}a(\overline{c})) = \top \wedge \mu(a) = j\} \\
g_f(p,r,0,0) =& \emptyset \\
g_f(p,r,0,i) =& g_f(p,r,0,i-1) \cup \\
& \{h\Theta \leftarrow l_1\Theta,\dots,l_n\Theta \mid r = h \leftarrow l_1,\dots,l_n \wedge \Theta \in ASGN(r) \wedge \\
& \quad \forall l \in body^+(r).\, l\Theta \in g_f(p,0,i-1) \wedge body^-(r) = \emptyset \wedge \mu(pred(h)) = 0\} \\
g_f(p,r,j,0) =& g_f(p,r,j-1) \\
g_f(p,r,j,i) =& g_f(p,r,j,i-1) \cup \\
& \{h\Theta \leftarrow l_1\Theta,\dots,l_n\Theta \mid r = h \leftarrow l_1,\dots,l_n \wedge \Theta \in ASGN(r) \wedge \\
& \quad \forall l \in body^+(r).\, l\Theta \in g_f(p,j,i-1) \wedge \\
& \quad \forall l \in body^-(r).\, atom(l)\Theta \notin g_f(p,j-1) \wedge \mu(pred(h)) = j\} \\
g_f(p,r,j) =& \bigcup_{i \in \mathbb{N}} g_f(p,r,j,i) \\
g_f(p,r) =& \bigcup_{j \in \mathbb{N}} g_f(p,r,j)
\end{aligned}$$

Proposition B.11 follows directly from g_f's definition and the semantics of stratified logic programs.

Proposition B.11. *For any* ProbLog *program p and any p-probabilistic assignment f, $g_f(p) = [\![instance(p,f)]\!]$.*

B.3.3 Auxiliary results about safe annotated disjunctions

Given a program p, a safe CPT schema $\langle \pi_H, \pi_V, \mu \rangle$, and a predicate pr such that $\mu(pr) \neq \emptyset$, each partition in $\pi_H(p, pr)$ identifies a row in the CPT associated with a ground atom (whose predicate is pr), whereas the first sequence produced by π_V identifies the variables determining the CPT's result. We now present some results showing the correctness of the requirements for the safety of CPT-schemas.

Proposition B.12 states that whenever two ground rules have the same head once we remove the constants through the transformation β, the corresponding original rules belong to the same set RR in $\pi_H(p, pr)$.

Proposition B.12. *Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1, \ldots, |pr|\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, $r'_1 \in ground(p, r_1)$ and $r'_2 \in ground(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in row_1 and row_2, where $\pi_V(r_1) = \langle row_1, sel_1, psw_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psw_2 \rangle$. If $args(head(r'_1))\downarrow_{K'} = args(head(r'_2))\downarrow_{K'}$, then there is an $RR \in \pi_H(p, pr)$ such that $r_1, r_2 \in \bigcup_{R \in RR} R$.*

Proof. Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1, \ldots, |pr|\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, $r'_1 \in ground(p, r_1)$ and $r'_2 \in ground(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in row_1 and row_2, where $\pi_V(r_1) = \langle row_1, sel_1, psw_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psw_2 \rangle$. We assume that $args(head(r'_1))\downarrow_{K'} = args(head(r'_2))\downarrow_{K'}$. Assume, for contradiction's sake, that there is no $RR \in \pi_H(p, pr)$ such that $r_1, r_2 \in \bigcup_{R \in RR} R$. From this and $r_1, r_2 \in p$, it follows that there are $RR_1, RR_2 \in \pi_H(p, pr)$, $R_1 \in RR_1$, and $R_2 \in RR_2$ such that $r_1 \in R_1$, $r_2 \in R_2$, and $RR_1 \neq RR_2$. From this and requirement (4.b) of schema safety, it follows that there is an annotated disjunction $DIS(a, b)$ that can be derived from p such that $a(args(head(r_1))\downarrow_{K'}) \in body^+(r_1)$ and $b(args(head(r_2))\downarrow_{K'}) \in body^+(r_2)$. From this, requirement (4.a) of schema safety, and $args(head(r'_1))\downarrow_{K'} = args(head(r'_2))\downarrow_{K'}$, it follows that $a(args(head(r'_1))\downarrow_{K'}) \in body^+(r'_1)$ and $b(args(head(r'_1))\downarrow_{K'}) \in body^+(r'_2)$. From this and the fact that both literals are positive, it follows that both $a(args(head(r'_1))\downarrow_{K'})$ and $b(args(head(r'_1))\downarrow_{K'})$ are in $ground(p)$. From Proposition B.2 and $DIS(a, b)$ can be derived from p, however, it follows that $a(args(head(r'_1))\downarrow_{K'})$ or $b(args(head(r'_1))\downarrow_{K'})$ are not in $ground(p)$, leading to a contradiction. □

Proposition B.13 states that whenever we have two ground rules with the same head, they are either the same rule or they represent different rows in the CPT.

Proposition B.13. *Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1, \ldots, |pr|\} \setminus K$. For any two distinct rules $r_1, r_2 \in p$ such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, all $r'_1 \in ground(p, r_1)$, and $r'_2 \in ground(p, r_2)$, if $args(head(r'_1))\downarrow_K = args(head(r'_2))\downarrow_K$ and $args(head(r'_1))\downarrow_{K'} = args(head(r'_2))\downarrow_{K'}$, then either $r'_1 = r'_2$ or there are $RR \in \pi_H(p, pr)$, $R_1 \in RR$, and $R_2 \in RR$ such that $r_1 \in R_1$, $r_2 \in R_2$, and $R_1 \neq R_2$.*

Proof. Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$, $K' = \{1, \ldots, |pr|\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, and $r'_1 \in ground(p, r_1)$, and $r'_2 \in ground(p, r_2)$. Assume that $args(head(r'_1))\downarrow_K = args(head(r'_2))\downarrow_K$ and $args(head(r'_1))\downarrow_{K'} = args(head(r'_2))\downarrow_{K'}$. Observe that $\langle \pi_H, \pi_V, \mu \rangle$ is a safe CPT schema. There are two cases:

- $r_1 = r_2$. From this and $args(head(r'_1))\downarrow_K = args(head(r'_2))\downarrow_K$ and $args(head(r'_1))\downarrow_{K'} = args(head(r'_2))\downarrow_{K'}$, it follows that the ground rules have the same heads. From this and the requirement (1) of schema safety, r_1 is strongly connected. From this and Proposition B.6, it follows that $r'_1 = r'_2$.
- $r_1 \neq r_2$. From $r_1 \neq r_2$, $args(head(r'_1))\downarrow_{K'} = args(head(r'_2))\downarrow_{K'}$, and Proposition B.12, it follows that there are $RR \in \pi_H(p, pr)$, $R_1 \in RR$, and $R_2 \in RR$ such that $r_1 \in R_1$ and $r_2 \in R_2$. From this, the requirements (2) and (3.a.i.B) of schema safety, $r_1 \neq r_2$, and $args(head(r'_1))\downarrow_K = args(head(r'_2))\downarrow_K$, it follows that r_1 and r_2 cannot be in the same partition, i.e., $R_1 \neq R_2$.

This completes the proof of our claim. □

Proposition B.14 states that whenever two distinct ground rules represent the same row in the CPT, they produce different values (i.e., the head is different) and they are part of the same set R.

Proposition B.14. *Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1, \ldots, |pr|\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, $r_1' \in ground(p, r_1)$ and $r_2' \in ground(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in row_1 and row_2, where $\pi_V(r_1) = \langle row_1, sel_1, psw_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psw_2 \rangle$. If $gRow_1(i) = gRow_2(i)$ for all $1 \leq i \leq min(|row_1|, |row_2|)$, $args(head(r_1'))\downarrow_{K'} = args(head(r_2'))\downarrow_{K'}$, and $r_1' \neq r_2'$, then $args(head(r_1'))\downarrow_K \neq args(head(r_2'))\downarrow_K$ and there exists an $RR \in \pi_H(p, pr)$ and an $R \in RR$ such that $r_1, r_2 \in R$.*

Proof. Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1, \ldots, |pr|\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, $r_1' \in ground(p, r_1)$ and $r_2' \in ground(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in row_1 and row_2, where $\pi_V(r_1) = \langle row_1, sel_1, psw_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psw_2 \rangle$. Assume that $gRow_1(i) = gRow_2(i)$ for all $1 \leq i \leq min(|row_1|, |row_2|)$, $args(head(r_1'))\downarrow_{K'} = args(head(r_2'))\downarrow_{K'}$, and $r_1' \neq r_2'$. Observe that $\langle \pi_H, \pi_V, \mu \rangle$ is a safe CPT schema. From $args(head(r_1'))\downarrow_{K'} = args(head(r_2'))\downarrow_{K'}$ and Proposition B.12, it follows that there exists an $RR \in \pi_H(p, pr)$, $R_1 \in RR$, and $R_2 \in RR$, such that $r_1 \in R_1$ and $r_2 \in R_2$. From this and $gRow_1(i) = gRow_2(i)$ for all $1 \leq i \leq min(|row_1|, |row_2|)$, it follows that $R_1 = R_2$ (indeed, according to the requirement (3.*b.iii*) of schema safety, rules in different partitions must differ either in the sign of some literals or in the constants used therein). Assume, for contradiction's sake, that $args(head(r_1'))\downarrow_K = args(head(r_2'))\downarrow_K$. From this, the fact that r_1 and r_2 are in the same partition R_1, and the requirement (3.*a.i.A*) of schema safety, it follows that $r_1 = r_2$. From this, $args(head(r_1'))\downarrow_{K'} = args(head(r_2'))\downarrow_{K'}$, and $args(head(r_1'))\downarrow_K = args(head(r_2'))\downarrow_K$, it follows that the heads of the two ground rules are the same. From this and the requirement (1) of schema safety, r_1 is strongly connected. From this and Proposition B.6, it follows that $r_1' = r_2'$, leading to a contradiction. □

Proposition B.15 states that whenever we fix the head and the part of the ground rules representing the row, then we have fully determined the rule.

Proposition B.15. *Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1, \ldots, |pr|\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, $r_1' \in ground(p, r_1)$ and $r_2' \in ground(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in row_1 and row_2, where $\pi_V(r_1) = \langle row_1, sel_1, psw_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psw_2 \rangle$. If $gRow_1(i) = gRow_2(i)$ for all $1 \leq i \leq min(|row_1|, |row_2|)$ and $head(r_1') = head(r_2')$, then $r_1' = r_2'$.*

Proof. Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $\langle \pi_H, \pi_V, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1, \ldots, |pr|\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $pred(head(r_1)) = pr$ and $pred(head(r_2)) = pr$, $r_1' \in ground(p, r_1)$ and $r_2' \in ground(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in row_1 and row_2, where $\pi_V(r_1) = \langle row_1, sel_1, psw_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psw_2 \rangle$. Assume that $gRow_1(i) = gRow_2(i)$ for all $1 \leq i \leq min(|row_1|, |row_2|)$ and $head(r_1') = head(r_2')$. From $head(r_1') = head(r_2')$ and Proposition B.13, it follows that either $r_1' = r_2'$ or there is a $RR \in \pi_H(p, pr)$ such that $R_1 \in RR$, $R_2 \in RR$, $r_1 \in R_1$, $r_2 \in R_2$, and $R_1 \neq R_2$. Assume, for contradiction's sake, that $r_1' \neq r_2'$. Then, it must be the case that there is a $RR \in \pi_H(p, pr)$ such that $R_1 \in RR$, $R_2 \in RR$, $r_1 \in R_1$, $r_2 \in R_2$, and $R_1 \neq R_2$. From this and requirement (3.*b.iii*), there is an $1 \leq i \leq min(|row_1|, |row_2|)$ such that r_1 and r_2 differ in at least a literal. This, however, contradicts $gRow_1(i) = gRow_2(i)$ for all $1 \leq i \leq min(|row_1|, |row_2|)$. Hence, $r_1' = r_2'$. □

B.3.4 Auxiliary results about Relaxed Acyclic Programs

Proposition B.16 states a simple fact about relaxed acyclic programs.

Proposition B.16. *Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $W = \langle \pi_V, \pi_H, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ. If $\mu(pr) \neq \emptyset$, then each ground atom $pr(\overline{c})$ can be derived only from one rule in $\alpha(\beta_W(p))$.*

Proof. Let p be a $(\Sigma, \mathbf{dom})$-ProbLog program, $W = \langle \pi_V, \pi_H, \mu \rangle$ be an acyclicity witness for p, and pr be a predicate symbol in Σ. Furthermore, assume that $\mu(pr) \neq \emptyset$. Assume for contradiction's sake, that there are two distinct rules r and r' from which we can derive $pr(\overline{c})$. There are two cases:

- r and r' are derived from rules r_1 and r_2 such that there are two distinct $RR_1, RR_2 \in \pi_H(p, pr)$, $r_1 \in \bigcup_{R \in RR_1} R$, and $r_2 \in \bigcup_{R \in RR_2} R$. From this and requirement (4.b) of schema safety, it follows that there is a disjointness annotation $DIS(a, b)$ (that can be derived from p) and indexes i, j such that $a(args(head(r))\downarrow_{\mu(pr)})$ and $b(args(head(r'))\downarrow_{\mu(pr)})$ are positive literals in r_1 and r_2. From requirement (4.a) of schema safety and the fact that $head(r) = head(r')$, it follows that $args(head(r))\downarrow_{\mu(pr)} = args(head(r)')\downarrow_{\mu(pr)}$. Furthermore, from requirements (4.b), (4.c), and the transformation's definition, it follows that $a(args(head(r))\downarrow_{\mu(pr)})$ and $b(args(head(r))\downarrow_{\mu(pr)})$ are positive literals in r and r'. From this and $args(head(r))\downarrow_{\mu(pr)}) = args(head(r')\downarrow_{\mu(pr)}$, it follows that $a(args(head(r))\downarrow_{\mu(pr)})$ and $b(args(head(r))\downarrow_{\mu(pr)})$ belong to $ground(p)$. This, however, contradicts $a(args(head(r))\downarrow_{\mu(pr)})$ or $b(args(head(r))\downarrow_{\mu(pr)})$ are not in $ground(p)$, which follows from $DIS(a, b)$ and Proposition B.2.
- r and r' are derived from rules r_1 and r_1' such that there is $RR \in \pi_H(p, pr)$, $r_1' \in \bigcup_{R \in RR} R$, and $r_2' \in \bigcup_{R \in RR} R$. From this and requirement (3.c), r and r' are the same rule, leading to a contradiction.

This completes the proof of our claim. □

Proposition B.17 states a simple results connecting the original program p and its transformed version $\alpha(\beta_W(p))$.

Proposition B.17. *Let* **dom** *be a finite domain,* $\langle \Sigma, \mathbf{dom} \rangle$ *be a database schema,* p *be a* $(\Sigma, \mathbf{dom})$*-relaxed acyclic* ProbLog *program,* W *be a witness for* p*, and* p' *be the program* $\alpha(\beta_W(p))$*. Then, (1)* $a \in ground(p)$ *implies* $a\downarrow_p \in ground(p')$*, and (2)* $s \in ground(p, r)$ *implies* $s\downarrow_p \in ground(p', r\downarrow_p)$*.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-relaxed acyclic ProbLog program, W be a witness for p, and p' be the program $\alpha(\beta_W(p))$. We now prove our first claim. Let $a \in ground(p)$. From this, it follows that there is an i such that $a \in ground(p, i)$. We now prove our claim by induction on i. The base case is $a \in ground(p, 0)$. From this, it follows that a is either a ground atom or a probabilistic ground atom. From this and the definition of α and β, $a\downarrow_p = a$. From this, it follows that $a \in p'$ and, therefore, $a \in ground(p')$. For the induction step, assume that our claim holds for all $j < i$, we now show that it holds also for i. The only interesting case is $a \in ground(p, i) \setminus ground(p, i-1)$. From this, it follows that there is a rule r and a ground rule s such that all $body^+(s) \subseteq ground(p, i-1)$. From this, $r\downarrow_p \in p'$, the fact that the transformation does not introduce new positive literals, and the induction hypothesis, it follows that $\{b\downarrow_p \mid b \in body^+(s)\} \subseteq ground(p')$. From this and $ground$'s definition, it follows that $a\downarrow_p \in ground(p')$. The proof of our second claim is similar to the first one. □

B.3.5 Auxiliary Lemmas

We are now ready to prove some auxiliary lemmas for the encoding's correctness.

Lemma B.4. *Let* $\langle \Sigma, \mathbf{dom} \rangle$ *be a database schema such that* **dom** *is a finite domain,* p_0 *be a* $(\Sigma,$ **dom**$)$*-relaxed acyclic* ProbLog *program,* W *be a witness for* p_0*,* p *be the transformed program* $\alpha(\beta_W(p))$*,* $BN = \langle N, E, CPT \rangle$ *be the Bayesian Network generated by Algorithm 2 having* p_0 *and* W *as input. Furthermore, let* r *be a rule in* p *and* D *be the function associating to each predicate symbol* pr *its domain (see Algorithm 2 for* D*'s definition). Finally, let* $\langle v_1, \ldots, v_{n+1} \rangle$ *a tuple in* $D(a_1) \times \ldots \times D(a_n) \times D(pred(head(r)))$*, where* $n = |body(r)|$ *and* $a_i = pred(body(r, i))$ *for* $1 \leq i \leq |body(r)|$*. The following statements hold:*

1. *If* $v_{n+1} \neq \bot$*, then* $satisfiable(r, \langle v_1, \ldots, v_{n+1} \rangle, D, W, p) = \top$ *iff there exists a rule* $r' \in [r]_{p_0, W}$ *such that* r' *matches* $\langle v_1, \ldots, v_n, v_{n+1} \rangle$*.*
2. $satisfiable(r, \langle v_1, \ldots, \bot \rangle, D, W, p) = \bot$ *iff there are a* $v_{n+1} \neq \bot$ *and a rule* $r' \in [r]_{p_0, W}$ *that match* $\langle v_1, \ldots, v_n, v_{n+1} \rangle$*.*
3. *If* $v_{n+1} \neq \bot$ *and* $v_{n+1} \neq \top$*, then there is at most one rule* $r' \in [r]_{p_0, W}$ *such that* r' *matches* $\langle v_1, \ldots, v_n, v_{n+1} \rangle$*.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p_0 be a $(\Sigma, \mathbf{dom})$-relaxed acyclic ProbLog program, W be a witness for p_0, p be the transformed program $\alpha(\beta_W(p))$, $BN = \langle N, E, CPT \rangle$ be the Bayesian Network generated by Algorithm 2 having p_0 and W as input. Furthermore, let r be a rule in p and D be the function associating to each predicate symbol pr its domain (see Algorithm 2 for D's definition). Finally, let $\langle v_1, \ldots, v_{n+1} \rangle$ a tuple in $D(a_1) \times \ldots \times D(a_n) \times D(pred(head(r)))$, where $n = |body(r)|$ and $a_i = pred(body(r, i))$ for $1 \leq i \leq |body(r)|$.

Statement 1. Assume that $v_{n+1} \neq \bot$. Then, the *satisfiable* procedure returns $\top$ iff there exists a rule $r' \in [r]_{p_0, W}$ such that $filter(head(r'), D, W) = v_{n+1}$ and for all $1 \leq i \leq |body(r')|$, $v_i \in filter(body(r', i), D, W)$. Observe that (1) $filter(head(r'), D, W) = v_{n+1}$ iff $h\uparrow_p = v_{n+1}$, (2) for any positive literal,

$v_i \in \mathit{filter}(\mathit{body}(r',i),D,W)$ iff $v_i \in \{\mathit{body}(r',i)\uparrow_p\}$, and (3) for any negative literal, $v_i \in \mathit{filter}(\mathit{body}(r',i),D,W)$ iff $v_i \in (D(\mathit{pred}(\mathit{body}(r',i)))\setminus\{\mathit{body}(r',i)\uparrow_p\})\cup\{\bot\}$. Therefore, *satisfiable* procedure returns $\top$ iff there exists a matching rule in $[r]_{p_0,W}$.

Statement 2. Assume that $v_{n+1}=\bot$. By inspecting the code of *satisfiable* it is easy to see that in case $v_{n+1}=\bot$ the function is equivalent to $\neg(\bigvee_{v_{n+1}\in D(\mathit{pred}(\mathit{head}(r)))\setminus\{\bot\}} \mathit{satisfiable}(r,\langle v_1,\ldots,\bot\rangle,D,W,p))$. From this, $\mathit{satisfiable}(r,\langle v_1,\ldots,\bot\rangle,D,W,p)=\top$ iff $\bigwedge_{v_{n+1}\in D(\mathit{pred}(\mathit{head}(r)))\setminus\{\bot\}} \neg\mathit{satisfiable}(r,\langle v_1,\ldots,v_{n+1}\rangle,D,W,p)$ holds. From this, $\mathit{satisfiable}(r,\langle v_1,\ldots,\bot\rangle,D,W,p)=\bot$ iff there exists a $v_{n+1}\neq\bot$ such that $\mathit{satisfiable}(r,\langle v_1,\ldots,v_{n+1}\rangle,D,W,p)$ holds. From this (and the proof of the previous claim), $\mathit{satisfiable}(r,\langle v_1,\ldots,\bot\rangle,D,W,p)=\bot$ iff there exists a $v_{n+1}\neq\bot$ and a rule $r'\in[r]_{p_0,W}$ that matches $\langle v_1,\ldots,v_n,v_{n+1}\rangle$.

Statement 3. Assume that $v_{n+1}\neq\bot$ and $v_{n+1}\neq\top$. Assume, for contradiction's sake, that there are two distinct rules $r',r''\in[r]_{p_0,W}$ such that $\overline{v}=\langle v_1,\ldots,v_n,v_{n+1}\rangle$ matches r' and r''. Let pr be the symbol $\mathit{pred}(\mathit{head}(r'))$ (note that $\mathit{pred}(\mathit{head}(r'))=\mathit{pred}(\mathit{head}(r''))$). From r' matches $\overline{v}$, it follows that (1) $\mathit{head}(r')\uparrow_p=v_{n+1}$, (2) for all $1\leq i\leq|\mathit{body}(r')|$, if $l_i\in\mathit{body}^+(r')$, then $\overline{v_i}=l_i\uparrow_p$, and (3) for all $1\leq i\leq|\mathit{body}(r')|$, if $l_i\in\mathit{body}^-(r')$, then $\overline{v_i}=\bot$ or $\overline{v_i}\neq l_i\uparrow_p$. Similarly, from r'' matches $\overline{v}$, it follows that (1) $\mathit{head}(r'')\uparrow_p=v_{n+1}$, (2) for all $1\leq i\leq|\mathit{body}(r'')|$, if $l_i\in\mathit{body}^+(r'')$, then $\overline{v_i}=l_i\uparrow_p$, and (3) for all $1\leq i\leq|\mathit{body}(r'')|$, if $l_i\in\mathit{body}^-(r'')$, then $\overline{v_i}=\bot$ or $\overline{v_i}\neq l_i\uparrow_p$. From this and $v_{n+1}\notin\{\bot,\top\}$, it follows that $\mathit{args}(\mathit{head}(r'))\downarrow_{\mu(pr)}=v_{n+1}$ and $\mathit{args}(\mathit{head}(r''))\downarrow_{\mu(pr)}=v_{n+1}$. From $r',r''\in[r]_{p_0,W}$ and requirement (3.*c*) of schema safety, it follows that there are $RR\in\pi_H(p,pr)$, $R_1\in RR$, and $R_2\in RR$ such that $r'\in R_1$ and $r''\in R_2$ (otherwise, r and r' would result in different rules in the transformed program). There are two cases:

- $R_1=R_2$. From this and requirement (3.*a.i.B*) of schema safety, $\mathit{args}(\mathit{head}(r'))\downarrow_{\mu(pr)}\neq\mathit{args}(\mathit{head}(r''))\downarrow_{\mu(pr)}$. This contradicts $\mathit{args}(\mathit{head}(r'))\downarrow_{\mu(pr)}=v_{n+1}$ and $\mathit{args}(\mathit{head}(r''))\downarrow_{\mu(pr)}=v_{n+1}$.
- $R_1\neq R_2$. From this and requirement (3.*b.iii*) of schema safety, there exists an $1\leq i\leq\mathit{min}(|\mathit{row}_1|,|\mathit{row}_2|)$, where $\pi_V(r')=\langle\mathit{row}_1,\mathit{sel}_1,\mathit{sw}_1\rangle$ and $\pi_V(r'')=\langle\mathit{row}_2,\mathit{sel}_2,\mathit{sw}_2\rangle$, such that:
  1. $\mu(\mathit{pred}(\mathit{row}_1)(i))\neq\emptyset$, $\mathit{row}_1(i)$ and $\mathit{row}_2(i)$ are positive literals, and $\mathit{args}(\mathit{row}_1(i))\downarrow_{K_{1,i}}\neq\mathit{args}(\mathit{row}_2(i))\downarrow_{K_{2,i}}$, where $K_{1,i}=\mu(\mathit{pred}(\mathit{row}_1(i)))$ and $K_{2,i}=\mu(\mathit{pred}(\mathit{row}_2(i)))$, or
  2. $\mu(\mathit{pred}(\mathit{row}_1)(i))=\emptyset$, $\mathit{row}_1(i)$ is a positive literal, and $\mathit{row}_2(i)$ is a negative one or vice versa.

  In the first case, there is an $1\leq i\leq\mathit{min}(|\mathit{body}(r')|,|\mathit{body}(r'')|)$ such that $\mathit{body}(r',i)$ and $\mathit{body}(r'',i)$ are positive literals and $\mathit{body}(r',i)\uparrow_p\neq\mathit{body}(r'',i)\uparrow_p$. This, however, contradicts that $v_i=\mathit{body}(r',i)\uparrow_p$ and $v_i=\mathit{body}(r'',i)\uparrow_p$. In the second case, there is an $1\leq i\leq\mathit{min}(|\mathit{body}(r')|,|\mathit{body}(r'')|)$ such that $\mathit{body}(r',i)$ is a positive literal and $\mathit{body}(r'',i)$ is a negative one (or vice versa). Hence, $\mathit{body}(r',i)\uparrow_p=\top$ and $\mathit{body}(r',i)\uparrow_p=\bot$ (or vice versa). This, again, contradicts $v_i=\mathit{body}(r',i)\uparrow_p$ and $v_i=\mathit{body}(r'',i)\uparrow_p$.

This completes the proof of our third claim. □

Lemma B.5. *Let $\langle\Sigma,\mathbf{dom}\rangle$ be a database schema such that* **dom** *is a finite domain, p be a $(\Sigma,\mathbf{dom})$-relaxed acyclic* ProbLog *program, W be a witness for p, $BN=\langle N,E,CPT\rangle$ be the Bayesian Network generated by Algorithm 2 having p and W as input, f be a p-probabilistic assignment, and ν be a consistent BN-variable assignment such that $\nu(X[v{::}a(\overline{c}),\emptyset,a(\overline{c})])=f(v{::}a(\overline{c}))$ for all $v{::}a(\overline{c})\in p$. Then, $a(\overline{c})\in g_f(p,j,i)$, for some j and i, iff $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$.*

Proof. Let $\langle\Sigma,\mathbf{dom}\rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma,\mathbf{dom})$-relaxed acyclic ProbLog program, W be a witness for p, $BN=\langle N,E,CPT\rangle$ be the Bayesian Network generated by Algorithm 2 having p and W as input, f be a p-probabilistic assignment, and ν be a consistent BN-variable assignment such that $\nu(X[v{::}a(\overline{c}),\emptyset,a(\overline{c})])=f(v{::}a(\overline{c}))$ for all $v{::}a(\overline{c})\in p$. Furthermore, let p' be the acyclic ProbLog program obtained after applying the α and β transformations. We claim that $a(\overline{c})\in g_f(p,\mu(a))$ iff $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$. From this, $a(\overline{c})\in g_f(p)$ iff $a(\overline{c})\in g_f(p,j,i)$ for some j and i, and $a(\overline{c})\in g_f(p)$ iff $a(\overline{c})\in g_f(p,\mu(a))$, it follows that $a(\overline{c})\in g_f(p,j,i)$, for some j and i, iff $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$. However, there may be some nodes in BN that do not correspond to any ground rule or atom in $g_f(p,r)$ or $g_f(p)$.

We now prove, by induction on $\mu(a)$ (defined in Section B.3.2), our claim that $a(\overline{c})\in g_f(p,\mu(a))$ iff $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$ (we denote the corresponding induction hypothesis as $(\star)$).

Base Case. For the base case, we assume that $\mu(a)=0$. We prove separately the two directions, namely (1) if $a(\overline{c})\in g_f(p,0,i)$ and $\mu(a)=0$, then $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$, and (2) if $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$ and $\mu(a)=0$, then $a(\overline{c})\in g_f(p,0,i)$. From this, it follows that if $\mu(a)=0$, then $a(\overline{c})\in g_f(p,0)$ iff $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$.

(⇒). We prove, by induction on i, that if $a(\overline{c})\in g_f(p,0,i)$, then $\nu(X[a(\overline{c})\downarrow_p])=a(\overline{c})\uparrow_p$ (we denote this induction hypothesis as (†)). From $a(\overline{c})\in g_f(p)$, $g_f(p)\subseteq\mathit{ground}(p)$, and Proposition B.17, then

$X[a(\overline{c})\downarrow_p] \in N$. The base case is as follows. If $a(\overline{c}) \in g_f(p,0,0)$, then there is a rule r such that either $r = a(\overline{c})$ or $r = v{::}a(\overline{c})$ and $a(\overline{c}) \in g_f(p,r,0,0)$. If $r = a(\overline{c})$, then $\nu(X[r,\emptyset,a(\overline{c})]) = \top$ due to ν's consistency, $g_f(p) \subseteq \mathit{ground}(p)$, Proposition B.17, $r\downarrow_p = r$, and $a(\overline{c})\downarrow_p = a(\overline{c})$. If $r = v{::}a(\overline{c})$, then $f(v{::}a(\overline{c})) = \top$ follows from $a(\overline{c}) \in g_f(p,0,0)$. From this, $g_f(p) \subseteq \mathit{ground}(p)$, Proposition B.17, $r\downarrow_p = r$, $a(\overline{c})\downarrow_p = a(\overline{c})$, and $\nu(X[v{::}a(\overline{c}),\emptyset,a(\overline{c})]) = f(v{::}a(\overline{c}))$, it follows that there is a variable $X[r,\emptyset,a(\overline{c})] \in N$ such that $\nu(X[r,\emptyset,a(\overline{c})]) = \top$. From $\nu(X[r,\emptyset,a(\overline{c})]) = \top$, BN's construction, and ν's definition, it follows that there are nodes $X[r,a(\overline{c})], X[a(\overline{c})] \in N$ such that $\nu(X[r,a(\overline{c})]) = \top$ and $\nu(X[a(\overline{c})]) = \top$ as well (note that $r\downarrow_p = r$, and $a(\overline{c})\downarrow_p = a(\overline{c})$ in this case). For the induction step, we assume that our claim holds for all $i' < i$. The only interesting case is when $a(\overline{c}) \in g_f(p,0,i) \setminus g_f(p,0,i-1)$. From this, it follows that there is rule $r \in p$ such that $a(\overline{c}) \leftarrow b_1,\ldots,b_m \in g_f(p,r,0,i)$. From this, it follows that $b_1,\ldots,b_m \in g_f(p,0,i-1)$ and $\mathit{body}^-(r) = \emptyset$. From this, $g_f(p,0,i-1) \subseteq \mathit{ground}(p)$, Proposition B.17, and the induction's hypothesis (†), it follows that there are nodes $X[b_1\downarrow_p],\ldots,X[b_m\downarrow_p] \in N$ and $\nu(X[b_1\downarrow_p]) = b_1\uparrow_p,\ldots,\nu(X[b_m\downarrow_p]) = b_m\uparrow_p$. From this, $\mathit{body}^-(r) = \emptyset$, $g_f(p,r,0,i) \subseteq g_f(p,r) \subseteq \mathit{ground}(p,r)$, Proposition B.17, and BN's construction, it follows that there is a variable $X[r\downarrow_p,\{b_1\downarrow_p,\ldots,b_m\downarrow_p\},a(\overline{c})\downarrow_p] \in N$ such that $\mathit{cpt}(X[r\downarrow_p,\{b_1\downarrow_p,\ldots,b_m\downarrow_p\},a(\overline{c})\downarrow_p])(\nu(X[b_1\downarrow_p]),\ldots,\nu(X[b_m\downarrow_p]),k) = 1$ iff $k = a(\overline{c})\uparrow_p$ (this follows from the definitions of *cpt*, *satisfiable*, and Lemma B.4). As a result, $\nu(X[r\downarrow_p,\{b_1\downarrow_p,\ldots,b_m\downarrow_p\},a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$ since ν is consistent. From this and BN's construction, it follows that there are nodes $X[r\downarrow_p,a(\overline{c})\downarrow_p]$, $X[a(\overline{c})\downarrow_p] \in N$ such that $\nu(X[r\downarrow_p,a(\overline{c})\downarrow_p]) = \top$ and $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$. This completes the proof of the if direction.

(⇐). To prove that if $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$, then $a(\overline{c}) \in g_f(p,0,i)$, for some i, we prove a stronger claim. Namely, if $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$, then $a(\overline{c}) \in g_f(p,0,\mathit{depth}(X[a(\overline{c})\downarrow_p]))$, where $\mathit{depth}(n) = 0$ if $\mathit{ancestors}(n)$ does not contain any variable of the form $X[b(\overline{v})]$ and $\mathit{depth}(n) = 1 + \mathit{max}_{n' \in \mathit{ancestors}(n)}\mathit{depth}(n')$ otherwise. We prove our claim by induction on $\mathit{depth}(X[a(\overline{c})\downarrow_p])$ (we denote the induction hypothesis as (△)). The base case is as follows. If $\mathit{depth}(X[a(\overline{c})\downarrow_p]) = 0$, then from $\nu(X[a(\overline{c})\downarrow_p]) = \top$ and ν's consistency, it follows that there must be a rule r, of the form $r = v{::}a(\overline{c})$ or $r = a(\overline{c})$, such that $X[r,a(\overline{c})] \in N$ and $\nu(X[r,a(\overline{c})]) = \top$ (note that, in this case, $r\downarrow_p = r$ and $a(\overline{c})\downarrow_p = a(\overline{c})$). From this and ν's consistency, there is a variable $X[r,\emptyset,a(\overline{c})] \in N$ such that $\nu(X[r,\emptyset,a(\overline{c})]) = \top$. If $r = a(\overline{c})$, then $a(\overline{c}) \in g_f(p,0,0)$ by definition. If $r = v{::}a(\overline{c})$, then from $\nu(X[r,\emptyset,a(\overline{c})]) = \top$ and $\nu(X[r,\emptyset,a(\overline{c})]) = f(v{::}a(\overline{c}))$, it follows that $f(v{::}a(\overline{c})) = \top$. From this, it follows that $a(\overline{c}) \in g_f(p,0,0)$. For the induction step, we assume that our claim holds for all random variables of depth less than k. From $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$ and ν's consistency, it follows that there is a rule $r \in p'$ such that $\nu(X[r,a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$. From this and ν's consistency, there is a set of positive ground literals $I = (b_1,\ldots,b_m)$ such that $X[r,(b_1\downarrow_p,\ldots,b_m\downarrow_p),a(\overline{c})\downarrow_p] \in N$ and $\nu(X[r,(b_1\downarrow_p,\ldots,b_m\downarrow_p),a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$. From this and BN's construction (cf. the *cpt* and *satisfiable* procedures and Lemma B.4), it follows that there is a rule $r' \in p$ such that (1) $r' \in [r]_p$, and (2) the values assigned by ν to the random variables associated to the atoms in I produce a grounding s' of r' that satisfies all constraints and is consistent (namely, the body of s' does not contain both an atom and its negation and multiple copies of the same CPT-like atom in r' are assigned to the same value in r). From this, BN's definition, and $\mu(a) = 0$, it follows that $\nu(X[b\downarrow_p]) = b\uparrow_p$ for all $b \in I$. From this and the induction's hypothesis (△), it follows that $b \in g_f(p,0,\mathit{depth}(X[a(\overline{c})]) - 1)$ for all $b \in I$. From this, there is a rule $r' \in p$ such that (1) $r' \in [r]_p$, and (2) r is satisfied by $\{b_1,\ldots,b_m\}$, i.e., there is a grounding s' of r' obtained by using a subset of the literals in $\{b_1,\ldots,b_m\}$ and adding repeated occurrences of the literals if needed. From this, it follows that s is in $g_f(p,r,0,\mathit{depth}(X[a(\overline{c})\downarrow_p]))$. From this, it follows that $a(\overline{c}) \in g_f(p,0,\mathit{depth}(X[a(\overline{c})\downarrow_p]))$. This completes the proof the only if direction.

Induction Step. For the induction's step, we assume that $b(\overline{d}) \in g_f(p,\mu(b))$ iff $\nu(X[b(\overline{d})\downarrow_p]) = b(\overline{d})\uparrow_p$ holds for all b such that $\mu(b) < \mu(a)$. We now prove that $a(\overline{c}) \in g_f(p,\mu(a))$ iff $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$ as well. In the following, let k be $\mu(a)$. We prove separately the two directions, namely (1) if $a(\overline{c}) \in g_f(p,\mu(a),i)$, for some i, then $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$, and (2) if $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$, then $a(\overline{c}) \in g_f(p,\mu(a),i)$, for some i. From this, it follows that $a(\overline{c}) \in g_f(p,\mu(a))$ iff $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$.

(⇒). We prove, by induction on i, that if $a(\overline{c}) \in g_f(p,k,i)$, for some i, then $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$ (we denote the induction hypothesis associated to this proof as (♣)). From $a(\overline{c}) \in g_f(p) \subseteq \mathit{ground}(p)$ and Proposition B.17, then $X[a(\overline{c})\downarrow_p] \in N$. The base case is as follows. Assume that $a(\overline{c}) \in g_f(p,k,0)$. From this and $a(\overline{c}) \notin g_f(p,k-1)$ (since $\mu(a) > k-1$), it follows that there is a rule r such that either $r = a(\overline{c})$ or $r = v{::}a(\overline{c})$. If $r = a(\overline{c})$, then there is a node $X[r,\emptyset,a(\overline{c})] \in N$ such that $\nu(X[r,\emptyset,a(\overline{c})]) = \top$ due to ν's consistency, $g_f(p,r) \subseteq \mathit{ground}(p,r)$, $r\downarrow_p = r$, $a(\overline{c})\downarrow_p = a(\overline{c})$, and Proposition B.17. If $r = v{::}a(\overline{c})$, then $f(v{::}a(\overline{c})) = \top$ follows from $a(\overline{c}) \in g_f(p,k,0)$. From this, $g_f(p,r) \subseteq \mathit{ground}(p,r)$, Proposition B.17, and $\nu(X[\{v{::}a(\overline{c})\},\emptyset,a(\overline{c})]) = f(v{::}a(\overline{c}))$, it follows that there is a node $X[r,\emptyset,a(\overline{c})] \in N$ such that $\nu(X[r,\emptyset,a(\overline{c})]) = \top$. From $\nu(X[r,\emptyset,a(\overline{c})]) = \top$, BN's definition, and *nu*'s definition, it follows that there are nodes $X[r,a(\overline{c})], X[a(\overline{c})] \in N$ such that $\nu(X[r,a(\overline{c})]) = \top$ and $\nu(X[a(\overline{c})]) = \top$ as well (note that $r\downarrow_p = r$ and $a(\overline{c})\downarrow_p = a(\overline{c})$). For

the induction step, we assume that our claim holds for all $i' < i$. Assume that $a(\overline{c}) \in g_f(p,k,i)$. The only interesting case is when $a(\overline{c}) \in g_f(p,k,i) \setminus g_f(p,k,i-1)$. From this, it follows that there is rule r such that $r' = a(\overline{c}) \leftarrow b_1,\ldots,b_m$ and $r' \in g_f(p,r,k,i)$. Furthermore, we denote by $b'_1,\ldots,b'_m$ the atoms $pos(b_1),\ldots,pos(b_m)$. From this, it follows that $b_e \in g_f(p,k,i-1)$ for all $b_e \in body^+(r')$ and $b'_d \notin g_f(p,k-1)$ for all $b_d \in body^-(r')$. From $b_e \in g_f(p,k,i-1)$ for all $b_e \in body^+(r')$, $g_f(p,r) \subseteq ground(p,r)$, Proposition B.17, and the induction's hypothesis (♣), it follows that there is a node $X[b_e{\downarrow_p}] \in N$ such that $\nu(X[b_e{\downarrow_p}]) = b_e{\uparrow_p}$ for all $b_e \in body^+(r')$. From $b'_d \notin g_f(p,k-1)$ for all $b_d \in body^-(r')$, it follows that $b'_d \notin g_f(p,\mu(pred(b_d)))$ for all $b_d \in body^-(r')$. From this, $r' \in g_f(p,r,k,i)$, $g_f(p,r,k,i) \subseteq ground(p,r)$, Proposition B.17, $\mu(pred(b_d)) < \mu(a)$ for all $b_d \in body^-(r')$, and the induction's hypothesis (⋆), it follows that there is a node $X[b'_d{\downarrow_p}] \in N$ such that $\nu(X[b'_d{\downarrow_p}]) \neq b'_d{\uparrow_p}$ for all $b_d \in body^-(r')$. From $r' \in g_f(p,r,k,i)$, $g_f(p,r,k,i) \subseteq ground(p,r)$, Proposition B.17, $\nu(X[b_e{\downarrow_p}]) = b_e{\uparrow_p}$ for all $b_e \in body^+(r')$, $\nu(X[b'_d{\downarrow_p}]) \neq b'_d{\uparrow_p}$ for all $b_d \in body^-(r')$, and *BN*'s construction (cf. the *cpt* and *satisfiable* procedures and Lemma B.4), there is $X[r{\downarrow_p},\{b'_1{\downarrow_p},\ldots,b'_m{\downarrow_p}\},a(\overline{c}){\downarrow_p}] \in N$ such that $cpt(X[r{\downarrow_p},\{b'_1{\downarrow_p},\ldots,b'_m{\downarrow_p}\},a(\overline{c}){\downarrow_p}])(\nu(X[b'_1{\downarrow_p}]),\ldots,\nu(X[b'_m{\downarrow_p}]),k) = 1$ iff $k = a(\overline{c}){\uparrow_p}$. As a result, $\nu(X[r{\downarrow_p},\{b'_1{\downarrow_p},\ldots,b'_m{\downarrow_p}\},a(\overline{c}){\downarrow_p}]) = \top$ since ν is consistent. From this and *BN*'s construction, it follows that there are variables $X[r{\downarrow_p},a(\overline{c}){\downarrow_p}], X[a(\overline{c}){\downarrow_p}] \in N$ such that $\nu(X[r{\downarrow_p},a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$ and $\nu(X[a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$. This completes the proof of the if direction.

(⇐). To prove that if $\nu(X[a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$, then $a(\overline{c}) \in g_f(p,k,i)$, for some i, we prove a stronger claim. Namely, if $\nu(X[a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$, then $a(\overline{c}) \in g_f(p,k,depth(X[a(\overline{c})]))$, where $depth(n)$ is as above. We prove our claim by induction on $depth(X[a(\overline{c}){\downarrow_p}])$ (we denote the induction hypothesis associated to this proof as (♠)). The base case is as follows. If $depth(X[a(\overline{c}){\downarrow_p}]) = 0$, then from $\nu(X[a(\overline{c}){\downarrow_p}]) = \top$ and ν's consistency, it follows that there must be a rule r, of the form $r = v{::}a(\overline{c})$ or $r = a(\overline{c})$, such that $X[r,a(\overline{c})] \in N$ and $\nu(X[r,a(\overline{c})]) = \top$ (note that $r{\downarrow_p} = r$ and $a(\overline{c}){\downarrow_p} = a(\overline{c})$). From this and ν's consistency, there is a variable $X[r,\emptyset,a(\overline{c})] \in N$ such that $\nu(X[r,\emptyset,a(\overline{c})]) = \top$. If $r = a(\overline{c})$, then $a(\overline{c}) \in g_f(p,k,0)$ by definition. If $r = v{::}a(\overline{c})$, then from $\nu(X[r,\emptyset,a(\overline{c})]) = \top$ and $\nu(X[r,\emptyset,a(\overline{c})]) = f(v{::}a(\overline{c}))$, it follows that $f(v{::}a(\overline{c})) = \top$. From this, it follows that $a(\overline{c}) \in g_f(p,k,0)$. For the induction step, we assume that our claim holds for all random variables of depth less than k. We now show that it holds also for a variable $X[a(\overline{c}){\downarrow_p}]$ such that $depth(X[a(\overline{c}){\downarrow_p}]) = i$. From $\nu(X[a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$ and ν's consistency, it follows that there is a rule $r' \in p'$ such that $X[r',a(\overline{c}){\downarrow_p}] \in N$ and $\nu(X[r',a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$. From this and ν's consistency, there is a set of ground atoms $I = (b_1,\ldots,b_m)$ such that $X[r',(b_1{\downarrow_p},\ldots,b_m{\downarrow_p}),a(\overline{c}){\downarrow_p}] \in N$, $\nu(X[r',(b_1{\downarrow_p},\ldots,b_m{\downarrow_p}),a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$, and $X[b{\downarrow_p}] \in N$ for all $b \in I$. From this and *BN*'s construction (cf. the *cpt* and *satisfiable* procedures and Lemma B.4), it follows that there is a rule $r \in p$ such that (1) $r \in [r']_p$, and (2) the values assigned by ν to the random variables associated to the atoms in I produce a grounding s of r that satisfies all constraints and is consistent (namely, the body of s does not contain both an atom and its negation and multiple copies of the same CPT-like atom in r' are assigned to the same value in r). Let I^+ be the atoms in I that are assigned to positive literals in s and I^- be the atoms in I that are assigned to negative literals (note that these two sets are disjoint). From $\nu(X[r',(b_1{\downarrow_p},\ldots,b_m{\downarrow_p}),a(\overline{c}){\downarrow_p}]) = a(\overline{c}){\uparrow_p}$, and ν's consistency, it follows that $\nu(X[b_e{\downarrow_p}]) = b_e{\uparrow_p}$ for all $b_e \in I^+$ and $\nu(X[b_d{\downarrow_p}]) \neq b_d{\uparrow_p}$ for all $b_d \in I^-$. From $\nu(X[b_e{\downarrow_p}]) = b_e{\uparrow_p}$ and $depth(X[b_e{\downarrow_p}]) < depth(X[a(\overline{c}){\downarrow_p}])$ for all $b_e \in I^+$ and the induction's hypothesis (♠), it follows that $b_e \in g_f(p,k,depth(X[b_e{\downarrow_p}]))$ for all $b_e \in I^+$. From this, the induction's hypothesis, and $depth(X[b_e{\downarrow_p}]) < depth(X[a(\overline{c}){\downarrow_p}])$, it follows that $b_e \in g_f(p,k,i-1)$ for all $b_e \in body^+(r')$. From $\nu(X[b_d{\downarrow_p}]) \neq b_d{\uparrow_p}$ and $\mu(pred(b_d)) < \mu(a)$ for all $b_d \in I^-$ and the induction's hypothesis (⋆), it follows that $b_d \notin g_f(p,\mu(pred(b_d)))$ for all $b_d \in I^-$. From this and $\mu(pred(b_d)) < \mu(a)$, it follows that $b_d \notin g_f(p,k-1)$ for all $b_d \in I^-$. From r' definition, $b_e \in g_f(p,k,i-1)$ for all $b_e \in I^+$, and $b_d \notin g_f(p,k-1)$ for all $b_d \in I^-$, it follows that $r' \in g_f(p,r,k,i)$. From this, it follows that $r' \in g_f(p,k,i)$ and $a(\overline{v}) \in g_f(p,k,i)$. This completes the proof the only if direction. □

Lemma B.6. *Let $\langle\Sigma,\mathbf{dom}\rangle$ be a database schema such that* **dom** *is a finite domain, p be a $(\Sigma,$* **dom***$)$-relaxed acyclic* ProbLog *program, W be a witness for p, s be a $(\Sigma,\mathbf{dom})$-structure, and $BN = \langle N,E,CPT\rangle$ be the Bayesian Network generated by Algorithm 2 having p and W as input. There is a p-probabilistic assignment f such that $prob(f) = k$ and $g_f(p) = s$ iff there is a BN-total assignment ν such that $[\![BN]\!](\nu) = k$, ν is consistent, and $\nu \models s$.*

Proof. Let $\langle\Sigma,\mathbf{dom}\rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma,\mathbf{dom})$-relaxed acyclic ProbLog program, W be a witness for p, s be a $(\Sigma,\mathbf{dom})$-structure, and $BN = \langle N,E,CPT\rangle$ be the Bayesian Network generated by Algorithm 2 having p and W as input. We now prove both directions of our claim.

(⇒). Let f be a p-probabilistic assignment f such that $prob(f) = k$ and $g_f(p) = s$. Furthermore, let ν be the following BN-total assignment:

$$\nu(n) = \begin{cases} f(v{::}a(\overline{c})) & \text{if } n = X[v{::}a(\overline{c}), \emptyset, a(\overline{c})] \\ \top & \text{if } n = X[a(\overline{c}), \emptyset, a(\overline{c})] \\ k & \text{if } k \in D(n) \wedge p(n) = \{P_1, \ldots, P_m\} \\ & \quad \wedge \nu(P_1) = v_1 \wedge \ldots \wedge \nu(P_m) = v_m \\ & \quad \wedge cpt(n)(v_1, \ldots, v_m, k) = 1 \end{cases}$$

The assignment ν is well-defined since (1) BN is a forest of poly-trees (see Proposition B.10), and (2) for all nodes of the form $X[v{::}a(\overline{c}), \emptyset, a(\overline{c})]$ and $X[a(\overline{c}), \emptyset, a(\overline{c})]$, $D(n) = \{\top, \bot\}$ by construction (cf. Algorithm 2). Furthermore, the assignment ν is consistent by construction. Indeed, for all variables $n \in N \setminus K(BN)$, the corresponding entry in the CPT is 1.

We now prove that $\nu \models s$. From Lemma B.5, ν's consistency, and $f(v{::}a(\overline{c})) = \nu(X[\{v{::}a(\overline{c})\}, \emptyset, a(\overline{c})])$ for all $v{::}a(\overline{c}) \in p$, it follows that $a(\overline{c}) \in g_f(p)$ iff $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$. From this and $g_f(p) = s$, it follows $\nu \models s$.

Finally, we show that $[\![BN]\!](\nu) = prob(f)$. In more detail, $[\![BN]\!](\nu) = \left[\left(\prod_{n \in N} cpt(n)\right)\right](\nu)$. This can be equivalently rewritten as follows: $[\![BN]\!](\nu) = \left[\left(\prod_{n \in K(BN)} cpt(n)\right) \cdot \left(\prod_{n \in N \setminus K(BN)} cpt(n)\right)\right](\nu)$. Furthermore, since ν is consistent and the CPTs associated with the variables in $N \setminus K(BN)$ are deterministic, $[\![BN]\!](\nu)$ can be simplified as $[\![BN]\!](\nu) = \left[\left(\prod_{n \in K(BN)} cpt(n)\right)\right](\nu)$. From this and BN's definition, it follows that $[\![BN]\!](\nu) = \prod_{\nu(X[v{::}a(\overline{c}), \emptyset, a(\overline{c})]) = \top} v \cdot \prod_{\nu(X[v{::}a(\overline{c}), \emptyset, a(\overline{c})]) = \bot} (1 - v)$. From this and ν's definition, it follows $[\![BN]\!](\nu) = \prod_{f(v{::}a(\overline{c})) = \top} v \cdot \prod_{f(v{::}a(\overline{c})) = \bot} (1 - v)$, which is equivalent to $prob(f)$.

(⇐). Let ν be a BN-total assignment such that $[\![BN]\!](\nu) = k$, ν is consistent, and $\nu \models s$, and f be the following p-probabilistic assignment: $f(v{::}a(\overline{c})) = \nu(X[\{v{::}a(\overline{c})\}, \emptyset, a(\overline{c})])$.

We now show that $[\![BN]\!](\nu) = prob(f)$. In more detail, $[\![BN]\!](\nu) = \left[\left(\prod_{n \in N} cpt(n)\right)\right](\nu)$. Furthermore, since ν is consistent and the CPTs associated with the variables in $N \setminus K(BN)$ are deterministic, $[\![BN]\!](\nu)$ can be simplified as $[\![BN]\!](\nu) = \left[\left(\prod_{n \in K(BN)} cpt(n)\right)\right](\nu)$. From this and BN's definition, it follows that $[\![BN]\!](\nu) = \prod_{\nu(X[v{::}a(\overline{c}), \emptyset, a(\overline{c})]) = \top} v \cdot \prod_{\nu(X[v{::}a(\overline{c}), \emptyset, a(\overline{c})]) = \bot} (1 - v)$. From this and ν's definition, it follows that $[\![BN]\!](\nu) = \prod_{f(v{::}a(\overline{c})) = \top} v \cdot \prod_{f(v{::}a(\overline{c})) = \bot} (1 - v)$, which is equivalent to $prob(f)$.

We still have to prove that $g_f(p) = s$. From Lemma B.5, ν's consistency, $f(v{::}a(\overline{c})) = \nu(X[\{v{::}a(\overline{c})\}, \emptyset, a(\overline{c})])$, it follows that $a(\overline{c}) \in g_f(p)$ iff $\nu(X[a(\overline{c})\downarrow_p]) = a(\overline{c})\uparrow_p$. From this and $\nu \models s$, it follows $g_f(p) = s$. □

B.3.6 Proof of the main result

Theorem B.2 shows the correctness of our encoding.

Theorem B.2. *Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that* **dom** *is a finite domain, p be a $(\Sigma, \mathbf{dom})$-relaxed acyclic* ProbLog *program, W be a witness for p, BN be the Bayesian Network generated by Algorithm 2 having p and W as input, and s be a $(\Sigma, \mathbf{dom})$-structure. Furthermore, let ν be the BN-partial assignment such that (1) for any ground atom $a(\overline{c})$, $\nu(X[a(\overline{c}\downarrow_p)]) = a(\overline{c})\uparrow_p$ iff $a(\overline{c}) \in s$, and (2) $\nu(v)$ is undefined otherwise. Then, $[\![p]\!](s) = [\![BN]\!](\nu)$.*

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, p be a $(\Sigma, \mathbf{dom})$-relaxed acyclic ProbLog program, W be a witness for p, BN be the Bayesian Network generated by Algorithm 2 having p and W as input, and s be a $(\Sigma, \mathbf{dom})$-structure. Furthermore, let ν be the BN-partial assignment such that (1) for any ground atom $a(\overline{c})$, $\nu(X[a(\overline{c}\downarrow_p)]) = a(\overline{c})\uparrow_p$ iff $a(\overline{c}) \in s$, and (2) $\nu(v)$ is undefined otherwise.

The probability $[\![p]\!](s)$ is $\Sigma_{f \in \mathcal{M}(p,s)} prob(f)$, where $\mathcal{M}(p, s)$ is the set of all assignments f such that $[\![instance(p, f)]\!] = s$. Equivalently, $\mathcal{M}(p, s)$ is the set of all probabilistic assignments f such that $g_f(p) = s$. Let K be the set of all total assignments that agree with ν for all variables of the form $X[a(\overline{c})\downarrow_p]$. The probability $[\![BN]\!](\mu)$ is $\Sigma_{\nu' \in K} [\![BN]\!](\nu')$. Since any non-consistent assignment has probability 0, $[\![BN]\!](\nu) = \Sigma_{\nu' \in K'} [\![BN]\!](\nu')$, where K' is the set of all consistent assignments in K. From this, it follows that $[\![p]\!](s) = [\![BN]\!](\nu)$ iff $\Sigma_{f \in \{f \mid g_f(p) = s\}} prob(f) = \Sigma_{\nu' \in K'} [\![BN]\!](\nu')$. The latter follows trivially from Lemma B.6, which establishes a one-to-one mapping from $\{f \mid g_f(p) = s\}$ and K' that preserves probabilities. □

B.4 Complexity of Inference

Here we prove our results about the complexity of inference for ProbLog programs.

B.4.1 Size of the encoding

Given a Bayesian Network $BN = \langle N, E, CPT \rangle$, the size of BN, denoted $|BN|$, is $|N| + |E| + \Sigma_{n \in N} |cpt(n)|$, where the size of a conditional probability table is just the number of rows in the table (i.e., the number of all assignments). The size of an atom $a(\overline{c})$ is $|a|$, whereas the size of a rule $h \leftarrow l_1, \ldots, l_n$ is $|h| + \Sigma_{1 \leq i \leq n} |l_i|$. Finally, the size of a program p is $\Sigma_{r \in p} |r|$. The *ground version* of p, denoted $gv(p)$, is $\bigcup_{r \in p} ground(p, r)$, namely the relaxed grounding of all the rules in p. Note that $ground(p) \subseteq gv(p)$.

Let p be a relaxed acyclic ProbLog program, W be a witness for p, $p' = \alpha(\beta_W(p))$ be the transformed program, $g = gv(p')$ be its ground version, and $bn(p, W) = \langle N, E, CPT \rangle$ be the corresponding Bayesian Network derived by Algorithm 2. The number of nodes in N is $O(|rules(p')| \cdot |g|)$. Indeed, there is a node $X[a(\overline{c})]$ for each ground atom in g. Moreover, for each rule $r \in p'$ and ground rule $r' \in g$, there are nodes $X[r, body(r'), head(r')]$ and $X[r, head(r')]$. Finally, the number of intermediate nodes generated by the *tree* procedure is twice the number of nodes of the form $X[r, body(r'), head(r')]$.

The number of edges in E is $O(|rules(p')|^2 \cdot |g|)$. Indeed, there is an edge $X[r, a(\overline{c})] \rightarrow X[a(\overline{c})]$ for each rule r and ground atom $a(\overline{c})$. Furthermore, there is an edge $X[b(\overline{v})] \rightarrow X[r, I, a(\overline{c})]$ for each rule r, ground rule $a(\overline{c}) \leftarrow I$, and atom $b(\overline{v}) \in I$. Finally, the number of edges introduced by the *tree* procedure is $O(|g|)$.

The size of cpt is $O(|rules(p')| \cdot |g| \cdot max(2, |rules(p)|)^{2+|rules(p)|})$. Indeed, we have a CPT for each node $n \in N$. The size of each CPT depends on (1) the number of parents for each node, and (2) the size of the domain associated to each variable. In the worst case, the size of the domain of each node is $max(2, |rules(p)|)$ (since the size of the domain of a CPT-like predicate depends only on the rules — there are no CPT-like predicates defined by ground atoms). The number of parents for intermediate nodes is 2, and the size of each CPT is $O(max(2, |rules(p)|)^3)$. The maximum number of parents for nodes of the form $X[r, a(\overline{c})]$ is 1, and the size of each CPT is $O(max(2, |rules(p)|)^2)$. The maximum number of parents for nodes of the form $X[r, I, a(\overline{c})]$ is $|rules(p')|$, and the size of each CPT is $O(max(2, |rules(p)|)^{|rules(p')+1|})$. Similarly, the maximum number of parents for nodes of the form $X[a(\overline{c})]$ is $1 + |rules(p)|$, and the size of each CPT is $O(max(2, |rules(p)|)^{|rules(p')+2|})$.

As a result, $|bn(p, W)|$ is $O(|rules(p')|^2 \cdot |g| \cdot max(2, |rules(p)|)^{2+|rules(p)|})$. Furthermore, since $|g| \in O(|prob(p')|^{|rules(p')|})$, it follows that $|bn(p, W)|$ is $O(|rules(p')|^2 \cdot |prob(p')|^{|rules(p')|} \cdot max(2, |rules(p)|)^{2+|rules(p)|})$. Finally, since $|prob(p')| \in O(|prob(p)|)$ and $|rules(p') \in O(|rules(p)|)$, we can simplify the result as follows: $|bn(p, W)|$ is $O(|rules(p)|^2 \cdot |prob(p)|^{|rules(p)|} \cdot max(2, |rules(p)|)^{2+|rules(p)|})$. Hence, the size of $bn(p, W)$ is polynomial in $|prob(p)|$.

B.4.2 Complexity Proofs

We first define the inference problem Inf. Afterwards, we analyze its complexity.

Definition B.1. Inf denotes the following decision problem:
Input: A database schema $\langle \Sigma, \mathbf{dom} \rangle$ such that **dom** is a finite domain, a ProbLog program p, a set of ground literals E, and a ground atom $a(\overline{c})$.
Output: The probability of $a(\overline{c})$ given evidence in E. □

The data complexity of $\text{Inf}(\Sigma, \mathbf{dom}, p, E, a)$ for relaxed acyclic programs can be obtained by (1) fixing $rules(p)$ and varying only $prob(p)$ (and indirectly **dom**), and (2) requiring the program to be relaxed acyclic. The data complexity of the Inf problem for relaxed-acyclic programs p is the complexity of the following decision problem:

Definition B.2. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain, R be a fixed set of ProbLog rules over Σ, E be a set of ground literals, $a(\overline{c})$ be a ground atom, and W be a CPT-schema. $\text{Inf}^{ra}_{\Sigma, R, E, a(\overline{c}), W}$ denotes the following problem:
Input: A set of probabilistic atoms E' such that (1) atoms in E and $a(\overline{c})$ refer only to constant values in E' and R, and (2) $R \cup E'$ is a relaxed acyclic ProbLog program and W is a safe CPT-schema and a witness for the acyclicity of $R \cup E'$.
Output: The probability of $a(\overline{c})$ given evidence in E. □

Theorem B.3. $\text{Inf}^{ra}_{\Sigma, R, E, a(\overline{c}), W}$ *is in* Ptime.

Proof. Let $\langle \Sigma, \mathbf{dom} \rangle$ be a database schema such that **dom** is a finite domain R be a fixed set of ProbLog rules over Σ, E be a set of ground literals, $a(\overline{c})$ be a ground atom, and W be a CPT-schema. We consider only inputs E' such that $E' \cup R$ is a relaxed acyclic ProbLog program and

W is a safe CPT-schema and a witness for the acyclicity of $R \cup E'$. The size of E' is the sum of the sizes of all atoms, where the size of an atom is its cardinality.

Let e be the number of distinct constants occurring in $E' \cup R$ and r be $|R|$. Computing $\textsc{Inf}(\Sigma, E', p, E, a)$ can be done in the following steps:

1. Transform the original program p into the program p' (by applying the α and β_W transformations).
2. Construct the ground version g of the program p'.
3. Construct the Bayesian Network $bn(p, W)$ from g.
4. Perform the inference on $bn(p, W)$.

The first step can be performed in linear time in the size of $O(e+r)$. The second step can be performed in $O(e^r)$ (because the grounding of p' can be done in $O(e^{|rules(p')|})$ and $|rules(p')| \leq r$). The third step can be performed in $O(r^4 \cdot e^{2 \cdot r} \cdot max(2, r)^{2+r})$ (constructing N and E can be done in $O(r^2 \cdot e^{2 \cdot r})$, whereas the *cpt* can be constructed in $O(r^4 \cdot e^r \cdot max(2, r)^{2+r})$). The fourth step can be performed in $O(|bn(p)|)$ since $bn(p)$ is a forest of poly-trees (see Proposition B.10). In particular, inference can be performed by (1) identifying the poly-tree that contains the atom $a(\overline{x})$ (in $O(|bn(p)|)$), (2) identifying the subset $E'' \subseteq E$ of all atoms in the poly-tree of $a(\overline{x})$ (again in $O(|bn(p)|)$), and (3) performing inference using belief-propagation (in $O(|bn(p)|)$ [105]). Therefore, the fourth step can be performed in $O(r^2 \cdot e^r \cdot max(2, r)^{2+r})$. As a result, the answer to $\textsc{Inf}$ can be computed in $O(r^2 \cdot e^{2 \cdot r} \cdot max(2, r)^{2+r})$. Furthermore, since r is fixed and the number of constants in E' is $O(|E'|)$, there is a $k \in \mathbb{N}$ such that $e \in O(|E'| + k)$. From this, it follows that there is a $k \in \mathbb{N}$ such that $\textsc{Inf}$ can be computed in $O(|E'|^k)$. Hence, the complexity of $\textsc{Inf}^{ra}_{\Sigma, R, E, a(\overline{c}), \mu}$ (and the data complexity of $\textsc{Inf}$) is $\textsc{Ptime}$. □

$\textsc{Inf}$ for $\textsc{ProbLog}$ is $\#P$-hard. This follows from the $\#P$-hardness of inference on arbitrary Bayesian networks (BNs) [105], which can be encoded in $\textsc{ProbLog}$. We now show that inference for $\textsc{ProbLog}$ programs is $\#P$-hard even in terms of data complexity. We do this using a reduction from the $\#\textsc{Sat}$ problem (counting the number of satisfying assignments of a propositional formula ϕ), which is $\#P$-hard [126].

Proposition B.18. $\textsc{Inf}$ *is* $\#P$*-hard in terms of data complexity for* $\textsc{ProbLog}$ *programs.*

Proof. We show this by reducing the $\#\textsc{Sat}$ problem to inference for a $\textsc{ProbLog}$ program p where (1) the formula can be encoded in $prob(p)$, and (2) the rules are fixed. Let ϕ be a propositional formula.

First-order Signature. Let Σ be the signature containing the following predicate symbols:

- e of arity 1, which is used to store the identifiers associated to all sub-expressions of ϕ,
- *state* of arity 1, which is used to store the propositions in the model,
- *sat* of arity 1, which is used to denote whether an expression is satisfiable or not,
- *conj* of arity 3 which is used to encode conjunctions,
- *disj* of arity 3 which is used to encode disjunctions,
- *neg* of arity 2 which is used to encode negations, and
- *prop* of arity 2 used to encode propositions.

Rules. We now define a fixed set of rules encoding the semantics of propositional logic.

$$sat(x) \leftarrow e(x), prop(x, y), state(y)$$
$$sat(x) \leftarrow e(x), neg(x, y), \neg sat(y)$$
$$sat(x) \leftarrow e(x), conj(x, y, z), sat(y), sat(z)$$
$$sat(x) \leftarrow e(x), disj(x, y, z), sat(y)$$
$$sat(x) \leftarrow e(x), disj(x, y, z), sat(z)$$

Encoding a formula ϕ. Given a formula ϕ, we define the following encoding using probabilistic ground atoms. We first associate to each sub-formula ψ of ϕ a unique identifier id_ψ. We also associate a unique identifier id_p to each propositional symbol p. For each sub-formula $\psi \wedge \gamma$, the ground atoms $e(id_{\psi \wedge \gamma})$ and $conj(id_{\psi \wedge \gamma}, id_\psi, id_\gamma)$ are in E'. For each sub-formula $\psi \vee \gamma$, the ground atoms $e(id_{\psi \vee \gamma})$ and $disj(id_{\psi \vee \gamma}, id_\psi, id_\gamma)$ are in E'. For each sub-formula $\neg\psi$, the ground atoms $e(id_{\neg\psi})$ and $neg(id_{\neg\psi}, id_\psi)$ are in E'. For each sub-formula ψ such that ψ is a propositional symbol p, the ground atoms $e(id_\psi)$ and $prop(id_\psi, id_p)$ are in E'. Finally, for each propositional symbol p, there is a propositional atom $1/2 :: state(id_p)$ in E'.

Reduction. There are $2^{n(\phi)}$ grounded instances of $R \cup E'$, where $n(\phi)$ is the number of distinct propositional symbols occurring in ϕ, each one with probability $1/2^{n(\phi)}$. The grounded instances represent all possible assignments of $\top$ and $\bot$ to the proposition symbols in ϕ. It is easy to see that $sat(id_\phi)$ can be derived in a grounded instance iff the instance represents a model for ϕ. Therefore, the probability $[\![R \cup E']\!](sat(id_\phi))$ is $k/2^{n(\phi)}$, where k is the number of models that satisfy ϕ. □

Note that the encoding shown in the previous proof can be tweaked to work for acyclic ProbLog programs but only for propositional formulae without repetitions of propositional symbols. We remark that the #Sat problem restricted to formulae without repetitions is no longer $\#P$-hard. Indeed, it can be solved in Ptime as follows: given a formula ϕ without repetitions, we can construct a poly-tree boolean Bayesian Network BN encoding ϕ in $O(|\phi|)$, where the nodes associated to propositional symbols p have a uniform probability distribution (i.e., p is $\top$ with probability $1/2$ and $\bot$ with probability $1/2$). Then, the probability associated to the root of ϕ is going to be $k/2^{n(\phi)}$, where k is the number of satisfying assignments. Since the inference on BN can be done in linear time in $|BN|$ and $|BN| \in O(|\phi|)$, then the whole problem is in Ptime.

B.5 Expressiveness

Here we show that any BN that consists of a forest of poly-trees can be represented as a relaxed acyclic program.

Proposition B.19. *Any BN that is a forest of poly-trees can be represented as a relaxed acyclic* ProbLog *program.*

Proof. Let bn be a BN that is a forest of poly-trees. We now construct the corresponding relaxed acyclic ProbLog program. In particular, for each random variable X in bn, we show how to equivalently encode it using ProbLog rules. Without loss of generality, we assume there is a unique mapping id from random variables in bn to predicate symbols identifiers. With a slight abuse of notation, we use X to refer both to the random variable and to the corresponding symbol $id(X)$.

Boolean Random Variables without parents. For each boolean random variable X such that (1) $p(X) = \emptyset$, and (2) $CPT(X) = \{\top \mapsto v, \bot \mapsto (1-v)\}$, we introduce a probabilistic atom $v::X$.

Non-Boolean Random Variables without parents. For each non-boolean random variable X with domain $\{q_1, \ldots, q_n\}$ such that (1) $p(X) = \emptyset$, and (2) $CPT(X) = \{q_1 \mapsto v_1, \ldots, q_n \mapsto v_n\}$, we introduce an annotated disjunction $v_1::X(q_1); \ldots; v_n::X(q_n)$.

Boolean Random Variables with parents. Let X be a boolean random variable X with parents $p(X) = \{Y_1, \ldots, Y_n\}$, $CPT(X)$ be the function associated with X, and $\overline{v}_1, \ldots, \overline{v}_m$ be all possible assignments to the variables in $p(X)$. We introduce m fresh predicate symbols $sw_{X,\overline{v}_1}, \ldots, sw_{X,\overline{v}_m}$. For each $\overline{v}_i$, we introduce the probabilistic atom $c_i::sw_{X,\overline{v}_i}$, where $CPT(X)(\overline{v}_i, \top) = c_i$. Finally, for each $\overline{v}_i$, we also introduce the following rules:

$$
\begin{aligned}
X \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_1), sw_{X,\overline{v}_i} \\
& \vdots \\
X \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_{2^{i-1}}), sw_{X,\overline{v}_i}
\end{aligned}
$$

where $\bar{l}_1, \ldots, \bar{l}_{2^{i-1}}$ are all possible values in $\{\top, \bot\}^{i-1}$, $\mathbf{sw}(l_1, \ldots, l_{i-1})$ is the list of literals $sw_{X,\overline{v}_1}[l_1]$, $\ldots$, $sw_{X,\overline{v}_{i-1}}[l_{i-1}]$, and given a symbol A, $A[\top] = A$, $A[\bot] = \neg A$, and $A[v] = A(v)$ if $v \notin \{\top, \bot\}$.

Non-Boolean Random Variables with parents. Let X be a non-boolean random variable X with domain $\{q_1, \ldots, q_m\}$ and parents $p(X) = \{Y_1, \ldots, Y_n\}$, $CPT(X)$ be the function associated with X, and $\overline{v}_1, \ldots, \overline{v}_n$ be all possible assignments to the variables in $p(X)$. For each $\overline{v}_1$, we introduce the probabilistic atoms $w_i^1::sw_{X,\overline{v}_i}^{q_1}, \ldots, w_i^m::sw_{X,\overline{v}_i}^{q_m}$, where for all $1 \leq i \leq n$, $w_i^j = c_i^j \cdot \left(1 - \sum_{1 \leq k < j} c_j^k\right)$ and $CPT(X)(\overline{v}_i, q_j) = c_i^j$. Finally, for each $\overline{v}_i$, we also introduce the following rules:

$$
\begin{aligned}
X[q_1] \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_1), sw_{X,\overline{v}_i}^{q_1} \\
X[q_2] \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_1), \neg sw_{X,\overline{v}_i}^{q_1}, sw_{X,\overline{v}_i}^{q_2} \\
& \vdots \\
X[q_m] \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_1), \neg sw_{X,\overline{v}_i}^{q_1}, \ldots, \neg sw_{X,\overline{v}_i}^{q_{m-1}}, sw_{X,\overline{v}_i}^{q_m} \\
& \vdots \\
X[q_1] \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_{2^{m(i-1)}}), sw_{X,\overline{v}_i}^{q_1} \\
X[q_2] \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_{2^{m(i-1)}}), \neg sw_{X,\overline{v}_i}^{q_1}, sw_{X,\overline{v}_i}^{q_2} \\
& \vdots \\
X[q_m] \leftarrow\ & Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_{2^{m(i-1)}}), \neg sw_{X,\overline{v}_i}^{q_1}, \ldots, \neg sw_{X,\overline{v}_i}^{q_{m-1}}, sw_{X,\overline{v}_i}^{q_m}
\end{aligned}
$$

where $\bar{l}_1, \ldots, \bar{l}_{2^{m(i-1)}}$ are all possible values in $\{\top, \bot\}^{m(i-1)}$, $\mathbf{sw}(l_1^1, \ldots, l_1^m, \ldots, l_{i-1}^1, \ldots, l_{i-1}^m)$ is the list of literals $sw^{q_1}_{X,\overline{v}_1}[l_1^1], \ldots, sw^{q_m}_{X,\overline{v}_1}[l_1^m], \ldots, sw^{q_1}_{X,\overline{v}_{i-1}}[l_{i-1}^1], \ldots, sw^{q_m}_{X,\overline{v}_{i-1}}[l_{i-1}^m]$, and given a predicate symbol A, $A[\top] = A$, $A[\bot] = \neg A$, and $A[v] = A(v)$ if $v \notin \{\top, \bot\}$. Observe that the above rules are equivalent to having, for each $\overline{v}_i$, the annotated disjunction $c_i^1::X[q_1]; \ldots; p_i^m::X[q_m] \leftarrow Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)]$ encoding the CPT's row.

Correctness of the encoding. The encoding presented above encodes the CPTs for the random variables. The probability that a random variable X has value v is exactly the same as the probability associated with the ground literal $X[v]$ (with the notation defined above). The encoding for variables without parents directly encodes the corresponding CPTs. For variables with parents, the only non-standard part is the use of $\mathbf{sw}(\bar{l}_{2^{i-1}}), sw_{X,\overline{v}_i}$. Note, however, that the literals in $\mathbf{sw}(\bar{l}_{2^{i-1}})$ do not influence the derivation since there is a rule for any possible values for them. We need them only for the encoding. Therefore, also the encoding of random variables with parents directly encodes the corresponding CPTs.

Relaxed Acyclicity. Let p be the program produced by the above construction. It is easy to see that all predicates associated with non-boolean random variables are safe annotated disjunctions (as we just encoded their CPTs). The horizontal partitioning π_H contains, for each non-boolean variable X, only one set RR that, in turns, contains a partition of the rules defining X based on the CPT row, whereas the vertical partitioning π_V is such that, for instance, $\pi_V(X[q_m] \leftarrow Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_1), \neg sw^{q_1}_{X,\overline{v}_i}, \ldots, \neg sw^{q_{m-1}}_{X,\overline{v}_i}, sw^{q_m}_{X,\overline{v}_i}) = \langle (Y_1[\overline{v}_i(1)], \ldots, Y_n[\overline{v}_i(n)], \mathbf{sw}(\bar{l}_1)), \epsilon, (\neg sw^{q_1}_{X,\overline{v}_i}, \ldots, \neg sw^{q_{m-1}}_{X,\overline{v}_i}, sw^{q_m}_{X,\overline{v}_i})\rangle$. Finally, the function μ is such that $\mu(X) = \emptyset$ for boolean variables X and $\mu(X) = \{1\}$ for non-boolean variables. Observe that the schema $\langle \pi_H, \pi_V, \mu\rangle$ is safe. Therefore, the program $\beta_W(p)$ is obtained by removing all constant values associated with the domains of non-boolean random values. Finally, the program $\alpha(\beta_W(p))$ collapses the rules for random variables with parents into a single rule (this follows from the use of **sw** in the rules' bodies). The acyclicity of $p' = \alpha(\beta_W(p))$ follows from (1) p' does not contain free-variables (i.e., both the literals and the ground atoms are only propositional facts), (2) bn is a forest of poly-trees, and (3) each predicate symbol $sw_{X,\overline{v}_i}$ occurs only in the rule of X. From (1), it follows that all rules are both strongly and weakly connected. From (2) and (3), it follows that (a) there are no directed cycles in $graph(p)$, and (b) for all undirected cycles U in $graph(p)$, there are U', U'' such that U is equivalent to $U' \cdot U''$ and U' is $P' \xleftarrow{r,i} P \xrightarrow{r,i} P'$ for some P, P', r, i. Since all rules are both strongly and weakly connected, it directly follows that the undirected unsafe structure $\langle P \xrightarrow{r,i} P', P \xrightarrow{r,i} P', P', U''\rangle$ is guarded. Therefore, p' is acyclic and p is a relaxed acyclic ProbLog program. □

Observe that the reduction from poly-tree Bayesian Networks to relaxed acyclic programs introduces an exponential blow-up. This is due to our use of the **sw** function for encoding the CPTs. We remark, however, that this exponential factor is not present when we ground the program, as the exponentially many rules collapse into a single rule. We also note that, in the worst case, the Bayesian Network corresponding to a relaxed acyclic ProbLog program is exponential in the program's size (as it depends on the program's grounding).

B.6 ANGERONA

Here we prove Angerona's security and complexity.

B.6.1 Security Proof

Let $C = \langle D, \Gamma\rangle$ be a system configuration and h be a C-history. The set $observations(h, u)$ is $\{\phi \mid \exists i.\, h(i) = \langle\langle u, \phi\rangle, \top, \top\rangle\} \cup \{\neg\phi \mid \exists i.\, h(i) = \langle\langle u, \phi\rangle, \top, \bot\rangle\}$.

We first prove that Angerona's result depends just on the queries in the history, and not on the actual database's state.

Proposition B.20. *Let $C = \langle D, \Gamma\rangle$ be a system configuration, ATK be a C-AtkLog model, and $\langle u, q\rangle$ be a C-query.* Angerona$(C, s, h, ATK, \langle u, q\rangle)$ = Angerona$(C, s', h', ATK, \langle u, q\rangle)$ *whenever* $observations(h, u) = observations(h', u)$.

Proof. Let $C = \langle D, \Gamma\rangle$ be a system configuration, ATK be an AtkLog model, and $\langle u, q\rangle$ be a C-query. Furthermore, let s, s' be C-states and h, h' be C-histories such that $observations(h, u) = observations(h', u)$. Assume, for contradiction's sake, that Angerona$(C, s, h, ATK, \langle u, q\rangle) \neq$ Angerona$(C, s', h', ATK, \langle u, q\rangle)$. Without loss of generality, assume that Angerona$(C, s, h, ATK, \langle u, q\rangle) = \top$ and Angerona$(C, s', h', ATK, \langle u, q\rangle) = \bot$. This happens iff either the result of *pox* or *secure* is different in the two cases. The results of *pox* and *secure*, however, depend just on the attacker's initial beliefs, the query q, and the set of formulae in $observations(h, u)$. From

this and $observations(h,u) = observations(h',u)$, it follows that $\textsc{Angerona}(C,s,h,ATK,\langle u,q\rangle) = \textsc{Angerona}(C,s',h',ATK,\langle u,q\rangle)$. □

We now prove a key result for our security proof, namely that considering only the sentences in $observations(h,u)$ is enough to determine secrecy-preservation.

Proposition B.21. *Let $C = \langle D,\Gamma\rangle$ be a system configuration, ATK be a C-AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, $u \in \mathcal{U}$ be a user, and $r = \langle s,h\rangle$ be a (C,f)-run. The following fact holds: $[\![r]\!]_{\sim_u} = \{db \in \Omega_D^\Gamma \mid \bigwedge_{\phi\in observations(h,u)}[\phi]^{db} = \top\}$.*

Proof. **(⇒).** Let $C = \langle D,\Gamma\rangle$ be a system configuration, ATK be an AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, $u \in \mathcal{U}$ be a user, and $r = \langle\langle db,U,P\rangle,h\rangle$ be a (C,f)-run. Furthermore, let $r' = \langle\langle db',U',P'\rangle,h'\rangle$ be a run in $[r]_{\sim_u}$. From this, it follows that $h|_u = h'|_u$. From this, it follows that all queries in $observations(h,u)$ have the same result in db and db'. Therefore, $db' \in \{db \in \Omega_D^\Gamma \mid \bigwedge_{\phi\in observations(h,u)}[\phi]^{db} = \top\}$.

(⇐). Let $C = \langle D,\Gamma\rangle$ be a system configuration, ATK be an AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, $u \in \mathcal{U}$ be a user, and $r = \langle\langle db,U,P\rangle,h\rangle$ be a (C,f)-run. Furthermore, let db' be a database state in $\{db' \in \Omega_D^\Gamma \mid \bigwedge_{\phi\in observations(h,u)}[\phi]^{db'}\}$. We now construct a C-state s and an history h' such that (1) $\langle s,h'\rangle$ is a run, (2) $s.db = db'$, and (3) $\langle s,h'\rangle \in [r]_{\sim_u}$. The C-state s is defined as $\langle db',U,P\rangle$, whereas the history $h' = h|_u$. We claim that $\langle s,h'\rangle$ is a run. From this and $h' = h|_u$, it follows that $\langle s,h'\rangle \sim_u r$. From this, it follows that $db' \in [\![r]\!]_{\sim_u}$.

To prove our claim that $\langle\langle db',U,P\rangle,h|_u\rangle$ is a run, we prove the stronger fact that for all $0 \le i \le |h|$, $\langle\langle db',U,P\rangle,h^i|_u\rangle$ is a run. We prove this by induction on i.

Base case. The base case $\langle\langle db',U,P\rangle,h^0|_u\rangle$ is trivial since $db' \in \Omega_D^\Gamma$ and $h^0|_u = \epsilon$, therefore $\langle\langle db',U,P\rangle,h^0|_u\rangle$ is a run.

Induction Step. Assume that $\langle\langle db',U,P\rangle,h^{i-1}|_u\rangle$ is a run. We now show that $\langle\langle db',U,P\rangle,h^i|_u\rangle$ is a run as well. Let $\langle\langle u',q\rangle,a,res\rangle$ be the last C-event in h^i. There are two cases:

- $u' \neq u$. From this, $h^i|_u = h^{i-1}|_u$. Therefore, $\langle\langle db',U,P\rangle,h^{i-1}|_u\rangle = \langle\langle db',U,P\rangle,h^i|_u\rangle$ and our claim directly follows from the induction hypothesis.
- $u' = u$. From this, $h^i|_u = h^{i-1}|_u{\cdot}\langle\langle u,q\rangle,a,res\rangle$. Assume, for contradiction's sake, that $\langle\langle db',U,P\rangle,h^i|_u\rangle$ is not a run. From $\langle\langle db',U,P\rangle,h^{i-1}|_u\rangle$ is a run (which directly follows from the induction hypothesis), it follows that there are only three cases:
  1. $f(\langle db',U,P\rangle,\langle u,q\rangle,h^{i-1}|_u) \neq a$. Since a is the security decision associated with the last event in h^i, it follows that $a = f(\langle db,U,P\rangle,\langle u,q\rangle,h^{i-1})$. From this, it follows that $f(\langle db,U,P\rangle,\langle u,q\rangle,h^{i-1}) \neq f(\langle db',U,P\rangle,\langle u,q\rangle,h^{i-1}|_u)$. From *observations*'s definition, it follows that $observations(h^{i-1},u) = observations(h^{i-1}|_u,u)$. From this and Proposition B.20, it follows that $f(\langle db,U,P\rangle,\langle u,q\rangle,h^{i-1}) = f(\langle db',U,P\rangle,\langle u,q\rangle,h^{i-1}|_u)$, leading to a contradiction.
  2. $a = \bot$ but $res \neq \dagger$. This contradicts the fact that the history is derived from h, which comes from a proper run.
  3. $a = \top$ but $res \neq [q]^{db'}$. From $res \neq [q]^{db'}$ and $res = [q]^{db}$ (since res comes from the run r), it follows that $[q]^{db} \neq [q]^{db'}$. There are two cases:
     - $[q]^{db} = \top$. From this and the definition of *observations*, it follows that $q \in observations(h,u)$. Therefore, $[q]^{db'} = \top$ follows from $db' \in \{db' \in \Omega_D^\Gamma \mid \bigwedge_{\phi\in observations(h,u)}[\phi]^{db'}\}$, leading to a contradiction.
     - $[q]^{db} = \bot$. From this and the definition of *observations*, it follows that $\neg q \in observations(h,u)$. From this and $db' \in \{db' \in \Omega_D^\Gamma \mid \bigwedge_{\phi\in observations(h,u)}[\phi]^{db'}\}$, it follows that $[\neg q]^{db'} = \top$. From this, $[q]^{db'} = \bot = [q]^{db}$, leading to a contradiction.

This completes the proof of our claim. □

Propositions B.22 and B.23 state the correctness of the *secure* and *pox* procedure.

Proposition B.22. *Let $C = \langle D,\Gamma\rangle$ be a system configuration, ATK be a C-AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, $ATK' = \lambda u \in \mathcal{U}.[\![ATK(u)]\!]_D$ be the (C,f)-attacker model associated to ATK, $r = \langle s,h\rangle$ be a run in $runs(C,f)$, and $\langle u,\psi,l\rangle$ be a secret in $r.S$. Then, $secure(C,ATK,h,\langle u,\psi,l\rangle)$ returns true iff $[\![ATK']\!](u,\langle s,h\rangle)([\![\psi]\!]) < l$.*

Proof. Let $r = \langle s,h\rangle$ be a run such that s is compatible with h. The *secure* procedure returns $[\![ATK(u)]\!]_D([\![\psi]\!]|\bigcap_{\phi\in observations(h,u)}[\![\phi]\!]) < l$. From Proposition B.21, it follows that $[\![r]\!]_{\sim_u} = \{db \in \Omega_D^\Gamma \mid \bigwedge_{\phi\in observations(h,u)}[\phi]^{db} = \top\} = \bigcap_{\phi\in observations(h,u)}[\![\phi]\!]$. We can therefore rewrite *secure*'s result as $[\![ATK(u)]\!]_D([\![\psi]\!]|[\![\langle s,h\rangle]\!]_{\sim_u}) < l$. This is equivalent to $[\![ATK']\!](u,\langle s,h\rangle)([\![\psi]\!]) < l$. □

Proposition B.23. *Let $C = \langle D, \Gamma \rangle$ be a system configuration, ATK be a C-AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, $ATK' = \lambda u \in \mathcal{U}.[\![ATK(u)]\!]_D$ be the (C, f)-attacker model associated to ATK, $r = \langle s, h \rangle$ be a run in $runs(C, f)$, and $\langle u, q \rangle$ be a query. Then, $pox(C, ATK, h, \langle u, q \rangle)$ returns true iff $[\![ATK']\!](u, \langle s, h \rangle)(q) > 0$.*

Proof. Let $r = \langle s, h \rangle$ be a run such that s is compatible with h. The *pox* procedure returns $[\![ATK(u)]\!]_D([\![q]\!] | \bigcap_{\phi \in observations(h,u)} [\![\phi]\!]) > 0$. From Proposition B.21, it follows that $[\![r]\!]_{\sim_u} = \{db \in \Omega_D^\Gamma \mid \bigwedge_{\phi \in observations(h,u)} [\phi]^{db} = \top\} = \bigcap_{\phi \in observations(h,u)} [\![\phi]\!]$. We can therefore rewrite *pox*'s result as $[\![ATK(u)]\!]_D([\![q]\!] | [\![\langle s, h \rangle]\!]_{\sim_u}) < l$. This is equivalent to $[\![ATK']\!](u, \langle s, h \rangle)([\![q]\!]) > 0$. □

We now prove that Angerona provides the desired security guarantees.

Theorem B.4. *Let $C = \langle D, \Gamma \rangle$ be a system configuration, ATK be a C-AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, and $ATK' = \lambda u \in \mathcal{U}.[\![ATK(u)]\!]_D$ be the (C, f)-attacker model associated to ATK. The PDP shown in Algorithm 1, parametrized with ATK, provides data confidentiality with respect to C and ATK'.*

Proof. Let $C = \langle D, \Gamma \rangle$ be a system configuration, ATK be a C-AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, and $ATK' = \lambda u \in \mathcal{U}.[\![ATK(u)]\!]_D$ be the (C, f)-attacker model associated to ATK. Furthermore, let f be the PDP shown in Algorithm 1. Assume, for contradiction's sake, that f does not provide confidentiality with respect to C and ATK'. From this, it follows that there is a run $r = \langle \langle db, U, P \rangle, h \rangle$, a user $u \in U$, a secret $\langle u, \phi, l \rangle \in P$, and a $0 \leq i \leq |h| - 1$ such that $[\![ATK']\!](u, r^i)([\![\phi]\!]) < l$ and $[\![ATK']\!](u, r^{i+1})([\![\phi]\!]) \geq l$. As a result, the i-th C-query is the one that leaked information. There are two cases:

- The C-query is $\langle u, q \rangle$, for some q. There are three cases:
  - The result of the PDP is $\top$ and the query q holds in the current state. From this, it follows that there is a database state (namely, the one in r) where the query q holds and the database is consistent with $observations(h^i, u)$. From this, it follows that $[\![ATK']\!](u, r^i)([\![q]\!]) > 0$. From this and Proposition B.23, it follows that $pox(C, ATK, h^i, \langle u, q \rangle)$ returns true. From this and the fact that the query has been authorized, it follows that $secure(C, ATK, h^{i+1}, \langle u, \phi, l \rangle)$ returned true. From this, the fact that r is actually a run (since q is satisfied in $r.db$), and Proposition B.22, it follows that $[\![ATK']\!](u, r^{i+1})([\![\phi]\!]) < l$, leading to a contradiction.
  - The result of the PDP is $\top$ and the query q does not hold in the current state. From this, it follows that there is a database state (namely, the one in r) where the query $\neg q$ holds and the database is consistent with $observations(h^i, u)$. From this, it follows that $[\![ATK']\!](u, r^i)([\![\neg q]\!]) > 0$. From this and Proposition B.23, it follows that $pox(C, ATK, h^i, \langle u, \neg q \rangle)$ returns true. From this and the fact that the query has been authorized, it follows that $secure(C, ATK, h^{i+1}, \langle u, \phi, l \rangle)$ returned true. From this, the fact that r is actually a run (since q is satisfied in $r.db$), and Proposition B.22, it follows that $[\![ATK']\!](u, r^{i+1})([\![\phi]\!]) < l$, leading to a contradiction.
  - The result of the PDP is $\bot$ (namely the query is not authorized). From this and Proposition B.20, it follows that for any run $r' \sim_u r^i$ the PDP result is the same. Therefore, $[\![r^i]\!]_{\sim_u} = [\![r^{i+1}]\!]_{\sim_u}$. From this, $[\![ATK']\!](u, r^i)([\![\phi]\!]) < l$, and AtkLog semantics, it follows that $[\![ATK']\!](u, r^{i+1})([\![\phi]\!]) < l$, leading to a contradiction.
- The C-query is $\langle u', q \rangle$, where $u' \neq u$ and q is a query. From this, it follows that $[\![ATK']\!](u, r^i)([\![\phi]\!]) = [\![ATK']\!](u, r^{i+1})([\![\phi]\!])$ since u's belief does not change in response to a query from another user (since $r^i|_u = r^{i+1}|_u$). From this and $[\![ATK']\!](u, r^i)([\![\phi]\!]) < l$, it follows that both $[\![ATK']\!](u, r^{i+1})([\![\phi]\!]) < l$ and $[\![ATK']\!](u, r^{i+1})([\![\phi]\!]) \geq l$, leading to a contradiction.

Since all cases ended in contradiction, this completes the proof of our claim. □

B.6.2 Complexity Proof

First, we formalize literal queries, a fragment of relational calculus that can be composed with relaxed acyclic programs without modifying acyclicity. Afterwards, we prove that for acyclic AtkLog models and literal queries, Angerona has Ptime data complexity.

A literal query is a quantifier-free relational calculus $(\Sigma, \mathbf{dom})$-formula either of the form $R(\bar{c})$ or $\neg R(\bar{c})$, where $R \in \Sigma$ and $\bar{c} \in \mathbf{dom}^{|R|}$. We say that a literal query $R(\bar{c})$ (or $\neg R(\bar{c})$) is compatible with a relaxed acyclic program p (with witness $W = \langle \pi_H, \pi_V, \mu \rangle$) iff $\mu(R) \neq \emptyset$ implies $\bar{c}\downarrow_{\mu(R)} \in pdom(R, p, \mu(R))$. We now show that literal queries can be composed with acyclic ProbLog programs without introducing cycles.

Proposition B.24. *Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a database schema, p be a relaxed acyclic program, W be a witness for p, R be a predicate symbol in Σ, $\bar{c} \in \mathbf{dom}^{|R|}$ be a tuple, and ϕ be a boolean D-query. If ϕ is a literal query compatible with p, then $p \cup PL(\phi)$ is a relaxed acyclic ProbLog program as well.*

Proof. Let $D = \langle \Sigma, \Gamma \rangle$ be a database schema, p be a relaxed acyclic program, and ϕ be a boolean D-query. Furthermore, let ϕ be a literal query. Assume, for contradiction's sake, that $p \cup PL(\phi)$ is not a relaxed acyclic program. This happens iff the program $\alpha(\beta_\mu(p \cup PL(\phi)))$ is not an acyclic program. This happens iff the program $\alpha(\beta_\mu(p \cup PL(\phi)))$ contains an unguarded cycle C. The only interesting case is when C contains rules from $PL(\phi)$. If $\phi = R(\overline{c})$, then $PL(\phi)$ contains a single rule $\mathit{fresh} \leftarrow R(\overline{c})$, where fresh is a fresh predicate symbol. Similarly, if $\phi = \neg R(\overline{c})$, then $PL(\phi)$ contains two rules $\mathit{fresh}_1 \leftarrow \mathit{fresh}_2$ and $\mathit{fresh}_2 \leftarrow \neg R(\overline{c})$, where fresh_1 and fresh_2 are fresh predicate symbols. From this, it follows that C is an undirected cycle (which is not a directed cycle). Observe that all rules in $PL(\phi)$ are both strongly and weakly connected for $\mathcal{T}$. Hence, it follows that there is always an unguarded cycle C' that can be obtained from C by removing the rules in $PL(\phi)$. This contradicts the fact that p is a relaxed acyclic program. □

Here, we show that for acyclic AtkLog models, Angerona has Ptime data complexity. Observe that we say that an AtkLog-model is relaxed acyclic if so are all the ProbLog programs in it.

Theorem B.5. *Let C be a system configuration and f be the PDP shown in Algorithm 1. For any relaxed acyclic C-AtkLog model ATK, any action (u, q) such that q is a literal query compatible with p, and any run $r \in runs(C, f)$ such that (1) all sentences in $observations(r.h, u)$, for any $u \in r.U$, are literal queries compatible with p, and (2) all secrets in $r.S$ are literal queries compatible with p, the algorithm shown in Algorithm 1 has* Ptime *data complexity.*

Proof. Let B be the ProbLog program obtained from ATK. The data complexity of f is its complexity when only the database $r.db$ and $prob(B)$ change. The algorithm in Algorithm 1 calls three times the *secure* procedure and twice the *pox* procedure. The set $observations(h, u)$ can be constructed in $O(|h|)$. From this, it follows that the program p has size $O(|prob(B)|+|rules(B)|+|h|)$. Furthermore, since all queries are literal queries, B is a relaxed acyclic ProbLog program, and Proposition B.24, it follows that p is a relaxed acyclic ProbLog program as well. From this and Theorem B.3, the inference can be performed in $O(r^2 \cdot e^r \cdot max(2, r)^{2+r})$, where $r \in O(|rules(B)| + (|r| + |prob(K)|)^{|rules(K)|})$ and $e \in O(|prob(B)|)$. Since $rules(B)$, r, and K are fixed, then the data complexity of the *secure* procedure is $O(|prob(B)|^k)$, for some $k \in \mathbb{N}$. From this and the fact that Algorithm 1 three times the *secure* procedure and twice the *pox* procedure per secret in $r.S$, it follows that the data complexity of Algorithm 1 is Ptime. □

B.6.3 Completeness Proof

We first introduce the notion of unconditionally secrecy-preserving query. Informally, a query $\langle u', q \rangle$ is unconditionally secrecy-preserving given a run r and a secret $\langle u, \psi, l \rangle$ iff disclosing the result of $\langle u', q \rangle$ in any run $r' \sim_u r$ does not violate the secret.

Definition B.3. Let $C = \langle D, \Gamma \rangle$ be a configuration, f be a C-PDP, and ATK be a (C, f)-attacker model. A query q is *unconditionally secrecy-preserving* for a (C, f)-run r, a user u, a secret $\langle u, \psi, l \rangle$, and ATK iff $[\![ATK]\!](u, r)(\psi) < l$ implies that (a) if $[\![ATK]\!](u, r)(q) > 0$, then $[\![ATK]\!](u, r)(\psi \mid \{db \in [\![r]\!]_{\sim_u} \mid [q]^{db} = \top\}) < l$, and (b) if $[\![ATK]\!](u, r)(\neg q) > 0$, then $[\![ATK]\!](u, r)(\psi \mid \{db \in [\![r]\!]_{\sim_u} \mid [q]^{db} = \bot\}) < l$. □

We say that a PDP is *complete* if it authorizes all unconditionally secrecy-preserving queries. Angerona is complete. This directly follows from (1) Proposition B.21, (2) the use of exact inference procedures for ProbLog programs, and (3) the fact that Angerona directly checks whether queries are unconditionally secrecy-preserving or not.

Proposition B.25. *Let $C = \langle D, \Gamma \rangle$ be a system configuration, ATK be a C-AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, $ATK' = \lambda u \in \mathcal{U}.[\![ATK(u)]\!]_D$ be the (C, f)-attacker model associated to ATK, $r = \langle \langle db, U, P \rangle, h \rangle$ be a run, and $\langle u, q \rangle$ be a C-query. If q is unconditionally secrecy-preserving for u, r, ATK', and all secrets $\langle u, \psi, l \rangle \in secrets(P, u)$, then f authorizes $\langle u, q \rangle$.*

Proof. Let $C = \langle D, \Gamma \rangle$ be a system configuration, ATK be a C-AtkLog model, f be the C-PDP obtained by parametrizing Algorithm 1 with C and ATK, $ATK' = \lambda u \in \mathcal{U}.[\![ATK(u)]\!]_D$ be the (C, f)-attacker model associated to ATK, $r = \langle \langle db, U, P \rangle, h \rangle$ be a run, and $\langle u, q \rangle$ be a C-query. Assume, for contradiction's sake, that our claim does not hold. Namely, there is query $\langle u, q \rangle$ such that (1) the query is unconditionally secrecy-preserving u, for all secrets $\langle u, \psi, l \rangle \in secrets(P, u)$, r, and ATK', and (2) Angerona (parametrized with ATK) does not authorize $\langle u, q \rangle$. Since Angerona does not authorize $\langle u, q \rangle$, it means that $\textsc{Angerona}(C, \langle db, U, P \rangle, h, ATK, \langle u, q \rangle) = \bot$. From this, it follows that there is a secret $\langle u, \psi, l \rangle \in secrets(P, u)$ such that:

- $secure(C, ATK, h, \langle u, \psi, l \rangle) = \top$, and
- one of the two cases hold:

- $pox(C, ATK, h, \langle u, q\rangle) = \top$ and $secure(C, ATK, h', \langle u, \psi, l\rangle) = \bot$, where $h' = h \cdot \langle\langle u, q\rangle, \top, \top\rangle$, or
- $pox(C, ATK, h, \langle u, \neg q\rangle) = \top$ and $secure(C, ATK, \langle db, U, P\rangle, h'', \langle u, \psi, l\rangle) = \bot$, where $h'' = h \cdot \langle\langle u, q\rangle, \top, \bot\rangle$.

Without loss of generality, we assume that $pox(C, ATK, h, \langle u, q\rangle) = \top$ and $secure(C, ATK, h'\langle u, \psi, l\rangle) = \bot$, where $h' = h \cdot \langle\langle u, q\rangle, \top, \top\rangle$ (the proof for the other case is identical). From $secure(C, ATK, h, \langle u, \psi, l\rangle) = \top$, Proposition B.22, and the fact that s is compatible with h, it follows that $[\![ATK']\!](u, r)(\psi) < l$. From $pox(C, ATK, h, \langle u, q\rangle) = \top$, Proposition B.22, and the fact that s is compatible with h, it follows that $[\![ATK']\!](u, r)(q) > 0$. From this and Proposition B.20, it follows that there is a system state $s' = \langle db', U, P\rangle$ such that (1) q holds in db', (2) s' is compatible with $h|_u$, and (3) $r' = \langle s', h|_u\rangle$ is a run indistinguishable from r. From this, $secure(C, ATK, h', \langle u, \psi, l\rangle) = \bot$, $secure(C, ATK, h', \langle u, \psi, l\rangle) = secure(C, ATK, \langle db, U, P\rangle, h'|_u, \langle u, \psi, l\rangle)$, and Proposition B.22, it follows that $[\![ATK']\!](u, \langle s', h'|_u\rangle)(\psi) \geq l$. From this and the definition of $h'|_u$, it follows that $[\![ATK']\!](u, r')(\psi \mid \{db \in [\![r]\!]_{\sim_u} \mid [q]^{db} = \top\}) \geq l$. From this, $r \sim_u r'$, and $[\![ATK']\!](u, r) = [\![ATK']\!](u, r')$ if $r \sim_u r'$, it follows that $[\![ATK']\!](u, r)(\psi \mid \{db \in [\![r]\!]_{\sim_u} \mid [q]^{db} = \top\}) \geq l$. Therefore, we have that $[\![ATK']\!](u, r)(\psi) < l$, $[\![ATK']\!](u, r)(q) > 0$, and $[\![ATK']\!](u, r)(\psi \mid \{db \in [\![r]\!]_{\sim_u} \mid [q]^{db} = \top\}) \geq l$. This contradicts the fact that q is an unconditionally secrecy-preserving query and completes the proof of our claim. □

Appendix C

Proofs for Chapter 6

Here, we provide proofs of all the results in Chapter 6. For simplicity's sake, we assume, without loss of generality, that all the relational calculus formulae do not use constant symbols inside predicates. For instance, instead of the formula $\exists x.\, R(x, 5, 10)$, we consider the equivalent formula $\exists x, y, z.\, R(x, y, z) \wedge y = 5 \wedge z = 10$. Furthermore, for readability, we write $secure_{P,u}(r, i \vdash_u \phi)$ instead of $secure_{P,\cong_{P,u}}(r, i \vdash_u \phi)$.

C.1 Soundness of the attacker model

Proposition C.1 states that our attacker model is *sound* with respect to the relational calculus semantics, i.e., every judgment $r, i \vdash_u \phi$ that holds in $\mathcal{ATK}_u$ is such that ϕ is satisfied in the i-th state of r. We first introduce the concept of *derivation*. Given a judgment $r, i \vdash_u \phi$, a *derivation of* $r, i \vdash_u \phi$ *with respect to* $\mathcal{ATK}_u$, or *a derivation of* $r, i \vdash_u \phi$ for short, is a proof tree, obtained by applying the rules defining $\mathcal{ATK}_u$, that ends in $r, i \vdash_u \phi$. With a slight abuse of notation, we use $r, i \vdash_u \phi$ to denote both the judgment and its derivation. The length of a derivation, denoted $|r, i \vdash_u \phi|$, is the number of rule applications in it. Note that a judgments $r, i \vdash_u \phi$ holds in $\mathcal{ATK}_u$ iff there is a derivation for it.

Proposition C.1. *Let P be an extended configuration, L be the P-LTS, u be a user, $r \in traces(L)$ be an L-run, $\phi \in RC_{bool}$ be a sentence, and $1 \leq i \leq |r|$. If $r, i \vdash_u \phi$ holds in $\mathcal{ATK}_u$, then $[\phi]^{db} = \top$, where $last(r^i) = \langle db, U, sec, T, V, c\rangle$.*

Proof. Let P be an extended configuration, L be the P-LTS, u be a user, $r \in traces(L)$ be an L-run, $\phi \in RC_{bool}$ be a sentence, and $1 \leq i \leq |r|$. Furthermore, let $r, i \vdash_u \phi$ be a judgment that holds in $\mathcal{ATK}_u$, i.e., there is a derivation d of $r, i \vdash_u \phi$ with respect to $\mathcal{ATK}_u$. We prove our claim by induction on the length of d.

Base Case: The base case is a derivation of length 1. There are a number of cases depending on the rule used to obtain $r, i \vdash_u \phi$.

1. ***SELECT** Success - 1.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{SELECT}, \phi\rangle \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db, U, sec, T, V, c'\rangle$. From the rules, it follows that $res(s) = \top$. From this and the LTS rules, it follows that $[\phi]^{db} = \top$.
2. ***SELECT** Success - 2.* The proof for this case is similar to that of ***SELECT** Success - 1.*
3. ***INSERT** Success.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, \bar{t}\rangle \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $R(\bar{t})$. From the LTS rules, it follows that $db = db'[R \oplus \bar{t}]$. From $\oplus$'s definition, it follows that $\bar{t} \in db(R)$. Therefore, from the RC's semantics, it follows that $[R(\bar{t})]^{db} = \top$. Since $\phi := R(\bar{t})$, it follows that $[\phi]^{db} = \top$.
4. ***INSERT** Success - FD.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\neg\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. We claim that $[\phi]^{db'}$ holds. From this claim and the LTS semantics, it follows that there is no tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db'(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. There are two cases:
   (a) The `INSERT` command causes an integrity exception, i.e., $Ex(s) \neq \emptyset$. From this and the LTS semantics, it follows that $db = db'$. From this and $[\phi]^{db'} = \top$, it follows that also $[\phi]^{db} = \top$.
   (b) The `INSERT` command does not cause any integrity exception, i.e., $Ex(s) = \emptyset$. From this, $[\phi]^{db'} = \top$, and $db(R) = db'(R) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$, it follows that there is no tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. From this, it follows that also $[\phi]^{db}$ holds.

   We now prove our claim that $[\phi]^{db'}$ holds. Assume, for contradiction's sake, that this is not the case. This means that there is a tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db'(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. Let db'' be the state $db'[R \oplus (\overline{v}, \overline{w}, \overline{q})]$. From this and $(\overline{v}', \overline{w}', \overline{q}') \in db'(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$, it follows that there are two tuples $(\overline{v}, \overline{w}, \overline{q})$ and $(\overline{v}, \overline{w}', \overline{q}')$ in $db''(R)$ such that $\overline{w}' \neq \overline{w}$. From this, it follows that $[\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}']^{db''} = \bot$. This contradicts the fact that $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}'$ is not in $Ex(s)$.

5. *INSERT Success - ID.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w})\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\exists \overline{y}.\, S(\overline{v}, \overline{y})$. We claim that $[\phi]^{db'}$ holds. From this claim and the LTS semantics, it follows that there is a tuple $(\overline{v}', \overline{w}')$ in $db'(S)$ such that $\overline{v}' = \overline{v}$. There are two cases:
   (a) The `INSERT` command causes an integrity exception, i.e., $Ex(s) \neq \emptyset$. From this and the LTS semantics, it follows that $db = db'$. From this and $[\phi]^{db'} = \top$, it follows that also $[\phi]^{db} = \top$.
   (b) The `INSERT` command does not cause any integrity exception, i.e., $Ex(s) = \emptyset$. From this, $[\phi]^{db'} = \top$, and $db(S) = db'(S)$, it follows that there a tuple $(\overline{v}', \overline{w}')$ in $db(S)$ such that $\overline{v}' = \overline{v}$. From this, it follows that also $[\phi]^{db}$ holds.

   We now prove our claim that $[\phi]^{db'}$ holds. Assume, for contradiction's sake, that this is not the case. This means that there is no tuple $(\overline{v}', \overline{w}')$ in $db'(S)$ such that $\overline{v}' = \overline{v}$. Let db'' be the state $db'[R \oplus (\overline{v}, \overline{w})]$. From this and the fact that there is no tuple $(\overline{v}', \overline{w}')$ in $db'(S)$ such that $\overline{v}' = \overline{v}$, it follows that there is a tuple $(\overline{v}, \overline{w})$ in $db''(R)$ and there is no tuple $(\overline{v}', \overline{w}')$ in $db''(S)$ such that $\overline{v}' = \overline{v}$. From this, it follows that $[\forall \overline{x}, \overline{z}.\, (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w}))]^{db''} = \bot$. This contradicts the fact that $\forall \overline{x}, \overline{z}.\, (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w}))$ is not in $Ex(s)$.
6. *DELETE Success.* The proof for this case is similar to that of *INSERT Success.*
7. *DELETE Success - ID.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{DELETE}, R, (\overline{v}, \overline{w})\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \overline{x} \neq \overline{v}) \vee \exists \overline{y}.\, (R(\overline{v}, \overline{y}) \wedge \overline{y} \neq \overline{w})$. We claim that $[\phi]^{db}$ holds. From this claim and the LTS semantics, it follows that there are two cases:
   (a) all tuples $(\overline{x}, \overline{y}) \in db(S)$ are such that $\overline{v} \neq \overline{x}$. There are two cases:
      i. The `DELETE` command causes an integrity exception, i.e., $Ex(s) \neq \emptyset$. From this and the LTS semantics, it follows that $db = db'$. From this and $[\phi]^{db'}$ holds, it follows that also $[\phi]^{db}$ holds.
      ii. The `DELETE` command does not cause any integrity exception, i.e., $Ex(s) = \emptyset$. From this, $[\phi]^{db'} = \top$, and $db(S) = db'(S)$, it follows that all tuples $(\overline{x}, \overline{y}) \in db(S)$ are such that $\overline{v} \neq \overline{x}$. Therefore, also $[\phi]^{db}$ holds.
   (b) there is a tuple $(\overline{v}, \overline{w}') \in db(R)$ such that $\overline{w} \neq \overline{w}'$. There are two cases:
      i. The `DELETE` command causes an integrity exception, i.e., $Ex(s) \neq \emptyset$. From this and the LTS semantics, it follows that $db = db'$. From this and $[\phi]^{db'}$ holds, it follows that also $[\phi]^{db}$ holds.
      ii. The `DELETE` command does not cause any integrity exception, i.e., $Ex(s) = \emptyset$. From this, $[\phi]^{db'} = \top$, and $db(R) = db'(R) \setminus \{(\overline{v}, \overline{w})\}$, it follows that there is a tuple $(\overline{v}, \overline{w}') \in db(R)$ such that $\overline{w} \neq \overline{w}'$. Therefore, also $[\phi]^{db}$ holds.

   We now prove our claim that $[\phi]^{db'}$ holds. Assume, for contradiction's sake, that this is not the case. This means that there is a tuple $(\overline{v}, \overline{z})$ in $db'(S)$ and there is no tuple $(\overline{v}, \overline{y}) \in db'(R)$ such that $\overline{y} \neq \overline{w}$. Let db'' be the state $db'[R \ominus (\overline{v}, \overline{w})]$. From this and the fact that there is a tuple $(\overline{v}, \overline{z})$ in $db'(S)$ and there is no tuple $(\overline{v}, \overline{y}) \in db'(R)$ such that $\overline{y} \neq \overline{w}$, it follows that there is a tuple $(\overline{v}, \overline{z})$ in $db''(S)$ and there is no tuple $(\overline{v}, \overline{y}) \in db''(R)$ such that $\overline{y} \neq \overline{w}$. From this, it follows that $[\forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})]^{db''} = \bot$. This contradicts the fact that $\forall \overline{x}, \overline{z}.\, (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})$ is not in $Ex(s)$.
8. *INSERT Exception.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSER}, R, \overline{t}\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\neg R(\overline{t})$. We claim that $[\neg R(\overline{t})]^{db'} = \top$ holds. From the LTS semantics, it follows that $db = db'$. Therefore, also $[\neg R(\overline{t})]^{db} = \top$ holds.
   We now prove our claim. Assume, for contradiction's sake, that $[\neg R(\overline{t})]^{db'} = \bot$. Therefore, $\overline{t} \in db'(R)$. From this and the definition of $\oplus$, it follows that $db' = db'[R \oplus \overline{t}]$. From the rules, it follows that $Ex(s) \neq \emptyset$. Therefore, from the LTS semantics, it follows that $db'[R \oplus \overline{t}] \notin \Omega_D^\Gamma$. From $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, it follows that $db' \in \Omega_D^\Gamma$. However, from $db' = db'[R \oplus \overline{t}]$ and $db' \in \Omega_D^\Gamma$, it follows that $db'[R \oplus \overline{t}] \in \Omega_D^\Gamma$ leading to a contradiction.
9. *DELETE Exception.* The proof for this case is similar to that of *INSERT Exception.*
10. *INSERT FD Exception.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. We claim that $[\phi]^{db'}$ holds. From this and the LTS semantics, it follows that there is a tuple $(\overline{v}, \overline{w}', \overline{q}')$ in $db'(R)$ such that $\overline{w}' \neq \overline{w}$. From this and $db = db'$, it follows that there is a tuple $(\overline{v}, \overline{w}', \overline{q}')$ in $db(R)$ such that $\overline{w}' \neq \overline{w}$. From this, it follows that also $[\phi]^{db}$ holds.
    We now prove our claim that $[\phi]^{db'}$ holds. Assume, for contradiction's sake, that this is not the case. This means that there is no tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db'(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. Therefore, for all tuples $(\overline{v}', \overline{w}', \overline{q}')$ in $db'(R)$, if $\overline{v} = \overline{v}'$, then $\overline{w}' = \overline{w}$. From this and $db'[R \oplus (\overline{v}, \overline{w}, \overline{q})](R) = db'(R) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$, it follows that for all tuples $(\overline{v}', \overline{w}', \overline{q}')$ in $db'[R \oplus (\overline{v}, \overline{w}, \overline{q})](R)$,

if $\overline{v} = \overline{v}'$, then $\overline{w}' = \overline{w}$. Furthermore, from $db' \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{v}', \overline{w}', \overline{q}')$ and $(\overline{v}', \overline{w}'', \overline{q}'')$ in $db(R)$ such that $\overline{v}' \neq \overline{v}$, then $\overline{w}' = \overline{w}$. From this and $db[R \oplus (\overline{v}, \overline{w}, \overline{q})](R) = db'(R) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$, it follows that for all tuples $(\overline{v}', \overline{w}', \overline{q}')$ and $(\overline{v}', \overline{w}'', \overline{q}'')$ in $db'[R \oplus (\overline{v}, \overline{w}, \overline{q})](R)$ such that $\overline{v}' \neq \overline{v}$, then $\overline{w}' = \overline{w}$. From these facts, it follows that $[\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}']^{db'[R \oplus (\overline{v}, \overline{w}, \overline{q})]} = \top$. This is in contradiction with the fact that the constraint $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}'$ is in $Ex(last(r^i))$.

11. *INSERT ID Exception.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}) \rangle \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c' \rangle$, and ϕ be $\forall \overline{x}, \overline{y}. S(\overline{x}, \overline{y}) \rightarrow \overline{x} \neq \overline{v}$. We claim that $[\phi]^{db'}$ holds. From this claim and the LTS semantics, it follows that there is no tuple $(\overline{v}, \overline{w}')$ in $db'(S)$. From this and $db(S) = db'(S)$, it follows that there no tuple $(\overline{v}, \overline{w}')$ in $db(S)$. From this, it follows that also $[\phi]^{db}$ holds.
    We now prove our claim that $[\phi]^{db'}$ holds. Assume, for contradiction's sake, that this is not the case. This means that there is a tuple $(\overline{v}, \overline{w}')$ in $db'(S)$, for some $\overline{w}'$. From $db' \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{x}, \overline{z}) \in db'(R)$ such that $\overline{x} \neq \overline{v}$, there is a tuple $(\overline{x}, \overline{y}) \in db'(S)$. From this, $(\overline{v}, \overline{w}')$ in $db'(S)$, $db'[R \oplus (\overline{v}, \overline{w})](S) = db'(S)$, and $db'[R \oplus (\overline{v}, \overline{w})](R) = db'(R) \cup \{(\overline{v}, \overline{w})\}$, it follows that for all tuples $(\overline{x}, \overline{z}) \in db'[R \oplus (\overline{v}, \overline{w})](R)$, there is a tuple $(\overline{x}, \overline{y}) \in db'[R \oplus (\overline{v}, \overline{w})](S)$. From these facts, it follows that $[\forall \overline{x}, \overline{z}. (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. S(\overline{x}, \overline{w}))]^{db'[R \oplus (\overline{v}, \overline{w})]} = \top$. This is in contradiction with the fact that the constraint $\forall \overline{x}, \overline{z}. (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. S(\overline{x}, \overline{w}))$ is in $Ex(last(r^i))$.
12. *DELETE FD Exception.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{DELETE}, R, (\overline{v}, \overline{w}) \rangle \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U, sec, T, V, c' \rangle$, and ϕ be $\exists \overline{z}. S(\overline{v}, \overline{z}) \wedge \forall \overline{y}. (R(\overline{v}, \overline{y}) \rightarrow \overline{y} = \overline{w})$. We claim that $[\phi]^{db'}$ holds. From this claim and the LTS semantics, it follows that there is a tuple $(\overline{v}, \overline{z})$ in $db'(S)$ and all tuples $(\overline{v}, \overline{y}) \in db'(R)$ are such that $\overline{y} = \overline{w}$. From $(\overline{v}, \overline{z})$ in $db'(S)$ and $db(S) = db'(S)$, it follows that $(\overline{v}, \overline{z})$ in $db'(S)$. From the fact that all tuples $(\overline{v}, \overline{y}) \in db'(R)$ are such that $\overline{y} = \overline{w}$ and $db(R) = db'(R)$, it follows that all tuples $(\overline{v}, \overline{y}) \in db(R)$ are such that $\overline{y} = \overline{w}$. From $(\overline{v}, \overline{z})$ in $db(S)$ and the fact that all tuples $(\overline{v}, \overline{y}) \in db(R)$ are such that $\overline{y} = \overline{w}$, it follows that $[\phi]^{db}$ holds.
    We now prove our claim that $[\phi]^{db'}$ holds. Assume, for contradiction's sake, that this is not the case. There are two cases:
    (a) all tuples $(\overline{x}, \overline{y}) \in db'(S)$ are such that $\overline{v} \neq \overline{x}$. From $db' \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{x}, \overline{y}) \in db(S)$ such that $\overline{v} \neq \overline{x}$, there is a tuple $(\overline{x}, \overline{z}) \in db(R)$. From these facts, $db'[R \ominus (\overline{v}, \overline{w})](S) = db'(S)$, and $db'[R \ominus (\overline{v}, \overline{w})](R) = db'(R) \setminus \{(\overline{v}, \overline{w})\}$, it follows that for all tuples $(\overline{x}, \overline{y}) \in db'[R \ominus (\overline{v}, \overline{w})](S)$, there is a tuple $(\overline{x}, \overline{z}) \in db'[R \ominus (\overline{v}, \overline{w})](R)$. From this, it follows that $[\forall \overline{x}, \overline{z}. (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. R(\overline{x}, \overline{w}))]^{db'[R \ominus (\overline{v}, \overline{w}))]} = \top$. This contradicts the fact that the constraint $\forall \overline{x}, \overline{z}.(S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. R(\overline{x}, \overline{w}))$ is in $Ex(last(r^i))$.
    (b) there is a tuple $(\overline{v}, \overline{w}') \in db'(R)$ such that $\overline{w} \neq \overline{w}'$. Furthermore, from $db' \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{x}, \overline{y}) \in db'(S)$ such that $\overline{v} \neq \overline{x}$, there is a tuple $(\overline{x}, \overline{z}) \in db'(R)$. From these facts, $db'[R \ominus (\overline{v}, \overline{w})](S) = db'(S)$, and $db'[R \ominus (\overline{v}, \overline{w})](R) = db'(R) \setminus \{(\overline{v}, \overline{w})\}$, it follows that for all tuples $(\overline{x}, \overline{y}) \in db'[R \ominus (\overline{v}, \overline{w})](S)$, there is a tuple $(\overline{x}, \overline{z}) \in db'[R \ominus (\overline{v}, \overline{w})](R)$. From this, it follows that $[\forall \overline{x}, \overline{z}. (S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. R(\overline{x}, \overline{w}))]^{db'[R \ominus (\overline{v}, \overline{w}))]} = \top$. This contradicts the fact that the constraint $\forall \overline{x}, \overline{z}.(S(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}. R(\overline{x}, \overline{w}))$ is in $Ex(last(r^i))$.
13. *Integrity Constraint.* The proof of this case follows trivially from the fact that for any state $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and any $\gamma \in \Gamma$, $[\gamma]^{db} = \top$ holds by definition.
14. *Learn GRANT/REVOKE Backward.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c' \rangle$, and t be a trigger whose `WHEN` condition is ϕ and whose action is either a `GRANT` or a `REVOKE`. From the rule's definition, it follows $sec \neq sec'$. We now prove that $[\phi]^{db} = \top$. Assume, for contradiction's sake, that $[\phi]^{db} = \bot$. From this and the LTS rules for the triggers, it follows that the trigger t is disabled. Therefore, according to the *Trigger Disabled* rule, $sec = sec'$, which leads to a contradiction.
15. *Trigger GRANT Disabled Backward.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c' \rangle$, and t be a trigger whose `WHEN` condition is ψ, and ϕ be $\neg\psi$. Furthermore, let $g \in \Omega_{U,D}^{sec}$ be the `GRANT` added by the trigger. From the rule's definition, it follows $g \notin sec'$. We now prove that $[\phi]^{db} = \top$. Assume, for contradiction's sake, that $[\phi]^{db} = \bot$. This would imply that the trigger t is enabled. There are two cases:
    (a) t's execution is authorized. Therefore, $g \in sec'$, which contradicts $g \notin sec'$.
    (b) t's execution is not authorized. This contradicts $secEx(s) = \bot$.
16. *Trigger REVOKE Disabled Backward.* The proof for this case is similar to that of *Trigger GRANT Disabled Backward.*
17. *Trigger INSERT FD Exception.* The proof for this case is similar to that of *INSERT FD Exception.*
18. *Trigger INSERT ID Exception.* The proof for this case is similar to that of *INSERT ID Exception.*
19. *Trigger DELETE ID Exception.* The proof for this case is similar to that of *DELETE ID Exception.*
20. *Trigger Exception.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$,

$last(r^{i-1}) = \langle db, U, sec', T, V, c' \rangle$, and t be a trigger whose `WHEN` condition is ϕ and whose action is act. From the rule's definition, it follows that t is enabled and that the evaluation of the `WHEN` condition is authorized. From this and the LTS's rules, it follows that $[\phi]^{db} = \top$.

21. *Trigger `INSERT` Exception.* The proof for this case is similar to that of *`INSERT` Exception.*
22. *Trigger `DELETE` Exception.* The proof for this case is similar to that of *`DELETE` Exception.*
23. *Trigger Rollback `INSERT`.* Let i be such that $r^i = r^{i-n-1} \cdot \langle u, \texttt{INSERT}, R, \bar{t} \rangle \cdot s_1 \cdot t_1 \cdot s_2 \cdot \ldots \cdot t_n \cdot s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$ and $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and ϕ be $\neg R(\bar{t})$. Furthermore, let $last(r^{i-n-1}) = \langle db', U', sec', T', V', c' \rangle$ and s_n be $\langle db, U, sec, T, V, c \rangle$. Assume, for contradiction's sake, that $[\phi]^{db} = \bot$. Therefore, $\bar{t} \in db(R)$. From the LTS rules, it follows that $db' = db$. From this and $\bar{t} \in db(R)$, it follows $\bar{t} \in db'(R)$. From r's definition and the LTS rule *`INSERT` Success - 2*, it follows that $\bar{t} \notin db'(R)$, which leads to a contradiction.
24. *Trigger Rollback `DELETE`.* The proof for this case is similar to that of *Trigger Rollback `INSERT`.*
25. *Learn from deny - actions.* There are $r, r', r'' \in traces(L)$, $1 < i \leq |r|$, $a \in \mathcal{A}_{D,u}$, $s, s' \in \Omega_M$, and ϕ such that: $r^i = r^{i-1} \cdot a \cdot s$, $r' = r'' \cdot a \cdot s'$, $r^{i-1} \cong_{P,u} r''$, $secEx(s') \neq secEx(s)$, $[\phi]^{last(r^{i-1}).db} = \top$, and $[\phi]^{last(r'').db} = \bot$. Then, $[\phi]^{last(r^{i-1}).db} = \top$ follows directly from the rule.
26. *Learn from deny - triggers.* The proof is similar to that of *Learn from deny - actions.*

This completes the proof of the base step.

Induction Step: Assume that the claim holds for any derivation of $r, j \vdash_u \psi$ such that $|r, j \vdash_u \psi| < |r, i \vdash_u \phi|$. We now prove that the claim also holds for $r, i \vdash_u \phi$. There are a number of cases depending on the rule used to obtain $r, i \vdash_u \phi$.

1. *View.* The proof of this case follows trivially from the semantics of the relational calculus extended over views.
2. *Propagate Forward `SELECT`.* Let i be such that $r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \psi \rangle \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and $last(r^i) = \langle db', U', sec', T', V', c' \rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this, the induction hypothesis, and $last(r^i) = \langle db', U', sec', T', V', c' \rangle$, it follows that $[\phi]^{db'} = \top$. From the LTS semantics, it follows that $db = db'$. From this and $[\phi]^{db'} = \top$, it follows that $[\phi]^{db} = \top$.
3. *Propagate Forward `GRANT/REVOKE`.* The proof for this case is similar to that of *Propagate Forward `SELECT`.*
4. *Propagate Forward `CREATE`.* The proof for this case is similar to that of *Propagate Forward `SELECT`.*
5. *Propagate Backward `SELECT`.* Let i be such that $r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \psi \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$ and $last(r^i) = \langle db, U, sec, T, V, c \rangle$. From the rule's definition, $r, i+1 \vdash_u \phi$ holds. From this, the induction hypothesis, $r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \psi \rangle \cdot s$, and $s = \langle db, U, sec, T, V, c \rangle$, it follows that $[\phi]^{db'} = \top$. From the LTS semantics, it follows that $db = db'$. From this and $[\phi]^{db'} = \top$, it follows that $[\phi]^{db} = \top$.
6. *Propagate Backward `GRANT/REVOKE`.* The proof is similar to that of *Propagate Backward `SELECT`.*
7. *Propagate Backward `CREATE TRIGGER`.* The proof is similar to that of *Propagate Backward `SELECT`.*
8. *Propagate Backward `CREATE VIEW`.* Let i be such that $r^{i+1} = r^i \cdot \langle u, \texttt{CREATE}, o \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$ and $last(r^i) = \langle db, U, sec, T, V, c \rangle$. From the rule's definition, $r, i+1 \vdash_u \phi'$ holds. From this, the induction hypothesis, $r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \psi \rangle \cdot s$, and $s = \langle db, U, sec, T, V, c \rangle$, it follows that $[\phi']^{db'} = \top$. From the definition of *replace*, it follows that $replace(\phi', o)$ and ϕ' are semantically equivalent. From this and $[\phi']^{db'} = \top$, $[replace(\phi', o)]^{db'} = \top$. From the LTS semantics, it follows that $db = db'$. From this and $[replace(\phi', o)]^{db'} = \top$, it follows that $[replace(\phi', o)]^{db} = \top$.
9. *Rollback Backward - 1.* Let i be such that $r^i = r^{i-n-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s_1 \cdot t_1 \cdot s_2 \cdot \ldots \cdot t_n \cdot s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. Furthermore, let s_n be $\langle db', U', sec', T', V', c' \rangle$ and $last(r^{i-n-1})$ be $\langle db, U, sec, T, V, c \rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this, the induction hypothesis, and $s_n = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, it follows that $[\phi]^{db'} = \top$. From the LTS semantics, it follows that $db = db'$ (because a roll-back happened). From this and $[\phi]^{db'} = \top$, it follows that $[\phi]^{db} = \top$.
10. *Rollback Backward - 2.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this, the induction hypothesis, $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s$, and $s = \langle db', U', sec', T', V', c' \rangle$, it follows that $[\phi]^{db'} = \top$. From the LTS semantics, it follows that $db = db'$ (because a roll-back happened). From this and $[\phi]^{db'} = \top$, it follows that $[\phi]^{db} = \top$.
11. *Rollback Forward - 1.* Let i be such that $r^i = r^{i-n-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s_1 \cdot t_1 \cdot s_2 \cdot \ldots \cdot t_n \cdot s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. Furthermore, let s_n be $\langle db, U, sec, T, V, c \rangle$ and $last(r^{i-n-1})$ be $\langle db', U', sec', T', V', c' \rangle$. From the rule's definition,

$r, i-n-1 \vdash_u \phi$ holds. From this, the induction hypothesis, and $last(r^{i-n-1}) = \langle db', U', sec', T', V', c' \rangle$, it follows that $[\phi]^{db'} = \top$. From the LTS semantics, it follows that $db = db'$ (because a roll-back happened). From this and $[\phi]^{db'} = \top$, it follows that $[\phi]^{db} = \top$.

12. *Rollback Forward - 2.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this, the induction hypothesis, and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$, it follows that $[\phi]^{db'} = \top$. From the LTS semantics, it follows that $db = db'$ (because a roll-back happened). From this and $[\phi]^{db'} = \top$, it follows that $[\phi]^{db} = \top$.
13. *Propagate Forward `INSERT/DELETE` Success.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this, the induction hypothesis, and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$, it follows that $[\phi]^{db'} = \top$. From $revise(r^{i-1}, \phi, r^i) = \top$, it follows that R does not occur in ϕ. From the LTS semantics, it follows that $db(R') = db'(R')$ for all $R' \neq R$. From this and the fact that R does not occur in ϕ, it follows that $[\phi]^{db} = \top$.
14. *Propagate Forward `INSERT` Success - 1.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s$, where op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ and $r, i-1 \vdash_u R(\bar{t})$ hold. From this, the induction hypothesis, and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$, it follows that $[\phi]^{db'} = \top$ and $[R(\bar{t})]^{db'} = \top$. From $[R(\bar{t})]^{db'} = \top$ and the relational calculus' semantics, it follows that $\bar{t} \in db'(R)$. From the LTS semantics, $db = db'[R \oplus \bar{t}]$. From this, it follows that $db(R') = db'(R')$ for all $R' \neq R$ and $db(R) = db'(R) \cup \{\bar{t}\}$. From this and $\bar{t} \in db'(R)$, it follows that $db(R) = db'(R)$. Therefore, $db = db'$. From this and $[\phi]^{db'} = \top$, it follows that $[\phi]^{db} = \top$.
15. *Propagate Forward `DELETE` Success - 1.* The proof for this case is similar to that of *Propagate Forward `INSERT` Success - 1.*
16. *Propagate Backward `INSERT/DELETE` Success.* The proof for this case is similar to that of *Propagate Forward `INSERT/DELETE` Success.*
17. *Propagate Backward `INSERT` Success - 1.* The proof for this case is similar to that of *Propagate Forward `INSERT` Success - 1.*
18. *Propagate Backward `DELETE` Success - 1.* The proof for this case is similar to that of *Propagate Forward `DELETE` Success - 1.*
19. *Reasoning.* Let Φ be a subset of $\{\phi \mid r, i \vdash_u \phi\}$ and $last(r^i) = \langle db, U, sec, T, V, c \rangle$. From the induction hypothesis, it follows that $[\phi]^{db} = \top$ for any $\phi \in \Phi$. From the rule's definition, it follows that $\Phi \models_{fin} \gamma$. From this and $[\phi]^{db} = \top$ for any $\phi \in \Phi$, it follows that $[\gamma]^{db} = \top$.
20. *Learn `INSERT` Backward - 3.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, \bar{t} \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, and ϕ be $\neg R(\bar{t})$. We prove that $[\neg R(\bar{t})]^{db} = \top$. Assume, for contradiction's sake, that $[\neg R(\bar{t})]^{db} = \bot$. From this, it follows that $\bar{t} \in db(R)$. From this and the LTS semantics, it follows that $db = db'$ because $db' = db[R \oplus \bar{t}]$. However, from the rule's definition, there is a ψ such that $r, i-1 \vdash_u \psi$ and $r, i \vdash_u \neg\psi$ hold. From this, the induction hypothesis, $s = \langle db', U', sec', T', V', c' \rangle$, and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, it follows that $[\psi]^{db} = \top$ and $[\neg\psi]^{db'} = \top$. Therefore, $[\psi]^{db} = \top$ and $[\psi]^{db'} = \bot$. Hence, $db \neq db'$ leading to a contradiction with $db = db'$.
21. *Learn `DELETE` Backward - 3.* The proof is similar to that of *Learn `INSERT` Backward - 3.*
22. *Propagate Forward Disabled Trigger.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, and t be a trigger. Furthermore, let ψ be t's condition where all free variables are replaced with the values in $tpl(last(r^{i-1}))$. From the rule's definition, it follows that $r, i-1 \vdash_u \neg\psi$ holds. From this and the induction hypothesis, it follows that $[\psi]^{db'} = \bot$. From this, the fact that ψ is t's `WHEN` condition, and the rule *Trigger Disabled*, it follows that $db = db'$. From the rule's definition, it follows that $r, i-1 \vdash_u \phi$ holds. From this, the induction hypothesis, and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, it follows that $[\phi]^{db'} = \top$. From this and $db = db'$, it follows that $[\phi]^{db} = \top$.
23. *Propagate Backward Disabled Trigger.* The proof for this case is similar to that of *Propagate Forward Disabled Trigger.*
24. *Learn `INSERT` Forward.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, and t be a trigger, and ϕ be $R(\bar{t})$. Furthermore, let ψ be t's condition where all free variables are replaced with the values in $tpl(last(r^{i-1}))$. From the rule's definition, it follows that $r, i-1 \vdash_u \psi$ holds. From this and the induction hypothesis, it follows that $[\psi]^{db'} = \bot$. Furthermore, from the rule's definition, it follows that $secEx(s) = \bot$ and $Ex(s) = \emptyset$. From this, the fact that ψ is t's `WHEN` condition, $[\psi]^{db'} = \bot$, and the rule *Trigger `DELETE-INSERT` Success*, it follows that $db = db'[R \oplus \bar{t}]$. From the definition of $\oplus$, it follows that $\bar{t} \in db(R)$. From this, it follows that $[\phi]^{db} = \top$.
25. *Learn `INSERT` - FD.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$,

$last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be $\neg\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. Furthermore, let ψ be t's condition where all free variables are replaced with the values in $tpl(last(r^{i-1}))$ and $\langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q}) \rangle$ be t's actual action. We claim that $db(R) = db'(R) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$. Furthermore, we claim that $[\phi]^{db}$ holds. From these claims, it follows that there is no tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. From this and $db(R) = db'(R) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$, it follows that there is no tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db'(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. From this, it follows that also $[\phi]^{db'}$ holds.
We now prove our claim that $db(R) = db'(R) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$. Assume, for contradiction's sake, that this is not the case. Since db is obtained from db', this would imply that the trigger t is disabled. Hence, this would imply that $[\psi]^{db'} = \bot$. From the rule's definition, $r, i-1 \vdash_u \psi$. From this, the induction's hypothesis, and $last(r^{i-1}) = \langle db', U, sec, T, V, c' \rangle$, it follows that $[\psi]^{db'} = \top$, which contradicts $[\psi]^{db'} = \bot$.
We now prove our claim that $[\phi]^{db}$ holds. Assume, for contradiction's sake, that this is not the case. This means that there is a tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. Note that, as we proved before, $(\overline{v}, \overline{w}, \overline{q}) \in db(R)$. Therefore, there are two tuples $(\overline{v}, \overline{w}, \overline{q})$ and $(\overline{v}, \overline{w}', \overline{q}')$ in $db(R)$ such that $\overline{w}' \neq \overline{w}$. From this, it follows that $[\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}')]^{db} = \bot$. This is in contradiction with the fact that the constraint $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}')$ is in Γ. Indeed, since the constraint is in Γ, any state in Ω_D^Γ must satisfy it.

26. *Learn* `INSERT` *- FD - 1.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be $\neg\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. Furthermore, let ψ be t's condition where all free variables are replaced with the values in $tpl(last(r^{i-1}))$ and $\langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q}) \rangle$ be t's actual action. From the rule's definition, $r, i-1 \vdash_u \psi$. From this, the induction's hypothesis, and $last(r^{i-1}) = \langle db', U, sec, T, V, c' \rangle$, it follows that $[\psi]^{db'} = \top$. From this and the LTS semantics, it follows that the trigger t is enabled in $last(r^{i-1})$. We now prove our claim that $[\phi]^{db'}$ holds. Assume, for contradiction's sake, that this is not the case. This means that there is a tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db'(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$. Let db'' be the state $db'[R \oplus (\overline{v}, \overline{w}, \overline{q})]$. From $db'' = db'[R \oplus (\overline{v}, \overline{w}, \overline{q})]$, and the fact that there is a tuple $(\overline{v}', \overline{w}', \overline{q}')$ in $db'(R)$ such that $\overline{v}' = \overline{v}$ and $\overline{w}' \neq \overline{w}$, it follows that there are two tuples $(\overline{v}, \overline{w}, \overline{q})$ and $(\overline{v}, \overline{w}', \overline{q}')$ in $db''(R)$ such that $\overline{w}' \neq \overline{w}$. From this, it follows that $[\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}')]^{db''} = \bot$. Since the trigger t is enabled, this contradicts the fact that $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}')$ is not in $Ex(s)$.
27. *Learn* `INSERT` *- ID.* The proof of this case is similar to that of *Learn* `INSERT` *- FD.* See also the proof of `INSERT` *Success - ID.*
28. *Learn* `INSERT` *- ID - 1.* The proof of this case is similar to that of *Learn* `INSERT` *- FD - 1.* See also the proof of `INSERT` *Success - ID.*
29. *Learn* `INSERT` *Backward - 1.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, $t \in \mathcal{TRIGGER}_D$, and ϕ be t's actual `WHEN` condition, where all free variables are replaced with the values in $tpl(last(r^{i-1}))$. From the rule's definition, it follows that there is a ψ such that $r, i-1 \vdash_u \psi$ and $r, i \vdash_u \neg\psi$. From this, the induction's hypothesis, $s = \langle db', U', sec', T', V', c' \rangle$, and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, it follows that $[\psi]^{db} = \top$ and $[\neg\psi]^{db'} = \top$. Therefore, $[\psi]^{db} = \top$ and $[\psi]^{db'} = \bot$. Hence, $db \neq db'$. We now prove that $[\phi]^{db} = \top$. Assume, for contradiction's sake, that $[\phi]^{db} = \bot$. From the rule's definition, it follows that $secEx(s) = \bot$. Therefore, $f(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi \rangle) = \top$. From this, $[\phi]^{db} = \bot$, and the rule *Trigger Disabled*, it follows that $db = db'$, which contradicts $db \neq db'$.
30. *Learn* `INSERT` *Backward - 2.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, $t \in \mathcal{TRIGGER}_D$, and ϕ be $\neg R(\overline{t})$. Furthermore, let $act = \langle u', \texttt{INSERT}, R, \overline{t} \rangle$ be t's actual action. From the rule's definition, it follows that there is a ψ such that $r, i-1 \vdash_u \psi$ and $r, i \vdash_u \neg\psi$. From this, the induction's hypothesis, $s = \langle db', U', sec', T', V', c' \rangle$, and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, it follows that $[\psi]^{db} = \top$ and $[\neg\psi]^{db'} = \top$. Therefore, $[\psi]^{db} = \top$ and $[\psi]^{db'} = \bot$. Hence, $db \neq db'$. We now prove that $[\phi]^{db} = \top$. Assume, for contradiction's sake, that $[\phi]^{db} = \bot$. Therefore, $\overline{t} \in db(R)$. From this and $act = \langle u', \texttt{INSERT}, R, \overline{t} \rangle$, it follows that $db' = db[R \oplus \overline{t}]$. From this and $\oplus$'s definition, it follows that $db'(R') = db(R')$ for all $R' \neq R$ and $db'(R) = db(R) \cup \{\overline{t}\}$. From $db'(R) = db(R) \cup \{\overline{t}\}$ and $\overline{t} \in db(R)$, it follows that $db'(R) = db(R)$. From this and $db'(R') = db(R')$ for all $R' \neq R$, it follows that $db' = db$, which contradicts $db \neq db'$.
31. *Learn* `DELETE` *Forward.* The proof of this case is similar to that of *Learn* `INSERT` *Forward.*
32. *Learn* `DELETE` *- ID.* The proof of this case is similar to that of *Learn* `INSERT` *- FD.* See also the proof of `DELETE` *Success - ID.*
33. *Learn* `DELETE` *- ID - 1.* The proof of this case is similar to that of *Learn* `INSERT` *- FD - 1.* See also the proof of `DELETE` *Success - ID.*
34. *Learn* `DELETE` *Backward - 1.* The proof is similar to that of *Learn* `INSERT` *Backward - 1.*

35. *Learn* `DELETE` *Backward - 2.* The proof is similar to that of *Learn* `INSERT` *Backward - 2.*
36. *Propagate Forward Trigger Action.* The proof of this case is similar to *Propagate Forward* `INSERT/DELETE` *Success.*
37. *Propagate Backward Trigger Action.* The proof of this case is similar to *Propagate Backward* `INSERT/DELETE` *Success.*
38. *Propagate Forward* `INSERT` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `INSERT` *Success - 1.*
39. *Propagate Forward* `DELETE` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `DELETE` *Success - 1.*
40. *Propagate Backward* `INSERT` *Trigger Action.* The proof of this case is similar to that of *Propagate Backward* `INSERT` *Success - 1.*
41. *Propagate Backward* `DELETE` *Trigger Action.* The proof of this case is similar to that of *Propagate Backward* `DELETE` *Success - 1.*
42. *Trigger FD* `INSERT` *Disabled Backward.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$, $t \in \mathcal{TRIGGER}_D$, and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, and ψ be $\neg\phi[\overline{x}^{|R'|} \mapsto tpl(last(r^{i-1}))]$. Furthermore, let $act = \langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q}) \rangle$ be t's actual action. From the rule's definition, it follows that $r, i-1 \vdash_u \exists \overline{y}, \overline{z}.R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$ holds. From this, the induction hypothesis, and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, it follows that $[\exists \overline{y}, \overline{z}.R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}]^{db} = \top$. Therefore, there is a tuple $(\overline{v}, \overline{w}', \overline{z}') \in db(R)$ such that $\overline{w}' \neq \overline{w}$. We now prove that $[\psi]^{db} = \top$. Assume, for contradiction's sake, that this is not the case, namely that $[\phi[\overline{x} \mapsto tpl(last(r^{i-1}))]]^{db} = \top$. There are two cases:
    (a) the trigger t is enabled and the action act is authorized. In this case, the database $db[R \oplus (\overline{v}, \overline{w}, \overline{q})] \notin \Omega_D^\Gamma$ because $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. (R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \rightarrow \overline{y} = \overline{y}' \in \Gamma$ and there is a tuple $(\overline{v}, \overline{w}', \overline{z}') \in db(R)$ such that $\overline{w}' \neq \overline{w}$. Therefore, the resulting state would be such that $Ex(s) \neq \emptyset$. This contradicts the fact that, according to the rule's definition, $Ex(s) = \emptyset$.
    (b) the trigger t is enabled and the action act is not authorized. Therefore, the resulting state would be such that $secEx(s) = \top$. This contradicts the fact that, according to the rule's definition, $secEx(s) = \bot$.
43. *Trigger ID* `INSERT` *Disabled Backward.* The proof of this case is similar to that of *Trigger FD* `INSERT` *Disabled Backward.*
44. *Trigger ID* `DELETE` *Disabled Backward.* The proof of this case is similar to that of *Trigger FD* `INSERT` *Disabled Backward.*

This completes the proof of the induction step.

This completes the proof of the theorem. □

C.2 Indistinguishability is an equivalence relation

We now show that our indistinguishability definition is, indeed, an equivalence relation over traces.

Proposition C.2. *Let $P = \langle M, f \rangle$ be an extended configuration, L be the P-LTS, and $u \in \mathcal{U}$ be a user. The indistinguishability relation $\cong_{P,u}$ is an equivalence relation over traces(L).*

Proof. We now prove that $\cong_{P,u}$ is reflexive, symmetric, and transitive. This implies the fact that $\cong_{P,u}$ is an equivalence relation over $traces(L)$. In the following, let $P = \langle M, f \rangle$ be an extended configuration, L be the P-LTS, and $u \in \mathcal{U}$ be a user. From the definition of data indistinguishability and Proposition 3.1, it follows that the data-indistinguishability relation $\cong_{M,u}^{data}$ is an equivalence relation over the set of all system states.

Reflexivity Let $r \in traces(L)$ be a run. It follows trivially that $r|_u = r|_u$. From this, it follows that $r|_u$ and $r|_u$ are consistent. It is easy to see that r is indistinguishable from r. Indeed, the database states are the same in r and r and the data-indistinguishability relation is reflexive.

Symmetry Let $r, r' \in traces(L)$ be two runs such that $r \cong_{P,u} r'$. From this, it follows that $r|_u$ and $r'|_u$ are consistent. Note that the consistency definition is symmetric. Therefore, also $r'|_u$ and $r|_u$ are consistent. From this and the symmetry of data indistinguishability, it follows that $\cong_{P,u}$ is symmetric.

Transitivity Let $r, r', r'' \in traces(L)$ be three runs such that $r \cong_{P,u} r'$ and $r' \cong_{P,u} r''$. From this it follows that $r|_u$ and $r'|_u$ are consistent and $r'|_u$ and $r''|_u$ are consistent. It is easy to see that also $r|_u$ and $r''|_u$ are consistent as well. From this and the transitivity of data indistinguishability, it follows that $\cong_{P,u}$ is transitive. □

C.3 Database Integrity Proofs

Here, we prove that the enforcement mechanism f from Section 6.8 provides database integrity.

C.3.1 Extend function

Proposition C.3 states that the *extend* function produces only views that are equivalent to those given as input.

Proposition C.3. *Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ be an M-system state, and $V' \subseteq V$ be a set of views with owner's privileges. For each view $v \in extend(M, s, V')$, there is a view $v' \in V'$ such that v and v' disclose the same data.*

Proof. Assume, for contradiction's sake, that there is a view $v \in extend(M, s, V')$ such that all the views in V' disclose different data from v. This is impossible because v has been obtained by a view $v' \in V'$ just by replacing the views with their definitions (and each view is semantically equivalent to its definition). □

C.3.2 A sound under-approximation of query determinacy

Before proving that *apprDet*, which is defined in Section 6.8.2, is a sound approximation of *determines*, we extend *determines* from sentences to formulae.

Given a system's configuration $M = \langle D, \Gamma \rangle$, a formula ϕ, a set of views V with owner's privileges, a set of tables T, and a well-formed assignment ν for ϕ, we say that V and T determine (ϕ, ν), denoted by $determines_M(T, V, \phi, \nu)$, iff $D, \Gamma \vdash Q \twoheadrightarrow \phi\nu$ holds, where Q is the set of queries containing all tables in T and all views in V. In the following, given a view $\langle u, o, q, m \rangle$, we denote by $def(\langle u, o, q, m \rangle)$ its definition q. Furthermore, given a set of tables T, a set of views V, and a database state db, we denote by $[\![db]\!]_{V,T}$ the set of all states $\{db' \in \Omega_D^\Gamma \mid \bigwedge_{t \in T} db(t) = db'(t) \wedge \bigwedge_{v \in V} [def(v)]^{db} = [def(v)]^{db'}\}$.

In Proposition C.4, we show that *apprDet* is, indeed, a sound under-approximation of query determinacy.

Proposition C.4. *Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ be an M-system state, $T' \subseteq D$ be a set of tables, $V' \subseteq V$ be a set of views with owner's privileges, and ϕ be a formula. If $apprDet(T', V', \phi, s, M) = \top$, then for all well-formed assignments ν for ϕ, $determines_M(T', V', \phi, \nu)$ holds.*

Proof. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ be an M-system state, $T' \subseteq D$ be a set of tables, $V' \subseteq V$ be a set of views with owner's privileges, and ϕ be a formula. We prove the lemma by structural induction over the formula ϕ.

Base Case: There are a number of alternatives.

$\phi := R(\overline{x})$ Assume that $apprDet(T, V, R(\overline{x}), s, M) = \top$. There are two cases:

1. $R \in T'$. In this case, the set T' trivially determines the formula $R(\overline{x})$ for any well-formed assignment ν. Therefore, $determines_M(T', V', R(\overline{x}), \nu)$ holds. Indeed, assume that this is not the case. Thus, there are three database states db, db_1, and db_2 such that $db_1, db_2 \in [\![db]\!]_{V',T'}$ and $[R(\overline{x})\nu]^{db_1} \neq [R(\overline{x})\nu]^{db_2}$. From this and the relational calculus semantics, it follows that $db_1(R) \neq db_2(R)$. From this, $R \in T'$, and $db_1, db_2 \in [\![db]\!]_{V',T'}$, it follows that $db_1(R) = db_2(R)$ leading to a contradiction.
2. there is a view v' in $extend(M, s, V')$ such that $def(v') = \{\overline{x} \mid R(\overline{x})\}$. This means that there is a sequences of views $V_1, \ldots, V_n$ in s such that $def(V_1) = \{\overline{x} \mid R(\overline{x})\}$, $def(V_2) = \{\overline{x} \mid V_1(\overline{x})\}$, $\ldots$, $def(V_n) = \{\overline{x} \mid V_{n-1}(\overline{x})\}$, and $V_n \in V'$. Therefore, the set V' trivially determines the formula $R(\overline{x})$ for any well-formed assignment ν, and V_n and R are equivalent. Therefore, $determines_M(T', V', R(\overline{x}), \nu)$ holds.

$\phi := V(\overline{x})$ Assume that $apprDet(T, V, V(\overline{x}), s, M) = \top$. There are two cases:

1. There is a view $\langle V, o, q, O \rangle \in V'$. In this case, the set V' trivially determines the formula $V(\overline{x})$ for any assignment ν that is well-formed for ϕ. Therefore, $determines_M(T', V', V(\overline{x}), \nu)$ holds.
2. there is a view v' in $extend(M, s, V')$ such that $def(v') = \{\overline{x} \mid V(\overline{x})\}$. This means that there is a sequences of views $V_1, \ldots, V_n$ in s such that $def(V_1) = \{\overline{x} \mid V(\overline{x})\}$, $def(V_2) = \{\overline{x} \mid V_1(\overline{x})\}$, $\ldots$, $def(V_n) = \{\overline{x} \mid V_{n-1}(\overline{x})\}$, and $V_n \in V'$. Therefore, the set V' trivially determines the formula $V(\overline{x})$ for any well-formed assignment ν, and V_n and V are equivalent. Therefore, $determines_M(T', V', V(\overline{x}), \nu)$ holds.

$\phi := x = v$ For any well-formed assignment ν, the empty set trivially determines the formula $x = v$.

$\phi := \top$ The proof of this case is similar to that of $\phi := x = v$.

$\phi := \bot$ The proof of this case is similar to that of $\phi := x = v$.

This concludes the proof of the base case.

Induction Step: Assume that the claim holds for all sub-formulae of ϕ. We now show that the claim holds also for ϕ. There are a number of cases:

$\boldsymbol{\phi := \psi \wedge \gamma}$ Assume that $apprDet(T', V', \psi \wedge \gamma, s, M) = \top$. There are two cases:

1. $apprDet(T', V', \psi, s, M) = \top$ and $apprDet(T', V', \gamma, s, M) = \top$. From the induction hypothesis, it follows that both $determines_M(T', V', \psi, \nu)$ and $determines_M(T', V', \gamma, \nu)$ hold for all well-formed assignments ν. Therefore, also $determines_M(T', V', \psi \wedge \gamma, \nu)$ holds for all well-formed assignments ν. Indeed, assume that this is not the case. Then, there are three database states db, db_1, and db_2 such that $db_1, db_2 \in [\![db]\!]_{V',T'}$ and $[(\psi \wedge \gamma)\nu]^{db_1} \neq [(\psi \wedge \gamma)\nu]^{db_2}$. From this and the relational calculus semantics, there are two cases:
   (a) $[\psi\nu]^{db_1} \neq [\psi\nu]^{db_2}$. From this, it follows that $determines_M(T', V', \psi, \nu)$ does not hold. This contradicts the fact that $determines_M(T', V', \psi, \nu)$ holds.
   (b) $[\gamma\nu]^{db_1} \neq [\gamma\nu]^{db_2}$. The proof of this case is similar to the previous one.
2. there is a view v' in $extend(M, s, V')$ such that $def(v') = \{\overline{x} \mid \psi \wedge \gamma\}$. From Proposition C.3, it follows that there is a view $v'' \in V'$ that is equivalent to v', and, therefore, to $\{\overline{x} \mid \psi \wedge \gamma\}$. Thus, $determines_M(T', V', \psi \wedge \gamma, \nu)$ holds for all assignments ν that are well-formed for ϕ.

$\boldsymbol{\phi := \psi \vee \gamma}$ This case is similar to $\psi \wedge \gamma$.

$\boldsymbol{\phi := \neg\psi}$ Assume that $apprDet(T', V', \neg\psi, s, M) = \top$. There are two cases:

1. $apprDet(T', V', \psi, s, M) = \top$. From the induction hypothesis, $determines_M(T', V', \psi, \nu)$ holds. Therefore, also $determines_M(T', V', \neg\psi, \nu)$ holds. Indeed, assume that this is not the case. This means that there are three database states db, db_1, and db_2 such that db_1, db_2 are in $[\![db]\!]_{V',T'}$ and $[\neg\psi\nu]^{db_1} \neq [\neg\psi\nu]^{db_2}$. From this and the relational calculus semantics, it follows that $[\psi\nu]^{db_1} \neq [\psi\nu]^{db_2}$. From this, it follows that $determines_M(T', V', \psi, \nu)$ does not hold. This contradicts the fact that $determines_M(T', V', \psi, \nu)$ holds.
2. there is a view v' in $extend(M, s, V')$ such that $def(v') = \{\overline{x} \mid \neg\psi\}$. From Proposition C.3, it follows that there is a view $v'' \in V'$ that is equivalent to v', and, therefore, to $\{\overline{x} \mid \neg\psi\}$. Thus, $determines_M(T', V', \neg\psi, \nu)$ holds for all well-formed assignments ν.

$\boldsymbol{\phi := \exists x.\, \psi}$ Assume that $apprDet(T', V', \exists x.\, \psi, s, M) = \top$. There are two cases:

1. $apprDet(T', V', \psi, s, M) = \top$. From the induction hypothesis, $determines_M(T', V', \psi, \nu)$ holds for all well-formed assignments ν. Therefore, also $determines_M(T', V', \exists x.\, \psi, \nu)$ holds for all well-formed assignments ν (note that any well-formed assignment for ψ is also a well-formed assignment for $\exists x.\, \psi$). Indeed, assume that this is not the case. This means that there are three database states db, db_1, and db_2 such that db_1, db_2 are in $[\![db]\!]_{V',T'}$ and $[(\exists x.\, \psi)\nu]^{db_1} \neq [(\exists x.\, \psi)\nu]^{db_2}$. From this and the relational calculus semantics, it follows that there is a value $v \in \mathbf{dom}$ such that $[\psi\nu[x \mapsto v]]^{db_1} \neq [\psi\nu[x \mapsto v]]^{db_2}$. Note that $\nu[x \mapsto v]$ is a well-formed assignment for ψ. Let's call this assignment ν'. From this, it follows that $[\psi\nu']^{db_1} \neq [\psi\nu']^{db_2}$. From this, it follows that $determines_M(T', V', \psi, \nu')$ does not hold. This contradicts the fact that $determines_M(T', V', \psi, \nu)$ holds for any well-formed assignment ν.
2. there is a view v' in $extend(M, s, V')$ such that $def(v') = \{\overline{x} \mid \exists x.\, \psi\}$. From Proposition C.3, it follows that there is a view $v'' \in V'$ that is equivalent to v', and, therefore, to $\{\overline{x} \mid \exists x.\, \psi\}$. Thus, $determines_M(T', V', \exists x.\, \psi, \nu)$ holds for all well-formed assignments ν.

$\boldsymbol{\phi := \forall x.\, \psi}$ This case is similar to $\exists x.\, \psi$.

This concludes the proof of the induction step.

This completes the proof. □

C.3.3 $\rightsquigarrow^{appr}_{auth}$ is a sound approximation of $\rightsquigarrow_{auth}$

We now show that $\rightsquigarrow^{appr}_{auth}$ is a sound approximation of $\rightsquigarrow_{auth}$. Namely, whenever $s \rightsquigarrow^{appr}_{auth} act$ holds, then $s \rightsquigarrow_{auth} act$ holds as well. A *derivation of* $s \rightsquigarrow^{appr}_{auth} act$ is a proof tree, obtained using the rules defining $\rightsquigarrow^{appr}_{auth}$, which ends in $s \rightsquigarrow^{appr}_{auth} act$. The size of a derivation is the number of $\rightsquigarrow^{appr}_{auth}$ rules that are used to show that $s \rightsquigarrow^{appr}_{auth} act$. In the following, we switch freely between statements of the form $s \rightsquigarrow^{appr}_{auth} act$ and their derivations. We denote the size of the derivation of $s \rightsquigarrow^{appr}_{auth} act$ as $|s \rightsquigarrow^{appr}_{auth} act|$.

Proposition C.5. *Let $M = \langle D, \Gamma \rangle$ be a system configuration, s be an M-state, c be an M-context, and $act \in \mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$ be an action or a trigger. If $s \rightsquigarrow^{appr}_{auth} act$, then $s \rightsquigarrow_{auth} act$.*

Proof. Let $M = \langle D, \Gamma \rangle$ be a system configuration, s be an M-state, c be an M-context, and $act \in \mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$. Furthermore, we assume that there is a derivation of $s \rightsquigarrow^{appr}_{auth} act$. We prove our claim by induction on the size of $s \rightsquigarrow^{appr}_{auth} act$'s derivation.

Base Case: We now show that, for all s and act such that $|s \rightsquigarrow^{appr}_{auth} act| = 1$, if $s \rightsquigarrow^{appr}_{auth} act$, then $s \rightsquigarrow_{auth} act$. Observe that the base case is trivial. Indeed, for the rules *INSERT DELETE admin*,

CREATE VIEW admin, *CREATE TRIGGER admin*, *SELECT*, *EXECUTE TRIGGER-3*, *GRANT-2*, *GRANT-5*, and *ADD USER*, if $s \rightsquigarrow^{appr}_{auth} act$, then $s \rightsquigarrow_{auth} act$ follows trivially from the rule's definition.

Induction Step: We now assume that, for all derivations of size less than $|s \rightsquigarrow^{appr}_{auth} act|$, it holds that if $s' \rightsquigarrow^{appr}_{auth} act'$, then $s' \rightsquigarrow_{auth} act'$. There are several cases:

1. Rule *INSERT DELETE*: Assume that $s \rightsquigarrow^{appr}_{auth} act$ holds and that $act = \langle u, \mathtt{op}', R, \bar{t}\rangle$, where op' is one of $\{\mathtt{INSERT}, \mathtt{DELETE}\}$. From the rule's definition, it follows that there is a grant $g = \langle op, u, \langle op', R\rangle, u'\rangle$ in $s.sec$ such that $s \rightsquigarrow^{appr}_{auth} g$. From this and the induction hypothesis, it follows that $s \rightsquigarrow_{auth} g$. Therefore, $s \rightsquigarrow_{auth} act$ holds (we can apply the *INSERT DELETE* rule in $\rightsquigarrow_{auth}$).
2. Rule *CREATE VIEW*: The proof is similar to the one for the *INSERT DELETE* rule.
3. Rule *CREATE TRIGGER*: The proof is similar to the one for the *INSERT DELETE* rule.
4. Rule *EXECUTE TRIGGER-2*: Assume that $s \rightsquigarrow^{appr}_{auth} act$ holds and that $act = \langle i, o, e, R, \phi, st, A\rangle$ such that $[\phi[\overline{x}^{|R|} \mapsto tpl(s)]]^{s.db} = \top$. From the rule's definition, it follows that both $s \rightsquigarrow^{appr}_{auth} action(st, ow, tpl(s))$ and $s \rightsquigarrow^{appr}_{auth} action(st, invoker(s), tpl(s))$ hold. From this and the induction hypothesis, both $s \rightsquigarrow_{auth} action(st, ow, tpl(s))$ and $s \rightsquigarrow_{auth} action(st, invoker(s), tpl(s))$ hold. From this and the *EXECUTE TRIGGER-2* rule in $\rightsquigarrow_{auth}$, it follows that also $s \rightsquigarrow_{auth} act$ holds.
5. Rule *EXECUTE TRIGGER-1*: The proof is similar to the one for the *EXECUTE TRIGGER-2* rule.
6. Rule *GRANT-1*: Assume that $s \rightsquigarrow^{appr}_{auth} act$ holds and that $act = \langle op, u, p, u'\rangle$, where $op \in \{\oplus, \oplus^*\}$. From the rule's definition, it follows that there is a grant $g = \langle \oplus^*, u', p, u''\rangle$ in $s.sec$ such that $s \rightsquigarrow^{appr}_{auth} g$. From this and the induction hypothesis, ti follows that $s \rightsquigarrow_{auth} g$. From this and the *GRANT-1* rule in $\rightsquigarrow_{auth}$, it follows that $s \rightsquigarrow_{auth} act$ holds.
7. Rule *GRANT-3*: Assume that $s \rightsquigarrow^{appr}_{auth} act$ holds and that $act = \langle op, u, p, o\rangle$, where $p = \langle \mathtt{SELECT}, v\rangle$, $v \in \mathcal{VIEW}^{owner}_{D}$, $op \in \{\oplus, \oplus^*\}$, and $o = owner(v)$ such that $o \neq admin$. Let T' be the set obtained through the aT function and V' be the set obtained through the aV function. From the rule's definition, it follows that $apprDet(T', V', def(v)) = \top$. From this and Proposition C.4, it follows that $determines_M(T', V', def(v))$ holds. We now show that for any $obj \in T' \cup V'$, $hasAccess(s', \{obj\}, o, \oplus^*)$ holds. There are four cases:
   (a) $o = admin$ *and* $obj \in D$. Since $obj \in T'$, it follows that there is a $g = \langle \oplus^*, o, \langle \mathtt{SELECT}, obj\rangle, u'\rangle$ such that $s \rightsquigarrow^{appr}_{auth} g$. From this and the induction hypothesis, it follows that $s \rightsquigarrow_{auth} g$. Therefore, $hasAccess(s', \{obj\}, o, \oplus^*)$ holds.
   (b) $o \neq admin$ *and* $obj \in D$. Since $obj \in T'$, it follows that there is a $g = \langle \oplus^*, o, \langle \mathtt{SELECT}, obj\rangle, u'\rangle$ in sec such that $s \rightsquigarrow^{appr}_{auth} g$. From this and the induction hypothesis, it follows that $s \rightsquigarrow_{auth} g$. Thus, $hasAccess(s', \{obj\}, o, \oplus^*)$ holds.
   (c) $o = admin$ *and* $obj \in V$. The proof of this case is similar to that of $o = admin$ and $obj \in D$.
   (d) $o \neq admin$ *and* $obj \in V$. The proof of this case is similar to that of $o \neq admin$ and $obj \in D$.

   Note that from $hasAccess(s', A, o, op)$ and $hasAccess(s', B, o, op)$, it follows that $hasAccess(s', A \cup B, o, op)$. Thus, $hasAccess(s, T' \cup V', o, \oplus^*)$ holds. From this, it follows that $s \rightsquigarrow_{auth} act$ holds because we can apply the corresponding rule in $\rightsquigarrow_{auth}$.
8. Rule *GRANT-4*: The proof is similar to the one for the *GRANT-3* rule.
9. Rule *REVOKE*: Assume that $s \rightsquigarrow^{appr}_{auth} act$ holds and that $act = \langle \ominus, u, p, u'\rangle$. From the rule's definition, it follows that $s' \rightsquigarrow^{appr}_{auth} g$ for any $g \in s'.sec$, where $s' = applyRev(s, \langle \ominus, u, p, u'\rangle)$. From the induction's hypothesis, it follows that $s' \rightsquigarrow_{auth} g$ for any $g \in s'.sec$. Therefore, we can apply the rule *REVOKE* of $\rightsquigarrow_{auth}$ to derive $s \rightsquigarrow_{auth} act$.

This completes our proof. □

C.3.4 f provides Database Integrity

We are now ready to prove that f provides database integrity.

Lemma C.1. *For any two states $s = \langle db, U, sec, T, V, c\rangle$, $s' = \langle db', U, sec, T, V, c'\rangle$ in Ω_M and any action $a \in \mathcal{A}_{D,\mathcal{U}}$:*

1. *$s \rightsquigarrow_{auth} a$ iff $s' \rightsquigarrow_{auth} a$, and*
2. *$s \rightsquigarrow^{appr}_{auth} a$ iff $s' \rightsquigarrow^{appr}_{auth} a$.*

Proof. It is easy to see that the only rules that depends on db, db', c, and c' are `EXECUTE TRIGGER - 1`, `EXECUTE TRIGGER - 2`, and `EXECUTE TRIGGER - 3`. Since they are not used to evaluate whether $s \rightsquigarrow_{auth} a$ and $s \rightsquigarrow^{appr}_{auth} a$ hold for actions in $\mathcal{A}_{D,\mathcal{U}}$, the lemma follows trivially. □

Lemma C.2. *Let M be a system configuration. Then, for all M-states $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ such that $trigger(s) = \epsilon$ and all actions $act \in \mathcal{A}_{D,\mathcal{U}}$, if $f_{int}(s, act) = \top$, then $s \rightsquigarrow_{auth} act$.*

Proof. We prove the theorem by contradiction. Assume, for contradiction's sake, that the claim does not hold. Therefore, there is a state s and an action act such that $f_{int}(s, act) = \top$, $trigger(s) = \epsilon$, and $s \not\rightsquigarrow_{auth} act$. Thus, from $f_{int}(s, act) = \top$, $trigger(s) = \epsilon$, and f_{int}'s definition, it follows $s \rightsquigarrow^{appr}_{auth} act$. From this and Proposition C.5, it follows that $s \rightsquigarrow_{auth} act$ leading to a contradiction. □

Lemma C.3. *Let M be a system configuration. For all M-states $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and all triggers $t \in T$ such that $trigger(s) = t$, the following hold:*

1. *If $f_{int}(s, c) = \top$ and $[\psi]^{db} = \bot$, then $s \rightsquigarrow_{auth} t$, where $c = trigCond(s) = \langle u, \texttt{SELECT}, \psi\rangle$.*
2. *If $f_{int}(s, c) = \top$, $[\psi]^{db} = \top$, and $f_{int}(s, a) = \top$, then $s \rightsquigarrow_{auth} t$, where $c = trigCond(s) = \langle u, \texttt{SELECT}, \psi\rangle$ and $a = trigAct(s)$.*

Proof. We prove both claims by contradiction.

Assume, for contradiction's sake, that the first claim does not hold. Therefore, there is a state s and a trigger t such that $f_{int}(s, c) = \top$ and $[\psi]^{db} = \bot$ and $s \not\rightsquigarrow_{auth} t$. From $[\psi]^{db} = \bot$, $trigger(s) = t$, and the rule `EXECUTE TRIGGER - 3`, it follows that $s \rightsquigarrow_{auth} t$ holds, which leads to a contradiction.

Assume, for contradiction's sake, that the second claim does not hold. Therefore, there is a state s and a trigger $t = \langle id, ow, e, R, \phi, st, m\rangle$ such that $f_{int}(s, c) = \top$, $[\psi]^{db} = \top$, $f_{int}(s, a) = \top$, and $s \not\rightsquigarrow_{auth} t$. If t is a trigger with owner's privileges, from $f_{int}(s, a) = \top$ and $trigger(s) = t$, it follows that $s \rightsquigarrow^{appr}_{auth} action(st, ow, tpl(s))$. From this and Proposition C.5, it follows that $s \rightsquigarrow_{auth} action(st, ow, tpl(s))$. From this, $t \in T$, $[\psi]^{db} = \top$, and $trigCond(s) = \langle u, \texttt{SELECT}, \psi\rangle$, it follows that we can apply the rule `EXECUTE TRIGGER - 1` and we can derive $s \rightsquigarrow_{auth} t$. If t is a trigger with activator's privileges, from $f_{int}(s, a) = \top$ and $trigger(s) = t$, it follows that $s \rightsquigarrow^{appr}_{auth} action(st, ow, tpl(s))$ and $s \rightsquigarrow^{appr}_{auth} action(st, invoker(s), tpl(s))$. From this Proposition C.5, it follows that $s \rightsquigarrow_{auth} action(st, ow, tpl(s))$ and $s \rightsquigarrow_{auth} action(st, invoker(s), tpl(s))$. From this, $t \in T$, $[\psi]^{db} = \top$, and $trigCond(s) = \langle u, \texttt{SELECT}, \psi\rangle$, it follows that we can apply the rule `EXECUTE TRIGGER - 2` and we can derive $s \rightsquigarrow_{auth} t$. In both cases we can derive $s \rightsquigarrow_{auth} t$, leading to a contradiction. □

We are now ready to prove our main result, namely that f provides database integrity.

Theorem C.1. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, f be the PDP defined in Section 6.8, and $P = \langle M, f\rangle$ be an extended configuration. The PDP f provides database integrity with respect to P.*

Proof. We prove the lemma by contradiction. Assume, for contradiction's sake, that f does not satisfy the database integrity property. There are three cases:

- there is a reachable state s and an action $act \in \mathcal{A}_{D,\mathcal{U}}$ such that $trigger(s) = \epsilon$, $f(s, act) = \top$, and $s \not\rightsquigarrow_{auth} act$. From $f(s, act) = \top$, it follows that $f_{int}(s, act) = \top$. From this, $trigger(s) = \epsilon$, and Lemma C.2, it follows $s \rightsquigarrow_{auth} act$, which leads to a contradiction.
- there is a reachable state s and a trigger $t \in \mathcal{TRIGGER}_D$ such that $trigger(s) = t$, $f(s, c) = \top$, $[\psi]^{s.db} = \bot$, and $s \not\rightsquigarrow_{auth} t$, where $c = \langle u, \texttt{SELECT}, \psi\rangle$ is t's condition. From $f(s, c) = \top$, it follows that $f_{int}(s, c) = \top$. From $f_{int}(s, c) = \top$, $[\psi]^{s.db} = \bot$, $trigger(s) = t$, and Lemma C.3, it follows $s \rightsquigarrow_{auth} t$, which leads to a contradiction.
- there is a reachable state s and a trigger $t \in \mathcal{TRIGGER}_D$ such that $trigger(s) = t$, $f(s, c) = \top$, $[\psi]^{s.db} = \top$, $f(s', a) = \top$, and $s \not\rightsquigarrow_{auth} t$, where $c = \langle u, \texttt{SELECT}, \psi\rangle$ is t's condition, a is t's action, and s' is the state obtained from s by updating the context's history. From $f(s', a) = \top$, it follows that $f_{int}(s', a) = \top$. Since s and s' are equivalent modulo the context's history and f_{int} does not depend on the context's history, it follows that $f_{int}(s, a) = \top$. From $f_{int}(s, c) = \top$, $[\psi]^{s.db} = \top$, $f_{int}(s, a) = \top$, $trigger(s) = t$, and Lemma C.3, it follows $s \rightsquigarrow_{auth} t$, which leads to a contradiction.

This completes the proof. □

We also prove that, by using f, any reachable state has a consistent policy. Observe that this is the underlying reason why f prevents Attacks 6.2 and 6.3.

Lemma C.4. *Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is the PDP defined in Section 6.8, and L be the P-LTS. For each reachable state $s = \langle db, U, sec, T, V, c\rangle$, $s \rightsquigarrow_{auth} g$ for all $g \in sec$.*

Proof. We claim that, for any run r, the state $last(r)$ is such that for all $p \in last(r).sec$, $last(r) \rightsquigarrow_{auth} p$. From this, the lemma follows trivially.

We now prove that for any run r, the state $last(r)$ is such that for all $p \in last(r).sec$, $last(r) \rightsquigarrow_{auth} p$. We do this by induction on the length of the run r.

Base Case: The base case consists of the runs containing only one initial state. Note that an initial state contains only grants issued by *admin*, together with views and triggers owned by *admin*. It is easy to see that for any permission $p = \langle op, u, pr, admin\rangle$ in a policy sec in an initial state s, it holds that $s \rightsquigarrow_{auth} p$. There are two cases:

1. The privilege pr in p is such that $pr \in \mathcal{PRIV}_D \setminus \mathcal{PRIV}_D^{\texttt{SELECT},\mathcal{VIEW}_D^{owner}}$. Then, $s \rightsquigarrow_{auth} p$ by the rule *GRANT-2*.
2. The privilege pr in p is such that pr is in the set $\mathcal{PRIV}_D^{\texttt{SELECT},\mathcal{VIEW}_D^{owner}}$. Recall that *admin* is the owner of all views in the state. Then, $s \rightsquigarrow_{auth} p$ by the rule *GRANT-3*. Indeed, *admin* can read (and delegate the `SELECT` permission over) all tables in the database. Therefore, $hasAccess(s, D, admin, \oplus^*)$ and $determines_M(D, \emptyset, q)$ hold for any query q.

This complete the proof for the base case.

Induction Step: We now assume that for all runs r' of length less than the length of r, the state $last(r')$ is such that for all $p \in last(r').sec$, $last(r') \rightsquigarrow_{auth} p$. Let r' be the run $r^{|r|-1}$. There are two cases, depending on whether act raises an exception or not.

1. $secEx(last(r)) = \bot$ and $Ex(last(r)) = \emptyset$. There are a number of cases depending on act:
   (a) act is $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$, $\langle u, \texttt{DELETE}, R, \bar{t}\rangle$, $\langle u, \texttt{SELECT}, q\rangle$, $\langle u, \texttt{ADD\_USER}, u'\rangle$, or $\langle u, \texttt{CREATE}, o\rangle$. In these cases, $last(r').sec = last(r).sec$. Furthermore, $last(r').U \subseteq last(r).U$, $last(r').T \subseteq last(r).T$, and $last(r').V \subseteq last(r).V$. From this and the fact that $last(r') \rightsquigarrow_{auth} g$ for all $g \in last(r').sec$, it follows that $last(r) \rightsquigarrow_{auth} g$ for all $g \in last(r).sec$.
   (b) act is $\langle op, u, p, u'\rangle$, where $op \in \{\oplus, \oplus^*\}$. From $secEx(last(r)) = \bot$, it follows that $last(r') \rightsquigarrow_{auth} act$. From the induction hypothesis, it follows that $last(r') \rightsquigarrow_{auth} g$ for all $g \in last(r').sec$. We claim that, for any grant statement g, if $\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} g$, then $\langle db', U, sec', T, V, c'\rangle \rightsquigarrow_{auth} g$ for any policy such that $sec \subseteq sec'$. From the claim, it follows that $last(r) \rightsquigarrow_{auth} act$ and $last(r) \rightsquigarrow_{auth} g$ for all $g \in last(r').sec$. From this and $last(r).sec = \{act\} \cup last(r').sec$, it follows that $last(r) \rightsquigarrow_{auth} g$ for all $g \in last(r).sec$.
   Our claim that, for any grant statement g, if $\langle db, U, sec, T, V, c\rangle \rightsquigarrow_{auth} g$, then $\langle db', U, sec', T, V, c'\rangle \rightsquigarrow_{auth} g$, where $sec \subseteq sec'$, follows trivially from the definition of the rules for `GRANT` statements.
   (c) act is $\langle \ominus, u, p, u'\rangle$. From $secEx(last(r)) = \bot$, it follows that $last(r') \rightsquigarrow_{auth} act$. From this, it follows that $s' \rightsquigarrow_{auth}^{appr} g$ for all $g \in s'.sec$, where $s' = applyRev(last(r'), act)$. From this and Proposition C.5, it follows that $s' \rightsquigarrow_{auth} g$ for all $g \in s'.sec$. Recall that $last(r)$ and s' are equivalent modulo the database and the context. From this, Lemma C.1, and $s' \rightsquigarrow_{auth} g$ for all $g \in s'.sec$, it follows that $last(r) \rightsquigarrow_{auth} g$ for all $g \in last(r).sec$.
   (d) act is a trigger and the `WHEN` condition is not satisfied. In this case, $last(r')$ and $last(r)$ are equivalent modulo the context. From this, the induction hypothesis, and Lemma C.1, it follows that $last(r) \rightsquigarrow_{auth} g$ for all $g \in last(r).sec$.
   (e) act is a trigger and the `WHEN` condition is satisfied. In this case, the proof is the same as the previous cases depending on the trigger's action.
2. $secEx(last(r)) = \top$ or $Ex(last(r)) \neq \emptyset$. From this and the LTS's rules, it follows that there is a state $s' \in \{last(r^i) \mid 1 \leq i \leq |r| - 1\}$ such that $sysState(last(r)) = sysState(s')$ (because there has been a roll-back). Let sec be the policy in s'. From the induction hypothesis, it follows that for all $p \in sec$, $s' \rightsquigarrow_{auth} p$. From this fact, the $\rightsquigarrow_{auth}$'s definition, $sysState(last(r)) = sysState(s')$, and Lemma C.1, it follows that for all $p \in last(r).sec$, also $last(r) \rightsquigarrow_{auth} p$.

This complete the proof for the induction step.

This completes the proof. □

C.4 Data Confidentiality Proofs

Here we prove that our PDP f provides data confidentiality.

C.4.1 A sound under-approximation of query containment

Proposition C.6 proves that the rules in Figure 6.27 are a sound under-approximation of query containment.

Proposition C.6. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, and $\phi(\overline{x})$ and $\psi(\overline{y})$ be two formulae. If $\phi \subseteq_M \psi$, according to the rules in Figure 6.27, then $\forall d \in \Omega_D^\Gamma. [\{\overline{x} \mid \phi\}]^d \subseteq [\{\overline{y} \mid \psi\}]^d$, where $\overline{x}$ (respectively $\overline{y}$) is the tuple defined by the variables in free(ϕ) (respectively free(ψ)) ordered according to $\preceq_{var}$.*

Proof. Observe that $\phi \subseteq_M \psi$ iff there is a finite derivation that ends in $\phi \subseteq_M \psi$ created using the rules in Figure 6.27. We prove our claim by induction on the derivation's length.

Base Case Assume that the derivation has length 1. There are four cases depending on the rule used to derive $\phi \subseteq_M \psi$:

1. Rule *Conjunction*. From the rule's definition, it follows that $free(\phi) = free(\phi \wedge \psi)$. Let $d \in \Omega_D^\Gamma$ and $\bar{t} \in [\{\overline{x} \mid \phi \wedge \psi\}]^d$. From $\bar{t} \in [\{\overline{x} \mid \phi \wedge \psi\}]^d$ and the definition of non-boolean query, it

follows that $[(\phi \wedge \psi)[\overline{x} \mapsto \overline{t}]]^d = \top$. From this, it follows that $[\phi[\overline{x} \mapsto \overline{t}]]^d = \top$. From this and the definition of non-boolean query, $\overline{t} \in [\{\overline{x} \mid \phi\}]^d$. Therefore, $[\{\overline{x} \mid \phi \wedge \psi\}]^d \subseteq [\{\overline{x} \mid \phi\}]^d$.

2. Rule *Disjunction*. From the rule's definition, it follows that $\mathit{free}(\phi) = \mathit{free}(\phi \vee \psi) = \overline{x}$. Let $d \in \Omega_D^\Gamma$ and $\overline{t} \in [\{\overline{x} \mid \phi\}]^d$. From $\overline{t} \in [\{\overline{x} \mid \phi\}]^d$ and the definition of non-boolean query, it follows that $[\phi[\overline{x} \mapsto \overline{t}]]^d = \top$. From this, it follows that $[(\phi \vee \psi)[\overline{x} \mapsto \overline{t}]]^d = \top$. From this and the definition of non-boolean query, $\overline{t} \in [\{\overline{x} \mid \phi \vee \psi\}]^d$. Therefore, $[\{\overline{x} \mid \phi\}]^d \subseteq [\{\overline{x} \mid \phi \vee \psi\}]^d$.
3. Rule *Identity*. From the rule's definition, it follows that $\mathit{free}(\phi) = \overline{x}$, $\mathit{free}(\psi) = \overline{y}$, and $\phi[\overline{x} \mapsto \overline{y}] = \psi$. Let $d \in \Omega_D^\Gamma$ and $\overline{t} \in [\{\overline{x} \mid \phi\}]^d$. From $\overline{t} \in [\{\overline{x} \mid \phi\}]^d$ and the definition of non-boolean query, it follows that $[\phi[\overline{x} \mapsto \overline{t}]]^d = \top$. From this and $\phi[\overline{x} \mapsto \overline{y}] = \psi$, it follows that $[\psi[\overline{y} \mapsto \overline{t}]]^d = \top$. From this and the definition of non-boolean query, $\overline{t} \in [\{\overline{y} \mid \psi\}]^d$. Therefore, $[\{\overline{x} \mid \phi\}]^d \subseteq [\{\overline{y} \mid \psi\}]^d$.
4. Rule *Inclusion Dependency*. From the rule's definition, it follows that $\gamma := \forall \overline{x}, \overline{z}.\, (R(\overline{x}, \overline{z}) \rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w}))$ is in Γ. Let $d \in \Omega_D^\Gamma$ and $\overline{t} \in [\{\overline{x} \mid \exists \overline{z}.\, R(\overline{x}, \overline{z})\}]^d$. From $\overline{t} \in [\{\overline{x} \mid \exists \overline{z}.\, R(\overline{x}, \overline{z})\}]^d$ and the definition of non-boolean query, it follows that $[\exists \overline{z}.\, R(\overline{t}, \overline{z})]^d = \top$. Therefore, there is a tuple $(\overline{t}, \overline{w}) \in d(R)$. From this and $\gamma \in \Gamma$, it follows that there is a tuple $(\overline{t}, \overline{w}') \in d(S)$. From this, it follows that $[\exists \overline{w}.\, S(\overline{t}, \overline{w})]^d = \top$. From this and the definition of non-boolean query, it follows that $\overline{t} \in [\{\overline{x} \mid \exists \overline{w}.\, S(\overline{x}, \overline{w})\}]^d$. Therefore, it follows that $[\{\overline{x} \mid \exists \overline{z}.\, R(\overline{x}, \overline{z})\}]^d \subseteq [\{\overline{x} \mid \exists \overline{w}.\, S(\overline{x}, \overline{w})\}]^d$ holds.

This completes the proof for the base case.

Induction Step Assume now that the claim holds for all derivations of length less than that of $\phi \subseteq_M \psi$. We now prove that it holds also for $\phi \subseteq_M \psi$. There are two cases:

1. Rule *Projection*. In this case, $\phi \subseteq_M \psi$ is of the form $\exists x_i.\, \alpha \subseteq_M \exists y_i.\, \beta$ and it is obtained by applying the rule *Projection* to $\alpha \subseteq_M \beta$. From the rule, it follows that $\alpha \subseteq_M \beta$ holds. Let $1 \leq u \leq n$ and $\overline{t}'$ (respectively $\overline{x}'$ and $\overline{y}'$) be the tuple obtained from $\overline{t}$ (respectively $\overline{x}$ and $\overline{y}$) by dropping the i-th value (respectively variable). We now prove that $[\{\overline{x}' \mid \exists x_i.\, \alpha\}]^d \subseteq [\{\overline{y}' \mid \exists y_i.\, \beta\}]^d$. Assume, for contradiction's sake, that this is not the case, namely there is a tuple $\overline{v}$ such that $\overline{v} \in [\{\overline{x}' \mid \exists x_i.\, \alpha\}]^d$ but $\overline{v} \notin [\{\overline{y}' \mid \exists y_i.\, \beta\}]^d$. From $\overline{v} \in [\{\overline{x}' \mid \exists x_i.\, \alpha\}]^d$ and the relational calculus semantics, it follows that there is a tuple $\overline{v}_1$, obtained by adding a value to $\overline{v}$ in the i-th position, such that $\overline{v}_1 \in [\{\overline{x} \mid \alpha\}]^d$. From this, $\alpha \subseteq_M \beta$, and the induction hypothesis, it follows that $\overline{v}_1 \in [\{\overline{y} \mid \beta\}]^d$. From this and the relational calculus semantics, it follows that $\overline{v} \in [\{\overline{y}' \mid \exists y_i.\, \beta\}]^d$. This contradicts the fact that $\overline{v} \notin [\{\overline{y}' \mid \exists y_i.\, \beta\}]^d$.
2. Rule *Transitivity*. From the rule, it follows that both $\phi \subseteq_M \gamma$ and $\gamma \subseteq_M \psi$ hold. Let $d \in \Omega_D^\Gamma$ and $\overline{t} \in [\{\overline{x} \mid \phi\}]^d$. From $\phi \subseteq_M \gamma$, $\overline{t} \in [\{\overline{x} \mid \phi\}]^d$, and the induction hypothesis, it follows that $\overline{t} \in [\{\overline{y} \mid \gamma\}]^d$. From this, $\gamma \subseteq_M \psi$, and the induction hypothesis, it follows that $\overline{t} \in [\{\overline{z} \mid \psi\}]^d$. Hence, $\phi \subseteq_M \psi$.

This completes the proof. □

C.4.2 Data security is a sound under-approximation of judgment's security

Here we introduce data security, a weaker notion than judgment security. Afterwards, we show that it is a sound under-approximation of judgment's security.

Let $P = \langle M, f \rangle$ be an extended configuration, L be the P-LTS, $u \in \mathcal{U}$ be a user, $r \in \mathit{traces}(L)$ be an L-run, $\phi \in RC_{bool}$ be a sentence, and $1 \leq i \leq |r|$. Furthermore, let s be the i-th state of r. The judgment $r, i \vdash_u \phi$ is *data-secure for M and u*, denoted by $\mathit{secure}^{data}_{P,u}(r, i \vdash_u \phi)$, iff for all s', $s'' \in [\![\mathit{sysState}(s)]\!]^{data}_{M,u}$, $[\phi]^{s'.db} = [\phi]^{s''.db}$, where $\cong^{data}_{M,u}$ is the data-indistinguishability relation defined in Chapter 6 and $[\![s]\!]^{data}_{M,u}$ is the set $\{s' \in \Pi_M | s \cong^{data}_{M,u} s'\}$.

Proposition C.7 states that $\mathit{secure}^{data}_{P,u}$ is a sound under approximation of $\mathit{secure}_{P,u}$. Observe that deciding whether $\mathit{secure}^{data}_{P,u}(r, i \vdash_u \phi)$ holds for a given judgment is still undecidable for the relational calculus (this directly follows from our results in Chapter 3).

Proposition C.7. *Let $P = \langle M, f \rangle$ be an extended configuration, L be the P-LTS, $u \in \mathcal{U}$ be a user, $r \in \mathit{traces}(L)$ be an L-run, $\phi \in RC_{bool}$ is a sentence, and $1 \leq i \leq |r|$. Given a judgment $r, i \vdash_u \phi$, if $\mathit{secure}^{data}_{P,u}(r, i \vdash_u \phi)$, then $\mathit{secure}_{P,u}(r, i \vdash_u \phi)$.*

Proof. We prove the claim by contradiction. Let $P = \langle M, f \rangle$ be an extended configuration, L be the P-LTS, $u \in \mathcal{U}$ be a user, $r \in \mathit{traces}(L)$ be an L-run, $\phi \in RC_{bool}$ is a sentence, and $1 \leq i \leq |r|$. Furthermore, let $s = \langle db, U, sec, T, V, c \rangle$ be the i-th state of r. Assume, for contradiction's sake, that $\mathit{secure}^{data}_{P,u}(r, i \vdash_u \phi)$ holds and $\mathit{secure}_{P,u}(r, i \vdash_u \phi)$ does not hold. We denote, for brevity's sake, the fact that $\mathit{secure}_{P,u}(r, i \vdash_u \phi)$ does not hold as $\neg \mathit{secure}_{P,u}(r, i \vdash_u \phi)$. From $\neg \mathit{secure}_{P,u}(r, i \vdash_u \phi)$, it follows that there is a run $r' \in \mathit{traces}(L)$, whose last state is $s' = \langle db', U', sec', T', V', c' \rangle$, such that $r^i \cong_{P,u} r'$ and $[\phi]^{db} \neq [\phi]^{db'}$. From the (P, u)-indistinguishability definition, it follows that $\mathit{sysState}(\mathit{last}(r^i))$ and $\mathit{sysState}(\mathit{last}(r'))$ are data indistinguishable according to M and u, i.e., $\mathit{sysState}(\mathit{last}(r^i)) \cong^{data}_{M,u} \mathit{sysState}(\mathit{last}(r'))$. From $\mathit{secure}^{data}_{P,u}(r, i \vdash_u \phi)$, it also follows that for all

$s', s'' \in [\![sysState(s)]\!]^{data}_{M,u}$, $[\phi]^{s'.db} = [\phi]^{s''.db}$. From this and the fact that $sysState(last(r^i)) \cong^{data}_{M,u} sysState(last(r'))$, it follows that $[\phi]^{db} = [\phi]^{db'}$, which contradicts $[\phi]^{db} \neq [\phi]^{db'}$. This completes the proof. □

C.4.3 Properties of $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$

We now show that the rewritings $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ provide the desired properties. First, in Lemma C.5 we prove that the two rewritings satisfy the following invariants: "if $\phi^{\top}_{s,u}$ holds in s, then also ϕ holds in s" and "if $\phi^{\bot}_{s,u}$ does not hold in s, then also ϕ does not hold in s". Afterwards, in Lemma C.6 we show that both $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ are secure. Then, in Lemma C.8 we prove that $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ are equivalent to $\phi^{\top}_{s',u}$ and $\phi^{\bot}_{s',u}$ for any two data indistinguishable M-state s and s'. Finally, in Proposition C.8 we show that both $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$ are domain-independent.

Lemma C.5. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be a partial M-state, $u \in U$ be a user, and ϕ be a D-formula. For all assignments ν that are well-formed for ϕ, the following conditions hold:*

- *if $[\phi^{\top}_{s,u}\nu]^{db} = \top$, then $[\phi\nu]^{db} = \top$, and*
- *if $[\phi^{\bot}_{s,u}\nu]^{db} = \bot$, then $[\phi\nu]^{db} = \bot$.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be a partial M-state, $u \in U$ be a user, and ϕ be a D-formula. Furthermore, let ν be an assignment that is well-formed for ϕ. We prove our claim by induction on the structure of the formula ϕ.

Base Case There are four cases:

1. $\phi := x = y$. In this case, $\phi^{\top}_{s,u} = \phi^{\bot}_{s,u} = \phi$. From this, it follows that $[(x = v)^{\top}_{s,u}\nu]^{db} = [(x = v)\nu]^{db}$ and $[(x = v)^{\bot}_{s,u}\nu]^{db} = [(x = v)\nu]^{db}$. Therefore, our claim follows trivially.
2. $\phi := \top$. The proof of this case is similar to that of $\phi := x = y$.
3. $\phi := \bot$. The proof of this case is similar to that of $\phi := x = y$.
4. $\phi := R(\overline{x})$. Let $\overline{t}$ be the tuple $\nu(\overline{x})$. Note that since ν is well-formed for ϕ, $\overline{t}$ is well-defined.
   Assume that $[\phi^{\top}_{s,u}\nu]^{db} = \top$. From this and $\phi^{\top}_{s,u} := \bigvee_{S \in R^{\top}_{s,u}} S(\overline{x})$, it follows that there is an $S \in R^{\top}_{s,u}$ such that $\overline{t} \in db(S)$. Since $S \in R^{\top}_{s,u}$, it follows that $S \subseteq_M R$. From $S \subseteq_M R$, $\overline{t} \in db(S)$, and Proposition C.6, it follows that $\overline{t} \in db(R)$. From this, it follows that $[\phi\nu]^{db} = \top$.
   Assume that $[\phi^{\bot}_{s,u}\nu]^{db} = \bot$. From this and $\phi^{\bot}_{s,u} := \bigwedge_{S \in R^{\bot}_{s,u}} S(\overline{x})$, it follows that there is an $S \in R^{\bot}_{s,u}$ such that $\overline{t} \notin db(S)$. Since $S \in R^{\bot}_{s,u}$, it follows that $R \subseteq_M S$. From $R \subseteq_M S$, $\overline{t} \notin db(S)$, and Proposition C.6, it follows that $\overline{t} \notin db(R)$. From this, it follows that $[\phi\nu]^{db} = \bot$.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all sub-formulae of ϕ. We now show that our claim holds also for ϕ. There are a number of cases depending on ϕ's structure.

1. $\phi := \psi \wedge \gamma$. Assume that $[\phi^{\top}_{s,u}\nu]^{db} = \top$. From this and $\phi^{\top}_{s,u} := \psi^{\top}_{s,u} \wedge \gamma^{\top}_{s,u}$, it follows that $[\psi^{\top}_{s,u}\nu]^{db} = \top$ and $[\gamma^{\top}_{s,u}\nu]^{db} = \top$. Since ν is well-formed for ϕ, it is also well-formed for ψ and γ because $free(\psi) \subseteq free(\phi)$ and $free(\gamma) \subseteq free(\phi)$. From $[\psi^{\top}_{s,u}\nu]^{db} = \top$ and the induction hypothesis, it follows that $[\psi\nu]^{db} = \top$. From $[\gamma^{\top}_{s,u}\nu]^{db} = \top$ and the induction hypothesis, it follows that $[\gamma\nu]^{db} = \top$. From $[\psi\nu]^{db} = \top$, $[\gamma\nu]^{db} = \top$, $\phi := \psi \wedge \gamma$, and the relational calculus semantics, it follows that $[\phi\nu]^{db} = \top$.
   Assume that $[\phi^{\bot}_{s,u}\nu]^{db} = \bot$. From this and $\phi^{\bot}_{s,u} := \psi^{\bot}_{s,u} \wedge \gamma^{\bot}_{s,u}$, there are two cases:
   (a) $[\psi^{\bot}_{s,u}\nu]^{db} = \bot$. From $[\psi^{\bot}_{s,u}\nu]^{db} = \bot$ and the induction hypothesis, it follows that $[\psi\nu]^{db} = \bot$. From this and $\phi := \psi \wedge \gamma$, it follows that $[\phi\nu]^{db} = \bot$
   (b) $[\gamma^{\bot}_{s,u}\nu]^{db} = \bot$. From $[\gamma^{\bot}_{s,u}\nu]^{db} = \bot$ and the induction hypothesis, it follows that $[\gamma\nu]^{db} = \bot$. From this and $\phi := \psi \wedge \gamma$, it follows that $[\phi\nu]^{db} = \bot$
2. $\phi := \psi \vee \gamma$. The proof of this case is similar to that of $\phi := \psi \wedge \gamma$.
3. $\phi := \neg\psi$. Assume that $[\phi^{\top}_{s,u}\nu]^{db} = \top$. From this and $\phi^{\top}_{s,u} := \neg\psi^{\bot}_{s,u}$, it follows that $[\psi^{\bot}_{s,u}\nu]^{db} = \bot$. From this and the induction hypothesis, it follows that $[\psi\nu]^{db} = \bot$. From this, $\phi := \neg\psi$, and the relational calculus semantics, it follows that $[\phi\nu]^{db} = \top$.
   Assume that $[\phi^{\bot}_{s,u}\nu]^{db} = \bot$. From this and $\phi^{\bot}_{s,u} := \neg\psi^{\top}_{s,u}$, it follows that $[\psi^{\top}_{s,u}\nu]^{db} = \top$. From this and the induction hypothesis, it follows that $[\psi\nu]^{db} = \top$. From this and $\phi := \neg\psi$, it follows that $[\phi\nu]^{db} = \bot$.
4. $\phi := \exists x.\, \psi$. Assume that $[\phi^{\top}_{s,u}\nu]^{db} = \top$. From this and $\phi^{\top}_{s,u} := \exists x.\, \psi^{\top}_{s,u}$, it follows that there is a $v \in \mathbf{dom}$ such that $[\psi^{\top}_{s,u}\nu[x \mapsto v]]^{db} = \top$. Note that since v is well-formed for ϕ, $\nu[x \mapsto v]$ is well-formed for ψ because $\phi := \exists x.\, \psi$. From this, $[\psi^{\top}_{s,u}\nu[x \mapsto v]]^{db} = \top$, and the induction hypothesis, it follows that $[\psi\nu[x \mapsto v]]^{db} = \top$. From this and $\phi := \exists x.\, \psi$, it follows that $[\phi\nu]^{db} = \top$.
   Assume that $[\phi^{\bot}_{s,u}\nu]^{db} = \bot$. From this and $\phi^{\bot}_{s,u} := \exists x.\, \psi^{\bot}_{s,u}$, it follows that for all $v \in \mathbf{dom}$, $[\psi^{\bot}_{s,u}\nu[x \mapsto v]]^{db} = \bot$. Note that since v is well-formed for ϕ, $\nu[x \mapsto v]$ is well-formed for ψ

because $\phi := \exists x.\, \psi$. From this, $[\psi^{\bot}_{s,u}\nu[x \mapsto v]]^{db} = \bot$, and the induction hypothesis, it follows that for all $v \in \mathbf{dom}$, $[\psi\nu[x \mapsto v]]^{db} = \bot$. From this and $\phi := \exists x.\, \psi$, it follows that $[\phi\nu]^{db} = \bot$.

5. $\phi := \forall x.\, \psi$. The proof of this case is similar to that of $\phi := \exists x.\, \psi$.

This completes the proof of the induction step.

This completes the proof of our claim. □

In Lemma C.6, we prove that our rewritings are secure.

Lemma C.6. *Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, $r \in traces(L)$ be a run, ϕ be a RC-formula, and $1 \leq i \leq r$. Furthermore, let s be the i-th state of r. For all assignments ν over **dom** that are well-formed for ϕ, $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u}\nu)$, $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u}\nu)$, and $secure^{data}_{P,u}(r, i \vdash_u \phi^{rw}_{s,u}\nu)$ hold.*

Proof. The security of $r, i \vdash_u \phi^{rw}_{s,u}$ follows trivially from that of $r, i \vdash_u \phi^{\top}_{s,u}$ and $r, i \vdash_u \phi^{\bot}_{s,u}$. Therefore, in the following we prove just that $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u}\nu)$ and $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u}\nu)$ hold. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be a partial M-state, $u \in U$ be a user, and ϕ be a D-formula. Furthermore, let ν be an assignment that is well-formed for ϕ. We prove our claim by induction on the structure of the formula ϕ.

Base Case There are four cases:

1. $\phi := x = y$. The claim holds trivially. Indeed, $\phi^{\top}_{s,u}\nu$ and $\phi^{\bot}_{s,u}\nu$ are always equivalent either to $\top$ or to $\bot$. Since for all $s', s'' \in [\![sysState(last(r^i))]\!]^{data}_{M,u}$, $[\top]^{s'.db} = [\top]^{s''.db}$ and $[\bot]^{s'.db} = [\bot]^{s''.db}$, it follows that both $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u}\nu)$ and $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u}\nu)$ hold.
2. $\phi := \top$. The proof of this case is similar to that of $\phi := x = y$.
3. $\phi := \bot$. The proof of this case is similar to that of $\phi := x = y$.
4. $\phi := R(\overline{x})$. Assume, for contradiction's sake, that $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u}\nu)$ does not hold. From this and $secure^{data}_{P,u}$'s definition, it follows that there are two M-system states $s' = \langle db', U, sec, T, V\rangle$ and $s'' = \langle db'', U, sec, T, V\rangle$ in $[\![sysState(last(r^i))]\!]^{data}_{M,u}$ such that $[\phi^{\top}_{s,u}\nu]^{db'} \neq [\phi^{\top}_{s,u}\nu]^{db''}$. Note that this rules out the cases in which $R^{v}_{s,u} = \emptyset$ for any $v \in \{\top, \bot\}$. We assume without loss of generality that $[\phi^{\top}_{s,u}\nu]^{db'} = \top$ and $[\phi^{\top}_{s,u}\nu]^{db''} = \bot$. From this and $\phi^{\top}_{s,u} := \bigvee_{S \in R^{\top}_{s,u}} S(\overline{x})$, it follows that there is a predicate symbol S in the extended vocabulary such that $\nu(\overline{x}) \in db'(S)$ and $\nu(\overline{x}) \notin db''(S)$. There are two cases:
   - S is a table in D or a view in V. Since $S \in R^{\top}_{s,u}$, it follows that $\langle \oplus, \texttt{SELECT}, S\rangle \in permissions(last(r^i), u)$. Note that $permissions(s', u)$ is identical to $permissions(s'', u)$ and $permissions(last(r^i), u)$ because all the states are in the same equivalence class. From $s' \cong^{data}_{M,u} s''$, $\langle \oplus, \texttt{SELECT}, S\rangle \in permissions(s', u)$, and the definition of data indistinguishability, it follows that $db'(S) = db''(S)$. From this, it follows that $\nu(\overline{x}) \in db'(S)$ iff $\nu(\overline{x}) \in db''(S)$, which contradicts $\nu(\overline{x}) \in db'(S)$ and $\nu(\overline{x}) \notin db''(S)$.
   - S is a projection of O, which is either a table in D or a view in V. From $S \in R^{\top}_{s,u}$ and $R^{\top}_{s,u}$'s definition, it follows that $\langle \oplus, \texttt{SELECT}, O\rangle \in permissions(last(r^i), u)$. From $s' \cong^{data}_{M,u} s''$, $\langle \oplus, \texttt{SELECT}, O\rangle \in permissions(s', u)$, and the definition of data indistinguishability, it follows that $db'(O) = db''(O)$. From this and the definition of S, it also follows that $db'(S) = db''(S)$[1]. From this, it follows that $\nu(\overline{x}) \in db'(S)$ iff $\nu(\overline{x}) \in db''(S)$, which contradicts $\nu(\overline{x}) \in db'(S)$ and $\nu(\overline{x}) \notin db''(S)$.

   The proof of $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u}\nu)$ is analogous.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all sub-formulae of ϕ. We now show that our claim holds also for ϕ. There are a number of cases depending on ϕ's structure.

1. $\phi := \psi \wedge \gamma$. Assume, for contradiction's sake, that $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u}\nu)$ does not hold. From this and $secure^{data}_{P,u}$'s definition, it follows that there are two M-system states $s' = \langle db', U, sec, T, V\rangle$ and $s'' = \langle db'', U, sec, T, V\rangle$ in $[\![sysState(last(r^i))]\!]^{data}_{M,u}$ such that $[\phi^{\top}_{s,u}\nu]^{db'} \neq [\phi^{\top}_{s,u}\nu]^{db''}$. We assume, without loss of generality, that $[\phi^{\top}_{s,u}\nu]^{db'} = \top$ and $[\phi^{\top}_{s,u}\nu]^{db''} = \bot$. From this and $\phi^{\top}_{s,u} = \psi^{\top}_{s,u} \wedge \gamma^{\top}_{s,u}$, it follows that $[\psi^{\top}_{s,u}\nu]^{db'} = \top$ and $[\psi^{\top}_{s,u}\nu]^{db''} = \bot$ or $[\gamma^{\top}_{s,u}\nu]^{db'} = \top$ and $[\gamma^{\top}_{s,u}\nu]^{db''} = \bot$. We assume, without loss of generality, that $[\psi^{\top}_{s,u}\nu]^{db'} = \top$ and $[\psi^{\top}_{s,u}\nu]^{db''} = \bot$. From this, it follows that $secure^{data}_{P,u}(r, i \vdash_u \psi^{\top}_{s,u}\nu)$ does not hold. From the induction hypothesis, it follows that $secure^{data}_{P,u}(r, i \vdash_u \psi^{\top}_{s,u}\nu)$ holds leading to a contradiction. The proof of $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u}\nu)$ is analogous.
2. $\phi := \psi \vee \gamma$. The proof of this case is similar to that of $\phi := \psi \wedge \gamma$.

[1] With a slight abuse of notation, we denote by $db(S)$ the materializaton of S in db.

3. $\phi := \neg\psi$. Assume, for contradiction's sake, that $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u}\nu)$ does not hold. From this and $secure^{data}_{P,u}$'s definition, it follows that there are two M-system states $s' = \langle db', U, sec, T, V\rangle$ and $s'' = \langle db'', U, sec, T, V\rangle$ in $[\![sysState(last(r^i))]\!]^{data}_{M,u}$ such that $[\phi^{\top}_{s,u}\nu]^{db'} \neq [\phi^{\top}_{s,u}\nu]^{db''}$. We assume, without loss of generality, that $[\phi^{\top}_{s,u}\nu]^{db'} = \top$ and $[\phi^{\top}_{s,u}\nu]^{db''} = \bot$. From this and $\phi^{\top}_{s,u} = \neg\psi^{\bot}_{s,u}$, it follows that $[\psi^{\bot}_{s,u}\nu]^{db'} = \bot$ and $[\psi^{\bot}_{s,u}\nu]^{db''} = \top$. From this, it follows that $secure^{data}_{P,u}(r, i \vdash_u \psi^{\bot}_{s,u}\nu)$ does not hold. From the induction hypothesis and $\phi := \neg\psi$, it follows that $secure^{data}_{P,u}(r, i \vdash_u \psi^{\bot}_{s,u}\nu)$ holds leading to a contradiction. The proof of $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u}\nu)$ is analogous.
4. $\phi := \exists x.\, \psi$. Assume, for contradiction's sake, that $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u}\nu)$ does not hold. From this and $secure^{data}_{P,u}$'s definition, it follows that there are two M-system states $s' = \langle db', U, sec, T, V\rangle$ and $s'' = \langle db'', U, sec, T, V\rangle$ in $[\![sysState(last(r^i))]\!]^{data}_{M,u}$ such that $[\phi^{\top}_{s,u}\nu]^{db'} \neq [\phi^{\top}_{s,u}\nu]^{db''}$. We assume, without loss of generality, that $[\phi^{\top}_{s,u}\nu]^{db'} = \top$ and $[\phi^{\top}_{s,u}\nu]^{db''} = \bot$. From this and $\phi^{\top}_{s,u} = \exists x.\, \psi^{\top}_{s,u}$, it follows that there is a $v' \in \mathbf{dom}$ such that $[\psi^{\top}_{s,u}\nu[x \mapsto v']]^{db'} = \top$ and there is no $v'' \in \mathbf{dom}$ such that $[\psi^{\top}_{s,u}\nu[x \mapsto v'']]^{db''} = \top$. Therefore, $[\psi^{\top}_{s,u}\nu[x \mapsto v']]^{db'} = \top$ and $[\psi^{\top}_{s,u}\nu[x \mapsto v']]^{db''} = \bot$. Note that $\nu[x \mapsto v']$ is well-formed for $\psi^{\top}_{s,u}$. From this, it follows that $secure^{data}_{P,u}(r, i \vdash_u \psi^{\top}_{s,u}\nu[x \mapsto v'])$ does not hold. However, from the fact that $\nu[x \mapsto v']$ is well-formed for $\psi^{\top}_{s,u}$ and the induction hypothesis, it follows that $secure^{data}_{P,u}(r, i \vdash_u \psi^{\top}_{s,u}\nu[x \mapsto v'])$ holds leading to a contradiction. The proof of $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u}\nu)$ is analogous.
5. $\phi := \forall x.\, \psi$. The proof of this case is similar to that of $\phi := \exists x.\, \psi$.

This completes the proof of the induction step.

This completes the proof of our claim. □

Lemma C.7 states that the result of the *bound* function is the same for any two data indistinguishable states.

Lemma C.7. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ and $s' = \langle db', U', sec', T', V'\rangle$ be two partial M-states, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a D-formula. If $s \cong^{data}_{M,u} s'$, then $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v)$.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ and $s' = \langle db', U', sec', T', V'\rangle$ be two partial M-states, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a D-formula. We prove our claim by induction on the structure of the formula ϕ.

Base Case There are four cases:

1. $\phi := y = z$. The result of $bound(\phi, s, u, x, v)$ and $bound(\phi, s', u, x, v)$ does not depend on s. Therefore, $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v)$.
2. $\phi := \top$. $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v) = \bot$.
3. $\phi := \bot$. $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v) = \bot$.
4. $\phi := R(\overline{x})$. The result of $bound(\phi, s, u, x, v)$ and $bound(\phi, s', u, x, v)$ depend only on the sets $R^v_{s,u}$ and $R^v_{s',u}$, which in turn depend on the content of the sets R^v_s, $R^v_{s'}$, $AUTH^*_{s,u}$, and $AUTH^*_{s',u}$. Assume that $s \cong^{data}_{M,u} s'$. From this, it follows that $R^v_s = R^v_{s'}$ (because D is the same and $V = V'$) and $AUTH^*_{s,u} = AUTH^*_{s',u}$ (because $sec = sec'$). From this, it follows that $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v)$.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all sub-formulae of ϕ. We now show that our claim holds also for ϕ. There are a number of cases depending on ϕ's structure.

1. $\phi := \psi \wedge \gamma$. Assume that $s \cong^{data}_{M,u} s'$. From this and the induction hypothesis, it follows that $bound(\psi, s, u, x, v) = bound(\psi, s', u, x, v)$ and $bound(\gamma, s, u, x, v) = bound(\gamma, s', u, x, v)$. From this and $bound(\phi, s, u, x, v) := bound(\psi, s, u, x, v) \vee bound(\gamma, s, u, x, v)$, it follows that $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v)$.
2. $\phi := \psi \vee \gamma$. The proof of this case is similar to that of $\phi := \psi \wedge \gamma$.
3. $\phi := \neg\psi$. Assume that $s \cong^{data}_{M,u} s'$. From this and the induction hypothesis, it follows that $bound(\psi, s, u, x, v) = bound(\psi, s', u, x, v)$. From this, $bound(\neg\psi, s, u, x, v) = bound(\psi, s, u, x, \neg v)$, and $bound(\neg\psi, s', u, x, v) = bound(\psi, s', u, x, \neg v)$, it follows that $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v)$.
4. $\phi := \exists y.\, \psi$. Assume that $s \cong^{data}_{M,u} s'$. There are two cases:
   (a) $x = y$. In this case, the proof is trivial as $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v) = \bot$.
   (b) $x \neq y$. In this case, $bound(\phi, s, u, x, v) = bound(\psi, s, u, x, v) \wedge bound(\psi, s, u, y, v)$ and $bound(\phi, s', u, x, v) = bound(\psi, s', u, x, v) \wedge bound(\psi, s', u, y, v)$. From $s \cong^{data}_{M,u} s'$ and the induction hypothesis, it follows that $bound(\psi, s, u, x, v) = bound(\psi, s', u, x, v)$ and

$bound(\psi, s, u, y, v) = bound(\psi, s', u, y, v)$. From this, $bound(\phi, s, u, x, v) = bound(\psi, s, u, x, v) \wedge bound(\psi, s, u, y, v)$, and $bound(\phi, s', u, x, v) = bound(\psi, s', u, x, v) \wedge bound(\psi, s', u, y, v)$, it follows that $bound(\phi, s, u, x, v) = bound(\phi, s', u, x, v)$.

5. $\phi := \forall x.\, \psi$. The proof of this case is similar to that of $\phi := \exists x.\, \psi$.

This completes the proof of the induction step.

This completes the proof of our claim. □

Lemma C.8 states that the formulae resulting from our rewriting are the same for any two data indistinguishable states.

Lemma C.8. *Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ and $s' = \langle db', U', sec', T', V' \rangle$ be two partial M-states, $u \in U$ be a user, and ϕ be a D-formula. If $s \cong^{data}_{M,u} s'$, then $\phi^{\top}_{s,u} = \phi^{\top}_{s',u}$, $\phi^{\bot}_{s,u} = \phi^{\bot}_{s',u}$, and $\phi^{rw}_{s,u} = \phi^{rw}_{s',u}$.*

Proof. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ and $s' = \langle db', U', sec', T', V' \rangle$ be two partial M-states, $u \in U$ be a user, and ϕ be a D-formula. We prove our claim by induction on the structure of the formula ϕ.

Base Case There are four cases:

1. $\phi := x = y$. The claim holds trivially. Indeed, $\phi^{\top}_{s,u} = \phi^{\bot}_{s,u} = \phi$.
2. $\phi := \top$. The proof of this case is similar to that of $\phi := x = y$.
3. $\phi := \bot$. The proof of this case is similar to that of $\phi := x = y$.
4. $\phi := R(\overline{x})$. The formulae $\phi^{\top}_{s,u}$ and $\phi^{\top}_{s',u}$ depend only on the sets $R^{\top}_{s,u}$ and $R^{\top}_{s',u}$, which in turn depends on $R^{\top}_{s}$, $R^{\top}_{s'}$, $AUTH^{*}_{s,u}$, and $AUTH^{*}_{s',u}$. If $s \cong^{data}_{M,u} s'$, then $R^{\top}_{s} = R^{\top}_{s'}$ (because D is the same and $V = V'$) and $AUTH^{*}_{s,u} = AUTH^{*}_{s',u}$ (because $sec = sec'$). Therefore, $\phi^{\top}_{s,u} = \phi^{\top}_{s',u}$. The proof for $\phi^{\bot}_{s,u}$ is analogous.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all formulae whose length is less than ϕ. We now show that our claim holds also for ϕ. There are a number of cases depending on ϕ's structure.

1. $\phi := \psi \wedge \gamma$. Assume that $s \cong^{data}_{M,u} s'$. From this and the induction hypothesis, it follows that $\psi^{\top}_{s,u} = \psi^{\top}_{s',u}$ and $\gamma^{\top}_{s,u} = \gamma^{\top}_{s',u}$. From this, $\phi := \psi \wedge \gamma$, $\phi^{\top}_{s,u} := \psi^{\top}_{s,u} \wedge \gamma^{\top}_{s,u}$, and $\phi^{\top}_{s',u} := \psi^{\top}_{s',u} \wedge \gamma^{\top}_{s',u}$, it follows that $\phi^{\top}_{s,u} = \phi^{\top}_{s',u}$. The proof of $\phi^{\bot}_{s,u} = \phi^{\bot}_{s',u}$ is analogous.
2. $\phi := \psi \vee \gamma$. The proof of this case is similar to that of $\phi := \psi \wedge \gamma$.
3. $\phi := \neg\psi$. Assume that $s \cong^{data}_{M,u} s'$. From this and the induction hypothesis, it follows that $\psi^{\top}_{s,u} = \psi^{\top}_{s',u}$ and $\psi^{\bot}_{s,u} = \psi^{\bot}_{s',u}$. From this, $\phi := \neg\psi$, $\phi^{\top}_{s,u} := \neg\psi^{\bot}_{s,u}$, and $\phi^{\top}_{s',u} := \neg\psi^{\bot}_{s',u}$, it follows that $\phi^{\top}_{s,u} = \phi^{\top}_{s',u}$. The proof of $\phi^{\bot}_{s,u} = \phi^{\bot}_{s',u}$ is analogous.
4. $\phi := \exists x.\, \psi$. Assume that $s \cong^{data}_{M,u} s'$. From this and the induction hypothesis, it follows that $\psi^{\top}_{s,u} = \psi^{\top}_{s',u}$. We remark that $bound(\psi, s, u, x, \top) = bound(\psi, s', u, x, \top)$, as proved in Lemma C.7. There are two cases:
   (a) $bound(\psi, s, u, x, \top) = \top$. From this, $bound(\psi, s, u, x, \top) = bound(\psi, s', u, x, \top)$, $\psi^{\top}_{s,u} = \psi^{\top}_{s',u}$, $\phi := \exists x.\, \psi$, $\phi^{\top}_{s,u} := \exists x.\, \psi^{\top}_{s,u}$, and $\phi^{\top}_{s',u} := \exists x.\, \psi^{\top}_{s',u}$, it follows that $\phi^{\top}_{s,u} = \phi^{\top}_{s',u}$.
   (b) $bound(\psi, s, u, x, \top) = \bot$. From this, $bound(\psi, s, u, x, \top) = bound(\psi, s', u, x, \top)$, and $\phi^{\top}_{s,u}$'s definition, it follows that $\phi^{\top}_{s',u} = \phi^{\top}_{s',u} = \bot$.

   The proof of $\phi^{\bot}_{s,u} = \phi^{\bot}_{s',u}$ is analogous.
5. $\phi := \forall x.\, \psi$. The proof of this case is similar to that of $\phi := \exists x.\, \psi$.

This completes the proof of the induction step.

The equivalence $\phi^{rw}_{s,u} = \phi^{rw}_{s',u}$ follows trivially from $\phi^{rw}_{s,u}$'s definition and the fact that $\phi^{\top}_{s,u} = \phi^{\top}_{s',u}$ and $\phi^{\bot}_{s,u} = \phi^{\bot}_{s',u}$. This completes the proof of our claim. □

Before proving the domain independence of $\phi^{\top}_{s,u}$ and $\phi^{\bot}_{s,u}$, we introduce some notation. The relation *gen*, introduced in [160], is the smallest relation defined by the rules in Figure C.1. Note that we extended *gen* by adding the rules *Equiv*, *Const 1*, and *Const 2*. A relational calculus formula ϕ is *allowed* iff it satisfies the following conditions:

- for all $x \in free(\phi)$, $gen(x, \phi)$ holds,
- for every sub-formula $\exists x.\psi$ in ϕ, $gen(x, \psi)$ holds, and
- for every sub-formula $\forall x.\psi$ in ϕ, $gen(x, \neg\psi)$ holds.

As shown in [160], every allowed formula is domain independent. Note that the addition of the *Equiv*, *Const 1*, and *Const 2* rules does not modify this result.

Lemma C.9 presents some useful equivalences that we use in our proof of domain independence.

Lemma C.9. *Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ be an M-system state, $u \in U$ be a user, and $v \in \{\top, \bot\}$. For any formulae ϕ and ψ, the following equivalences hold:*

- $(\neg\phi)^{v}_{s,u} \equiv \neg\phi^{\neg v}_{s,u}$,

$$\frac{x \in \overline{x}}{gen(x, R(\overline{x}))}\ \text{Pred} \qquad \frac{gen(x, push(\neg\phi))}{gen(x, \neg\phi)}\ \text{Neg}$$

$$\frac{x \neq y \quad gen(x,\phi)}{gen(x, \exists y.\phi)}\ \text{Exists} \qquad \frac{x \neq y \quad gen(x,\phi)}{gen(x, \forall y.\phi)}\ \text{For all}$$

$$\frac{gen(x,\phi) \quad gen(x,\psi)}{gen(x, \phi \vee \psi)}\ \text{Or} \qquad \frac{gen(x,\psi) \quad \phi \equiv \psi}{gen(x,\phi)}\ \text{Equiv}$$

$$\frac{v \in \mathbf{dom}}{gen(x, x = v)}\ \text{Const 1} \qquad \frac{v \in \mathbf{dom}}{gen(x, v = x)}\ \text{Const 2}$$

$$\frac{gen(x,\phi)}{gen(x, \phi \wedge \psi)}\ \text{And 1} \qquad \frac{gen(x,\psi)}{gen(x, \phi \wedge \psi)}\ \text{And 2}$$

$$push(\phi) = \begin{cases} \neg\psi \vee \neg\gamma & \text{if } \phi := \neg(\psi \wedge \gamma) \\ \neg\psi \wedge \neg\gamma & \text{if } \phi := \neg(\psi \vee \gamma) \\ \forall x.\, \neg\psi & \text{if } \phi := \neg\exists x.\psi \\ \exists x.\, \neg\psi & \text{if } \phi := \neg\forall x.\psi \\ \psi & \text{if } \phi := \neg\neg\psi \\ x \neq y & \text{if } \phi := \neg(x = y) \\ x = y & \text{if } \phi := \neg(x \neq y) \end{cases}$$

FIGURE C.1: Rules defining the *gen* relation.

- $(\phi)^v_{s,u} \wedge (\psi)^v_{s,u} \equiv (\phi \wedge \psi)^v_{s,u}$,
- $(\phi)^v_{s,u} \vee (\psi)^v_{s,u} \equiv (\phi \vee \psi)^v_{s,u}$,
- $(\exists x.\, \phi)^v_{s,u} \equiv (\neg\forall x.\, \neg\phi)^v_{s,u}$, *and*
- $(\forall x.\, \phi)^v_{s,u} \equiv (\neg\exists x.\, \neg\phi)^v_{s,u}$.

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ, ψ be two formulae.

- $(\neg\phi)^v_{s,u} \equiv \neg\phi^{\neg v}_{s,u}$. This case follows trivially from the definition of the rewriting.
- $(\phi)^v_{s,u} \wedge (\psi)^v_{s,u} \equiv (\phi \wedge \psi)^v_{s,u}$. This case follows trivially from the definition of the rewriting.
- $(\phi)^v_{s,u} \vee (\psi)^v_{s,u} \equiv (\phi \vee \psi)^v_{s,u}$ This case follows trivially from the definition of the rewriting.
- $(\exists x.\, \phi)^v_{s,u} \equiv (\neg\forall x.\, \neg\phi)^v_{s,u}$. There are two cases:
  1. $bound(\phi, s, u, x, v) = \top$. From this, it follows that $(\exists x.\, \phi)^v_{s,u} = \exists x.\, \phi^v_{s,u}$. From $(\neg\phi)^v_{s,u} \equiv \neg\phi^{\neg v}_{s,u}$, it follows that $(\neg\forall x.\, \neg\phi)^v_{s,u} \equiv \neg(\forall x.\, \neg\phi)^{\neg v}_{s,u}$. From the definition of *bound*, it follows that $bound(\neg\phi, s, u, x, \neg v) = bound(\phi, s, u, x, \neg\neg v)$. From this and $v = \neg\neg v$, it follows that $bound(\neg\phi, s, u, x, \neg v) = bound(\phi, s, u, x, v)$. From this and $bound(\phi, s, u, x, v) = \bot$, it follows that $bound(\neg\phi, s, u, x, \neg v) = \top$. From this, it follows that $(\forall x.\, \neg\phi)^{\neg v}_{s,u} = \forall x.\, (\neg\phi)^{\neg v}_{s,u}$. From this and $(\neg\phi)^v_{s,u} \equiv \neg\phi^{\neg v}_{s,u}$, it follows that $(\forall x.\, \neg\phi)^{\neg v}_{s,u} \equiv \forall x.\, \neg\phi^v_{s,u}$. From this and $(\neg\forall x.\, \neg\phi)^v_{s,u} \equiv \neg(\forall x.\, \neg\phi)^{\neg v}_{s,u}$, it follows that $(\neg\forall x.\, \neg\phi)^v_{s,u} \equiv \neg\forall x.\, \neg\phi^v_{s,u}$. From this and standard RC equivalences, it follows that $(\neg\forall x.\, \neg\phi)^v_{s,u} \equiv \exists x.\, \phi^v_{s,u}$.
  2. $bound(\phi, s, u, x, v) = \bot$. From this, it follows that $(\exists x.\, \phi)^v_{s,u} = \neg v$. From $(\neg\phi)^v_{s,u} \equiv \neg\phi^{\neg v}_{s,u}$ and $(\neg\forall x.\, \neg\phi)^v_{s,u}$, it follows that $(\neg\forall x.\, \neg\phi)^v_{s,u} \equiv \neg(\forall x.\, \neg\phi)^{\neg v}_{s,u}$. From the definition of *bound*, it follows that $bound(\neg\phi, s, u, x, \neg v) = bound(\phi, s, u, x, \neg\neg v)$. From this and $v = \neg\neg v$, it follows that $bound(\neg\phi, s, u, x, \neg v) = bound(\phi, s, u, x, v)$. From this and $bound(\phi, s, u, x, v) = \bot$, it follows that $bound(\neg\phi, s, u, x, \neg v) = \bot$. From this, it follows that $(\forall x.\, \neg\phi)^{\neg v}_{s,u} = v$. From this and $(\neg\forall x.\, \neg\phi)^v_{s,u} \equiv \neg(\forall x.\, \neg\phi)^{\neg v}_{s,u}$, it follows that $(\neg\forall x.\, \neg\phi)^v_{s,u} \equiv \neg v$. From this and $(\exists x.\, \phi)^v_{s,u} = \neg v$, it follows that $(\exists x.\, \phi)^v_{s,u} \equiv (\neg\forall x.\, \neg\phi)^v_{s,u}$.
- $(\forall x.\, \phi)^v_{s,u} \equiv (\neg\exists x.\, \neg\phi)^v_{s,u}$. The proof of this case is similar to that of $(\exists x.\, \phi)^v_{s,u} \equiv (\neg\forall x.\, \neg\phi)^v_{s,u}$.

This completes the proof. □

Lemma C.10. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a formula. Furthermore, let $x \in free(\phi) \cap free(\phi^v_{s,u})$. If $gen(x, \phi)$ holds, then $gen(x, \phi^v_{s,u})$ holds.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a formula. Furthermore, let $x \in free(\phi) \cap free(\phi^v_{s,u})$. We prove our claim by induction on the length of ϕ. In the following, the length of ϕ is the number of predicates, quantifiers, negations, conjunctions, and disjunctions in ϕ.

Base Case There are four cases:

1. $\phi := x = y$. In this case, the claim holds trivially.
2. $\phi := \top$. In this case, the claim holds trivially.
3. $\phi := \bot$. In this case, the claim holds trivially.
4. $\phi := R(\overline{x})$. Assume $gen(x, \phi)$ holds. From this, it follows that x is one of the free variables in $\overline{x}$. Furthermore, from $x \in free(\phi^v_{s,u})$, it follows that $R^v_{s,u} \neq \emptyset$. There are two cases:
   (a) $\phi^v_{s,u}$ is a conjunction of predicates $S(\overline{x})$ such that $gen(x, S(\overline{x}))$ holds. From the rule *And 1*, it follows that $gen(x, \phi^v_{s,u})$ holds.
   (b) $\phi^v_{s,u}$ is a disjunction of predicates $S(\overline{x})$ such that $gen(x, S(\overline{x}))$ holds. From the rule *Or*, it follows that $gen(x, \phi^v_{s,u})$ holds.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all formulae whose length is less than ϕ's length. We now show that our claim holds also for ϕ. There are a number of cases depending on ϕ's structure.

1. $\phi := \psi \wedge \gamma$. Assume that $gen(x, \phi)$ holds. From this and the rules *And 1* and *And 2*, it follows that either $gen(x, \psi)$ or $gen(x, \gamma)$ hold. Assume, without loss of generality, that $gen(x, \psi)$ holds. From this and the induction hypothesis, it follows that $gen(x, \psi^v_{s,u})$ holds. From this, $\phi^v_{s,u} := \psi^v_{s,u} \wedge \gamma^v_{s,u}$, and the rule *And 1*, it follows that $gen(x, \phi^v_{s,u})$ holds.
2. $\phi := \psi \vee \gamma$. Assume that $gen(x, \phi)$ holds. From this and the rule *Or*, it follows that both $gen(x, \psi)$ and $gen(x, \gamma)$ hold. From this and the induction hypothesis, it follows that both $gen(x, \psi^v_{s,u})$ and $gen(x, \gamma^v_{s,u})$ hold. From this, $\phi^v_{s,u} := \psi^v_{s,u} \vee \gamma^v_{s,u}$, and the rule *Or*, it follows that $gen(x, \phi^v_{s,u})$ holds.
3. $\phi := \neg\psi$. Assume that $gen(x, \phi)$ holds. From this and the rule *Not*, it follows that $gen(x, push(\neg\psi))$. There are a number of cases depending on ψ. In the following, we exploit standard relational calculus equivalences, see, for instance, [10], and the equivalences we proved in Lemma C.9.
   (a) $\psi := (\alpha \wedge \beta)$. In this case, $push(\neg\psi)$ is $(\neg\alpha \vee \neg\beta)$. From this and $gen(x, push(\neg\psi))$, it follows $gen(x, (\neg\alpha \vee \neg\beta))$. From this and the *Or* rule, it follows that $gen(x, \neg\alpha)$ and $gen(x, \neg\beta)$ hold. From this and the induction hypothesis, it follows that $gen(x, (\neg\alpha)^v_{s,u})$ and $gen(x, (\neg\beta)^v_{s,u})$. From this and the *Or* rule, it follows that $gen(x, (\neg\alpha)^v_{s,u} \vee (\neg\beta)^v_{s,u})$. From this, $(\neg\alpha)^v_{s,u} \vee (\neg\beta)^v_{s,u} \equiv (\neg\alpha \vee \neg\beta)^v_{s,u}$, and the *Equiv* rule, it follows that $gen(x, (\neg\alpha \vee \neg\beta)^v_{s,u})$. From this, $(\neg\alpha \vee \neg\beta)^v_{s,u} \equiv (\neg(\alpha \wedge \beta))^v_{s,u}$, and the *Equiv* rule, it follows that $gen(x, (\neg(\alpha \wedge \beta))^v_{s,u})$. From this, $(\neg(\alpha \wedge \beta))^v_{s,u} \equiv \neg(\alpha \wedge \beta)^{\neg v}_{s,u}$, and the *Equiv* rule, it follows that $gen(x, \neg(\alpha \wedge \beta)^{\neg v}_{s,u})$. From this and $\psi := \alpha \wedge \beta$, it follows that $gen(x, \neg\psi^{\neg v}_{s,u})$. From this and $\phi^v_{s,u} := \neg\psi^{\neg v}_{s,u}$, it follows that $gen(x, \phi^v_{s,u})$ holds.
   (b) $\psi := (\alpha \vee \beta)$. The proof is similar to the $\psi := \neg(\alpha \wedge \beta)$ case.
   (c) $\psi := \exists y.\, \alpha$. In this case, $push(\neg\psi)$ is $\forall y.\, \neg\alpha$. From this and $gen(x, push(\neg\psi))$, it follows $gen(x, \forall y.\, \neg\alpha)$. From this and the induction hypothesis, it follows that $gen(x, (\forall y.\, \neg\alpha)^v_{s,u})$. From this, $\neg\neg(\forall y.\, \neg\alpha)^v_{s,u} \equiv (\forall y.\, \neg\alpha)^v_{s,u}$, and the *Equiv* rule, it follows that $gen(x, \neg\neg(\forall y.\, \neg\alpha)^v_{s,u})$. From this, $\neg\neg(\forall y.\, \neg\alpha)^v_{s,u} \equiv \neg(\neg\forall y.\, \neg\alpha)^{\neg v}_{s,u}$, and the *Equiv* rule, it follows that $gen(x, \neg(\neg\forall y.\, \neg\alpha)^{\neg v}_{s,u})$. From this, $\neg(\neg\forall y.\, \neg\alpha)^{\neg v}_{s,u} \equiv \neg(\exists y.\, \neg\neg\alpha)^{\neg v}_{s,u}$, and the *Equiv* rule, it follows that $gen(x, \neg(\exists y.\, \neg\neg\alpha)^{\neg v}_{s,u})$. From this, $\neg(\exists y.\, \neg\neg\alpha)^{\neg v}_{s,u} \equiv \neg(\exists y.\, \alpha)^{\neg v}_{s,u}$, and the *Equiv* rule, it follows that $gen(x, \neg(\exists y.\, \alpha)^{\neg v}_{s,u})$. From this and $\psi := \exists y.\, \alpha$, it follows that $gen(x, \neg\psi^{\neg v}_{s,u})$. From this and $\phi^v_{s,u} := \neg\psi^{\neg v}_{s,u}$, it follows that $gen(x, \phi^v_{s,u})$ holds.
   (d) $\psi := \forall y.\, \alpha$. The proof is similar to the $\psi := \neg\exists y.\, \alpha$ case.
   (e) $\psi := \neg\alpha$. In this case, $push(\neg\psi)$ is α. From this and $gen(x, push(\neg\psi))$, it follows $gen(x, \alpha)$. From this and the induction hypothesis, it follows that $gen(x, \alpha^v_{s,u})$. From this, $\neg\neg\alpha^v_{s,u} \equiv \alpha^v_{s,u}$, and the *Equiv* rule, it follows that $gen(x, \neg\neg\alpha^v_{s,u})$. From this, $\neg\neg\alpha^v_{s,u} \equiv \neg(\neg\alpha)^{\neg v}_{s,u}$, and the *Equiv* rule, it follows that $gen(x, \neg(\neg\alpha)^{\neg v}_{s,u})$. From this and $\psi := \neg\alpha$, it follows that $gen(x, \neg\psi^{\neg v}_{s,u})$. From this and $\phi^v_{s,u} := \neg\psi^{\neg v}_{s,u}$, it follows that $gen(x, \phi^v_{s,u})$ holds.
   (f) $\psi := x = y$. The proof for this case is trivial.
   (g) $\psi := x \neq y$. The proof for this case is trivial.
4. $\phi := \exists x.\, \psi$. Assume that $gen(x, \phi)$ holds. From this and the rule *Exists*, it follows that $gen(x, \psi)$ holds. From this and the induction hypothesis, it follows that $gen(x, \psi^v_{s,u})$ holds. From this, $\phi^v_{s,u} := \exists x.\, \psi^v_{s,u}$, and the rule *Exists*, it follows that $gen(x, \phi^v_{s,u})$ holds.
5. $\phi := \forall x.\, \psi$. The proof of this case is similar to that of $\phi := \exists x.\, \psi$.

This completes the proof of the induction step.

This completes the proof of our claim. □

Lemma C.11. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a formula. For every sub-formula $\exists x.\, \psi$ of ϕ, if $gen(x, \psi)$ holds and $x \in free(\psi) \cap free(\psi^v_{s,u})$, then $gen(x, \psi^v_{s,u})$ holds.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a formula. We prove our claim by induction on the length of ϕ.

In the following, the size of ϕ is the number of predicates, quantifiers, negations, conjunctions, and disjunctions in ϕ.

Base Case The claim is vacuously satisfied for the base cases as there is no sub-formula of the form $\exists x.\, \psi$.

Induction Step Assume that our claim holds for all formulae whose length is less than ϕ. We now show that our claim holds also for ϕ. There are a number of cases depending on ϕ's structure.

1. $\phi := \psi \wedge \gamma$. Let α be a sub-formula of ϕ of the form $\exists x.\, \beta$ such that $gen(x, \beta)$ holds and $x \in free(\beta) \cap free(\beta^v_{s,u})$. The formula α is either a sub-formula of ψ or a sub-formula of γ. From this and the induction hypothesis, it follows that $gen(x, \beta^s_{v,u})$ holds.
2. $\phi := \psi \vee \gamma$. The proof of this case is similar to that of $\phi := \psi \wedge \gamma$.
3. $\phi := \neg\psi$. Let α be a sub-formula of ϕ of the form $\exists x.\, \beta$ such that $gen(x, \beta)$ holds and $x \in free(\beta) \cap free(\beta^v_{s,u})$. Since $\phi := \neg\psi$, the formula α is also a sub-formula of ψ. From this and the induction hypothesis, it follows that $gen(x, \beta^s_{v,u})$ holds.
4. $\phi := \exists x.\, \psi$. Let α be a sub-formula of ϕ of the form $\exists x.\, \beta$ such that $gen(x, \beta)$ holds and $x \in free(\beta) \cap free(\beta^v_{s,u})$. There are two cases:
   (a) α is a sub-formula of ψ. From this and the induction hypothesis, it follows that $gen(x, \beta^s_{v,u})$ holds.
   (b) $\alpha = \phi$. From $gen(x, \beta)$, $x \in free(\beta) \cap free(\beta^v_{s,u})$, and Lemma C.10, it follows that $gen(x, \beta^v_{s,u})$ holds.
5. $\phi := \forall x.\, \psi$. The proof of this case is similar to that of $\phi := \exists x.\, \psi$.

This completes the proof of the induction step.

This completes the proof of our claim. □

Lemma C.12. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a formula. For every sub-formula $\forall x.\, \psi$ of ϕ, if $gen(x, \psi)$ holds and $x \in free(\psi) \cap free(\psi^v_{s,u})$, then $gen(x, (\neg\psi)^v_{s,u})$ holds.*

Proof. The proof is similar to that of Lemma C.11. □

Lemma C.13. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and ϕ be a formula. Let $Q \in \{\exists, \forall\}$ be a quantifier and $subs_Q(\phi)$ be the set of sub-formulae of ϕ of the form $Q\, x.\, \psi$. There is a surjective function f from $subs_Q(\phi)$ to $subs_Q(\phi^v_{s,u})$ such that for any $Q\, x.\, \psi$ in $subs_Q(\phi)$, if $f(Q\, x.\, \psi)$ is defined, then $f(Q\, x.\, \psi)^v_{s,u} = Q\, x.\, \psi^v_{s,u}$.*

Proof. The claim directly follows from the definition of $\phi^v_{s,u}$. □

Finally, Proposition C.8 states that our rewriting is domain independent whenever the original formula and the views can be proved to be allowed (which implies that they are domain independent).

Proposition C.8. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be an M-system state, $u \in U$ be a user, and ϕ be a formula. If ϕ is allowed and all views in V are allowed, then $\phi^\top_{s,u}$, $\phi^\bot_{s,u}$, and $\phi^{rw}_{s,u}$ are domain independent.*

Proof. From Lemmas C.10–C.13, it follows that if ϕ is allowed, then both $\phi^\top_{s,u}$ and $\phi^\bot_{s,u}$ are allowed. Since every allowed formula is domain independent [160], it follows that both $\phi^\top_{s,u}$ and $\phi^\bot_{s,u}$ are domain independent. Finally, the domain independence of $\phi^{rw}_{s,u}$ follows easily from its definition and the domain independence of $\phi^\top_{s,u}$ and $\phi^\bot_{s,u}$. □

C.4.4 The *secure* function is a sound under-approximation of data security

We now prove the main result about our rewriting, namely that the *secure* function is, indeed, a sound, under-approximation of the notion of judgment's security.

Proposition C.9. *Let $P = \langle M, f\rangle$ be an extended configuration, L be the P-LTS, $u \in \mathcal{U}$ be a user, $r \in traces(L)$ be an L-run, $\phi \in RC_{bool}$ be a sentence, and $1 \leq i \leq |r|$. Furthermore, let s be the i-th state in r. The following statements hold:*

1. *Given a judgment $r, i \vdash_u \phi$, if $secure(u, \phi, s) = \top$, then $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds.*
2. *Given a judgment $r, i \vdash_u \phi$, if $secure(u, \phi, s) = \top$, then $secure_{P,u}(r, i \vdash_u \phi)$ holds.*

Proof. Note that the second statement follows trivially from Proposition C.7 and the first statement. Therefore, in the following we prove just that given a judgment $r, i \vdash_u \phi$, if $secure(u, \phi, s) = \top$, then $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds.

Let $P = \langle M, f\rangle$ be an extended configuration, L be the P-LTS, $u \in \mathcal{U}$ be a user, $r \in traces(L)$ be an L-run, $\phi \in RC_{bool}$ be a sentence, and $1 \leq i \leq |r|$. Furthermore, let $s = \langle db, U, sec, T, V, c\rangle$ be the i-th state in r. Assume that $secure(u, \phi, s) = \top$. From this and *secure*'s definition, $[\phi^{rw}_{s,u}]^{db} = \bot$. In the

following, with a slight abuse of notation we ignore the *inline* and *ext* functions in $\phi^{rw}_{s,u}$'s definition. This is without loss of generality since *inline* and *ext* do not modify ϕ's result. From this and $\phi^{rw}_{s,u}$'s definition, it follows that either $[\phi^{\top}_{s,u}]^{db} = \top$ or $[\phi^{\bot}_{s,u}]^{db} = \bot$. Note that from Lemma C.6, it follows that $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u})$ and $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u})$. Furthermore, let Δ be the equivalence class $[\![sysState(s)]\!]^{data}_{M,u}$. There are two cases:

1. $[\phi^{\top}_{s,u}]^{db} = \top$. From $secure^{data}_{P,u}(r, i \vdash_u \phi^{\top}_{s,u})$, it follows that for all $s', s'' \in \Delta$, $[\phi^{\top}_{s,u}]^{s'.db} = [\phi^{\top}_{s,u}]^{s''.db}$. From this, $s \in \Delta$, and $[\phi^{\top}_{s,u}]^{db} = \top$, it follows that $[\phi^{\top}_{s,u}]^{s'.db} = \top$ for all $s' \in \Delta$. From Lemma C.8, it follows that for all $s', s'' \in \Delta$, $\phi^{\top}_{s,u} = \phi^{\top}_{s',u} = \phi^{\top}_{s'',u}$. From this and the fact that for all $s' \in \Delta$, $[\phi^{\top}_{s,u}]^{s'.db} = \top$, it follows that for all $s' \in \Delta$, $[\phi^{\top}_{s',u}]^{s'.db} = \top$. From this and Lemma C.5, it follows that for all $s' \in \Delta$, $[\phi]^{s'.db} = \top$. From this, it follows that for all $s', s'' \in \Delta$, $[\phi]^{s'.db} = [\phi]^{s''.db}$. From this, r's definition, and $secure^{data}_{P,u}$, it follows that $secure^{data}_{P,u}(r, i \vdash_u \phi)$.
2. $[\phi^{\bot}_{s,u}]^{db} = \bot$. From $secure^{data}_{P,u}(r, i \vdash_u \phi^{\bot}_{s,u})$, it follows that for all $s', s'' \in \Delta$, $[\phi^{\bot}_{s,u}]^{s'.db} = [\phi^{\bot}_{s,u}]^{s''.db}$. From this, $s \in \Delta$, and $[\phi^{\bot}_{s,u}]^{db} = \bot$, it follows that $[\phi^{\bot}_{s,u}]^{s'.db} = \bot$ for all $s' \in \Delta$. From Lemma C.8, it follows that for all $s', s'' \in \Delta$, $\phi^{\bot}_{s,u} = \phi^{\bot}_{s',u} = \phi^{\bot}_{s'',u}$. From this and the fact that for all $s' \in \Delta$, $[\phi^{\bot}_{s,u}]^{s'.db} = \bot$, it follows that for all $s' \in \Delta$, $[\phi^{\bot}_{s',u}]^{s'.db} = \bot$. From this and Lemma C.5, it follows that for all $s' \in \Delta$, $[\phi]^{s'.db} = \bot$. From this, it follows that for all $s', s'' \in \Delta$, $[\phi]^{s'.db} = [\phi]^{s''.db}$. From this, r's definition, and $secure^{data}_{P,u}$, it follows that $secure^{data}_{P,u}(r, i \vdash_u \phi)$.

This completes the proof of our claim. □

Lemma C.14 states that the *secure* function produces the same result for indistinguishable states.

Lemma C.14. *Let M be a system configuration, $u \in U$ be a user, $s, s' \in \Omega_M$ be two M-states such that $sysState(s) \cong^{data}_{M,u} sysState(s')$, and ϕ be a sentence. Then, $secure(u, \phi, s) = \top$ iff $secure(u, \phi, s') = \top$.*

Proof. Let M be a system configuration, $u \in U$ be a user, $s = \langle db, U, sec, T, V, c\rangle$ and $s' = \langle db', U', sec', T', V', c'\rangle$ be two M-states such that $sysState(s) \cong^{data}_{M,u} sysState(s')$, and ϕ be a sentence. We now prove that $secure(u, \phi, s) = secure(u, \phi, s')$. Assume, for contradiction's sake, that $secure(u, \phi, s) \neq secure(u, \phi, s')$. From this, it follows that $[\phi^{rw}_{s,u}]^{db} \neq [\phi^{rw}_{s',u}]^{db'}$. From $sysState(s) \cong^{data}_{M,u} sysState(s')$ and Lemma C.8, it follows that $\phi^{rw}_{s,u} = \phi^{rw}_{s',u}$. From this and $[\phi^{rw}_{s,u}]^{db} \neq [\phi^{rw}_{s',u}]^{db'}$, it follows that $[\phi^{rw}_{s,u}]^{db} \neq [\phi^{rw}_{s,u}]^{db'}$. This contradicts $secure^{data}_{P,u}(r, i \vdash_u \phi^{rw}_{s,u})$, which has been proved in Lemma C.6. This completes the proof of our claim. □

C.4.5 Auxiliary results about f_{conf}

Here we prove some auxiliary results about f_{conf}.

Lemma C.15 states that the result of f^u_{conf} is the same for all data indistinguishable states.

Lemma C.15. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $u \in \mathcal{U}$ be a user, $s, s' \in \Omega_M$ be two M-states such that $sysState(s) \cong^{data}_{M,u} sysState(s')$, $invoker(s) = invoker(s')$, and $tr(s) = tr(s')$, and a be an action in $\mathcal{A}_{D,\mathcal{U}}$. Then, $f^u_{conf}(s, a) = f^u_{conf}(s', a)$.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $u \in \mathcal{U}$ be a user, $s = \langle db, U, sec, T, V, c\rangle$ and $s' = \langle db', U', sec', T', V', c'\rangle$ be two M-states such that $sysState(s) \cong^{data}_{M,u} sysState(s')$, $invoker(s) = invoker(s')$, and $tr(s) = tr(s')$, and a be an action in $\mathcal{A}_{D,u}$. There are a number of cases depending on the action a.

1. $a = \langle u', \texttt{SELECT}, \phi\rangle$. Assume, for contradiction's sake, that $f^u_{conf,}(s, a) \neq f^u_{conf}(s', a)$. This happens iff $secure(u, \phi, s) \neq secure(u, \phi, s')$. This contradicts Lemma C.14 because $sysState(s) \cong^{data}_{M,u} sysState(s')$.
2. $a = \langle u', \texttt{INSERT}, R, \bar{t}\rangle$. We claim that $noLeak(s, a, u) = noLeak(s', a, u)$. Assume, for contradiction's sake, that $f^u_{conf}(s, a) \neq f^u_{conf}(s', a)$. This happens iff there is a formula ϕ, which has been derived using the *getInfo*, *getInfoV*, or *getInfoD* functions, such that $secure(u, \phi, s) \neq secure(u, \phi, s')$. This contradicts Lemma C.14 because $sysState(s) \cong^{data}_{M,u} sysState(s')$.
   We prove our claim that $noLeak(s, a, u) = noLeak(s', a, u)$ for any two states s and s' such that $sysState(s) \cong^{data}_{M,u} sysState(s')$. Assume, for contradiction's sake, that this is not the case. Without loss of generality we assume that $noLeak(s, a, u) = \top$ and $noLeak(s', a, u) = \bot$. From $noLeak(s, a, u) = \top$, it follows that for all views V such that $\langle \oplus, \texttt{SELECT}, V\rangle \in permissions(s, u)$ and $R \in tDet(V, s, M)$, for all $o \in tDet(V, s, M)$, $\langle \oplus, \texttt{SELECT}, o\rangle$ is in $permissions(s, u)$. From $sysState(s) \cong^{data}_{M,u} sysState(s')$, it follows that $sec = sec'$. From this, $permissions(s, u) = permissions(s', u)$. From $noLeak(s', a, u) = \bot$, there are two views or tables V' and o such that

$\langle \oplus, \texttt{SELECT}, V' \rangle \in permissions(s', u)$, $\langle \oplus, \texttt{SELECT}, o \rangle \notin permissions(s', u)$, $R \in tDet(V', s', M)$, and $o \in tDet(V', s', M)$. Note that $tDet(V', s', M) = tDet(V', s, M)$ because query determinacy does not consider the database state. From this and $permissions(s, u) = permissions(s', u)$, it follows that there is a view V' such that $\langle \oplus, \texttt{SELECT}, V' \rangle \in permissions(s, u)$ and $R \in tDet(V', s, M)$, such that there is a table $o \in tDet(V', s, M)$ for which $\langle \oplus, \texttt{SELECT}, o \rangle \notin permissions(s, u)$. This contradicts $noLeak(s, a, u) = \top$.

3. $a = \langle u', \texttt{DELETE}, R, \bar{t} \rangle$. The proof of this case is similar to the $a = \langle u', \texttt{INSERT}, R, \bar{t} \rangle$ case.
4. $a = \langle op, u'', p, u' \rangle$, where $op \in \{\oplus, \oplus^*\}$. Assume, for contradiction's sake, that $f^u_{conf,}(s, a) \neq f^u_{conf}(s', a)$. Note that this happens iff $p = \langle \texttt{SELECT}, o \rangle$ for some o. Without loss of generality, we further assume that $f^u_{conf,}(s, a) = \top$ and $f^u_{conf}(s', a) = \bot$. From $f^u_{conf,}(s, a) = \top$, it follows that $\langle \oplus, \texttt{SELECT}, o \rangle \in permissions(s, u)$. From $sysState(s) \cong^{data}_{M,u} sysState(s')$, it follows $permissions(s, u) = permissions(s', u)$. From this and $\langle \oplus, \texttt{SELECT}, o \rangle \in permissions(s, u)$, it follows that $\langle \oplus, \texttt{SELECT}, o \rangle$ is in $permissions(s', u)$. From $f^u_{conf,}(s', a) = \bot$, it follows that $\langle \oplus, \texttt{SELECT}, o \rangle \notin permissions(s, u)$. This contradicts $\langle \oplus, \texttt{SELECT}, o \rangle \in permissions(s', u)$.
5. For any other action a, the proof is trivial.

This completes the proof of our claim. □

Lemma C.16. *Let P be an extended configuration, L be the P-LTS, $r \in traces(L)$ be a run, u be a user, γ be a sentence, and Φ be a set of sentences such that $\Phi \models_{fin} \gamma$. If, for all $\phi \in \Phi$, $secure_{P,u}(r, i \vdash_u \phi)$ holds and $[\phi]^{last(r).db} = \top$, then $secure_{P,u}(r, i \vdash_u \gamma)$ holds and $[\gamma]^{last(r^i).db} = \top$.*

Proof. Let P be an extended configuration, L be the P-LTS, $r \in traces(L)$ be a run, u be a user, γ be a sentence, and Φ be a set of sentences such that $\Phi \models_{fin} \gamma$ such that for all $\phi \in \Phi$, $secure_{P,u}(r, i \vdash_u \phi)$ holds and $[\phi]^{last(r^i).db} = \top$. We now show that $secure_{P,u}(r, i \vdash_u \gamma)$ holds and $[\gamma]^{last(r^i).db} = \top$. From $\Phi \models_{fin} \gamma$,$[\phi]^{last(r^i).db} = \top$ for all $\phi \in \Phi$, and $\models_{fin}$'s definition, it follows that $[\gamma]^{last(r^i).db} = \top$. Assume, for contradiction's sake, that $secure_{P,u}(r, i \vdash_u \gamma)$ does not hold. From this and $[\gamma]^{last(r^i).db} = \top$, it follows that there is a run $r' \in traces(L)$ such that $r^i \cong_{P,u} r'$ such that $[\gamma]^{last(r').db} = \bot$. We claim that for all $\phi \in \Phi$, $[\phi]^{last(r').db} = \top$. From this and $\Phi \models_{fin} \gamma$, it follows that $[\gamma]^{last(r').db} = \top$, which contradicts $[\gamma]^{last(r').db} = \bot$.

We now prove our claim that for all $\phi \in \Phi$, $[\phi]^{last(r').db} = \top$ for any trace r' such that $r^i \cong_{P,u} r'$. From $secure_{P,u}(r, i \vdash_u \phi)$, it follows that $[\phi]^{last(r^i).db} = [\phi]^{last(r').db}$. From this and $[\phi]^{last(r^i).db} = \top$, it follows that $[\phi]^{last(r').db} = \top$. □

Lemma C.17 states that the result of $f^{user(a,s)}_{conf}$ is the same for all data indistinguishable states.

Lemma C.17. *Let $M = \langle D, \Gamma \rangle$ be a system configuration, a be an action in $\mathcal{A}_{D,\mathcal{U}}$, and $s, s' \in \Omega_M$ be two M-states such that $sysState(s) \cong^{data}_{M,user(a,s)} sysState(s')$, $invoker(s) = invoker(s')$, and $trigger(s) = trigger(s')$. Then, $f^{user(a,s)}_{conf}(s, a) = \top$ iff $f^{user(a,s')}_{conf}(s', a) = \top$.*

Proof. Let $s = \langle db, U, sec, T, V, c \rangle$ and $s' = \langle db', U', sec', T', V', c' \rangle$ be two M-states. We assume that s and s' satisfy $sysState(s) \cong^{data}_{M,user(a,s)} sysState(s')$, $invoker(s) = invoker(s')$, and $trigger(s) = trigger(s')$. We first show that $user(a, s) = user(a, s')$. Since $trigger(s) = trigger(s')$, there are two cases:

- $trigger(s) = \epsilon$. In this case, the result of $user(a, s)$ depends just on a. Therefore, $user(a, s) = user(a, s')$.
- $trigger(s) \neq \epsilon$. In this case, $user(a, s) = invoker(s)$ and $user(a, s') = invoker(s')$. From $invoker(s) = invoker(s')$, it follows that $user(a, s) = user(a, s')$.

Let u be the user $user(a, s)$. From Lemma C.15, it follows that $f^u_{conf}(s, a) = f^u_{conf}(s', a)$. This completes the proof. □

C.4.6 Equivalence class preservation

We now introduce the concept of an action that preserves the equivalence class induced by an indistinguishability relation $\cong$. We use this concept in our confidentiality proof (instantiated with the indistinguishability relation $\cong_{P,u}$). In the following, given an extended configuration P, a run r, and an indistinguishability relation $\cong$, we denote by $[\![r]\!]_{P,\cong}$ the equivalence class of r defined by $\cong$ over $traces(L)$, where L is the P-LTS.

Definition C.1. Let $P = \langle M, f \rangle$ be an extended configuration, where $M = \langle D, \Gamma \rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, $r \in traces(L)$ be a run, and a be an action in $\mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$. We denote by $extend(r, a)$, where r is a run and a is an action, the run $r' \in traces(L)$, where $s \in \Omega_M$ and $r' = r{\cdot}a{\cdot}s$, obtained by executing the action a at the end of the run r'. If there is no such run, then $extend(r, a)$ is undefined. We say that *a preserves the equivalence*

class for r, P, *and* $\cong$ iff (1) $extend(r, a)$ is defined, and (2) there is a bijection b between $[\![r]\!]_{P,\cong}$ and $[\![extend(r, a)]\!]_{P,\cong}$ such that for all $r' \in [\![r]\!]_{P,\cong}$, $extend(r', a)$ is defined and $b(r') = extend(r', a)$. □

Furthermore, given a formula ϕ, we denote by $tables(\phi)$ the set of relation schemas used in ϕ', where ϕ' is obtained from ϕ by replacing all views with their definitions until we reach a fixpoint.

Lemma C.18. *Let* $P = \langle M, f\rangle$ *be an extended configuration, where* $M = \langle D, \Gamma\rangle$ *is a system configuration and* f *is an* M*-PDP,* L *be the* P*-LTS,* u *be a user in* $\mathcal{U}$*,* r *be a run in* $traces(L)$*,* $a \in \mathcal{A}_{D,u}$ *be an* `INSERT` *or* `DELETE` *action* $\langle u, op, R, \bar{t}\rangle$*,* ϕ *be a sentence,* i *be such that* $1 \leq i \leq |r|$*,* $triggers(last(r^i)) = \epsilon$*, and* $r^{i+1} = extend(r^i, a)$*, and* $\cong$ *be an indistinguishability relation. If (1)* a *preserves the equivalence class for* r^i*,* P*, and* $\cong$*, and (2) the execution of* a *does not change any table in* $tables(\phi)$ *for any run* $v \in [\![r^i]\!]_{P,\cong}$*, then* $secure_{P,\cong}(r, i \vdash_u \phi)$ *holds iff* $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ *holds.*

Proof. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, u be a user in $\mathcal{U}$, r be a run in $traces(L)$, $a \in \mathcal{A}_{D,u}$ be an `INSERT` or `DELETE` action $\langle u, op, R, \bar{t}\rangle$, ϕ be a sentence, and i be such that $1 \leq i \leq |r|$, $triggers(last(r^i)) = \epsilon$, and $r^{i+1} = extend(r^i, a)$. Assume that (1) a preserves the equivalence class for r^i, P, and $\cong$, and (2) the execution of a does not change any table in $tables(\phi)$ for any run $v \in [\![r^i]\!]_{P,\cong}$. Without loss of generality, assume that a is an `INSERT` action. In the following, we denote the *extend* function by e. Furthermore, we also denote the fact that $secure_{P,\cong}(r, i, u, \phi)$ does not hold as $\neg secure_{P,\cong}(r, i, u, \phi)$. From Definition C.1 and a preserves the equivalence class for r^i, P, and $\cong$, it follows that $e(r', a)$ is defined for any $r' \in [\![r^i]\!]_{P,\cong}$. Assume, for contradiction's sake, that our claim does not hold. There are two cases:

- $secure_{P,\cong}(r, i \vdash_u \phi)$ holds and $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ does not hold. From $secure_{P,\cong}(r, i \vdash_u \phi)$, it follows that for all $r' \in [\![r^i]\!]_{P,\cong}$, $[\phi]^{last(r').db} = [\phi]^{last(r^i).db}$. We claim that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$. From this and $[\phi]^{last(r').db} = [\phi]^{last(r^i).db}$ for all $r' \in [\![r^i]\!]_{P,\cong}$, it follows that $[\phi]^{last(r^i).db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$. From $\neg secure_{P,u}(r, i+1 \vdash_u \phi)$, it follows that there is a run $r' \in [\![r^{i+1}]\!]_{P,\cong}$ such that $[\phi]^{last(r^{i+1}).db} \neq [\phi]^{last(r').db}$. From this, $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ for any $r' \in [\![r^i]\!]_{P,\cong}$, and $e(r^i, a) = r^{i+1}$, it follows that $[\phi]^{last(r^i).db} \neq [\phi]^{last(r').db}$. Let b be the bijection showing that a preserves the equivalence class with respect to r^i, P, and $\cong$. From $e(r^i, a) = r^{i+1}$ and $r' \in [\![r^{i+1}]\!]_{P,\cong}$, it follows that $r' \in [\![e(r^i, a)]\!]_{P,u}$. From this, it follows that there is an $r'' = b^{-1}(r')$ such that $r'' \in [\![r^i]\!]_{P,\cong}$ and $e(r'', a) = r'$. From this and $[\phi]^{last(v).db} = [\phi]^{last(e(v,a)).db}$ for any $v \in [\![r^i]\!]_{P,\cong}$, it follows that $[\phi]^{last(r'').db} = [\phi]^{last(r').db}$. From this and $[\phi]^{last(r^i).db} \neq [\phi]^{last(r').db}$, it follows that $[\phi]^{last(r^i).db} \neq [\phi]^{last(r'').db}$ and $r'' \in [\![r^i]\!]_{P,\cong}$. This contradicts the fact that for all $r' \in [\![r^i]\!]_{P,\cong}$, $[\phi]^{last(r').db} = [\phi]^{last(r^i).db}$. Indeed, $r'' \in [\![r^i]\!]_{P,\cong}$ and $[\phi]^{last(r^i).db} \neq [\phi]^{last(r'').db}$.
  We prove our claim that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$. Assume that this is not the case. This implies that the content of one of the relations that determines ϕ is different in $last(r').db$ and $last(e(r', a)).db$. This is impossible. Indeed, if a's execution has been successful, i.e., $secEx(last(e(r', a))) = \bot$ and $Ex(last(e(r', a))) = \emptyset$, then a's execution does not change any table in $tables(\phi)$, and the set of relations that determines ϕ is always a subset of $tables(\phi)$. This leads to a contradiction, and, therefore, $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds. Similarly, if a's execution has not been successful, i.e., $secEx(last(e(r', a))) = \top$ or $Ex(last(e(r', a))) \neq \emptyset$, then $last(r').db$ is the same as $last(e(r', a)).db$, and the claim holds trivially.
- $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ holds and $secure_{P,\cong}(r, i \vdash_u \phi)$ does not hold. We have already shown that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r]\!]_{P,\cong}$. From $\neg secure_{P,\cong}(r, i \vdash_u \phi)$, it follows that there is $r' \in [\![r^i]\!]_{P,\cong}$ such that $[\phi]^{last(r^i).db} \neq [\phi]^{last(r').db}$. Let b the bijection showing that a preserves the equivalence class with respect to r, P, and $\cong$. Since $r' \in [\![r^i]\!]_{P,\cong}$, then let $r'' = b(r') = e(r', a)$. From $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$, it follows that $[\phi]^{last(r^i).db} \neq [\phi]^{last(e(r',a)).db}$. From this, $e(r^i, a) = r^{i+1}$, and $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ for all $r' \in [\![r]\!]_{P,\cong}$, it follows that $[\phi]^{last(r^{i+1}).db} \neq [\phi]^{last(e(r',a)).db}$. From this and $e(r', a) \in [\![r^{i+1}]\!]_{P,\cong}$, it follows $\neg secure_{P,\cong}(r, i+1 \vdash_u \phi)$. This contradicts the fact that $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ holds.

This completes the proof. □

Lemma C.19. *Let* $P = \langle M, f\rangle$ *be an extended configuration, where* $M = \langle D, \Gamma\rangle$ *is a system configuration and* f *is an* M*-PDP,* L *be the* P*-LTS,* u *be a user in* $\mathcal{U}$*,* r *be a run in* $traces(L)$*,* $a \in \mathcal{A}_{D,u}$ *be a* `SELECT` *or* `CREATE` *action,* ϕ *be a sentence,* $\cong$ *be an indistinguishability relation, and* i *be such that* $1 \leq i \leq |r|$*,* $triggers(last(r^i)) = \epsilon$*, and* $r^{i+1} = extend(r^i, a)$*. If* a *preserves the equivalence class for* r^i*,* P*, and* $\cong$*, then* $secure_{P,\cong}(r, i \vdash_u \phi)$ *holds iff* $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ *holds.*

Proof. The proof of this statement is similar to that of Lemma C.18. □

Lemma C.20. *Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, u be a user in $\mathcal{U}$, r be a run in $traces(L)$, $a \in \mathcal{A}_{D,u}$ be a* `GRANT` *or* `REVOKE` *action, ϕ be a sentence, $\cong$ be an indistinguishability relation, and i be such that $1 \leq i \leq |r|$, $triggers(last(r^i)) = \epsilon$, and $r^{i+1} = extend(r^i, a)$. If a preserves the equivalence class for r^i, P, and $\cong$, then $secure_{P,\cong}(r, i \vdash_u \phi)$ holds iff $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ holds.*

Proof. The proof of this statement is similar to that of Lemma C.18. □

Lemma C.21. *Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, u be a user in $\mathcal{U}$, r be a run in $traces(L)$, a be a trigger in $\mathcal{TRIGGER}_D$, ϕ be a sentence, $\cong$ be an indistinguishability relation, and i be such that $1 \leq i \leq |r|$, $invoker(last(r^i)) = u$, and $r^{i+1} = extend(r^i, a)$. If (1) a preserves the equivalence class for r^i, P, and $\cong$, (2) if a's action is either an* `INSERT` *or* `DELETE`*, then t's execution does not change any table in $tables(\phi)$ for any run $v \in [\![r^i]\!]_{P,\cong}$, and (3) $secEx(last(extend(v, a)) = \bot$ and $Ex(last(extend(v, a)) = \emptyset$ for any run $v \in [\![r^i]\!]_{P,\cong}$, then $secure_{P,\cong}(r, i \vdash_u \phi)$ holds iff $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ holds.*

Proof. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, u be a user in $\mathcal{U}$, r be a run in $traces(L)$, $\cong$ be an indistinguishability relation, a be a trigger in $\mathcal{TRIGGER}_D$, ϕ be a sentence, and i be such that $1 \leq i \leq |r|$, $invoker(last(r^i)) = u$, and $r^{i+1} = extend(r^i, a)$. Assume also that (1) a preserves the equivalence class for r^i, P, and $\cong$, and (2) $secEx(last(extend(r^i, a)) = \bot$ and $Ex(last(extend(r^i, a)) = \emptyset$. In the following, we denote the $extend$ function by e. Furthermore, we also denote the fact that $secure_{P,\cong}(r, i \vdash_u \phi)$ does not hold as $\neg secure_{P,\cong}(r, i \vdash_u \phi)$. From Definition C.1 and the fact that a preserves the equivalence class for r^i, P, and $\cong$, it follows that $e(r', a)$ is defined for any $r' \in [\![r^i]\!]_{P,\cong}$. Assume, for contradiction's sake, that our claim does not hold. There are two cases:

- $secure_{P,\cong}(r, i \vdash_u \phi)$ holds and $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ does not hold. From $secure_{P,\cong}(r, i \vdash_u \phi)$, it follows that $[\phi]^{last(r^i).db} = [\phi]^{last(r').db}$ for any $r' \in [\![r^i]\!]_{P,\cong}$. We claim that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$. From $\neg secure_{P,\cong}(r, i+1 \vdash_u \phi)$, it follows that there is a $r'' \in [\![r^{i+1}]\!]_{P,\cong}$ such that $[\phi]^{last(r'').db} \neq [\phi]^{last(r^{i+1}).db}$. Let b the bijection showing that a preserves the equivalence class with respect to r^i, P, and $\cong$. Since $r^{i+1} = e(r^i, a)$ and $r' \in [\![e(r, a)]\!]_{P,\cong}$, then there is a run $v \in [\![r^i]\!]_{P,\cong}$ such that $v = b^{-1}(r'')$. From this, $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$, and the fact that $[\phi]^{last(r'').db} \neq [\phi]^{last(r^{i+1}).db}$, it follows that $[\phi]^{last(v).db} \neq [\phi]^{last(r^{i+1}).db}$. From this, $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$, and $r^{i+1} = e(r^i, a)$, it follows $[\phi]^{last(v).db} \neq [\phi]^{last(r^i).db}$. This contradicts the fact that $[\phi]^{last(r^i).db} = [\phi]^{last(r').db}$ for any $r' \in [\![r^i]\!]_{P,\cong}$.
  We now prove that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$. Assume, for contradiction's sake, that there is a run $r' \in [\![r^i]\!]_{P,\cong}$ such that $[\phi]^{last(r').db} \neq [\phi]^{last(e(r',a)).db}$. Since executing the trigger does not throw security or integrity exceptions in any run $r' \in [\![r^i]\!]_{P,\cong}$, there are three cases:
  - the trigger a is not enabled in $e(r', a)$. From this and the LTS semantics, it follows that $last(r').db = last(e(r', a)).db$. From this, it therefore follows that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$. This contradicts our assumption.
  - the trigger a is enabled in $e(r', a)$ and its action is a `GRANT` or a `REVOKE` command. From this and the LTS semantics, it therefore follows that $last(r').db = last(e(r', a)).db$. From this, it thus follows that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$. This contradicts our assumption.
  - the trigger a is enabled in $e(r', a)$ and its action is a `INSERT` or a `DELETE` command. Thus, from $[\phi]^{last(r').db} \neq [\phi]^{last(e(r',a)).db}$, it follows that the content of one of the relations that determines ϕ is different in $last(r').db$ and $last(e(r', a)).db$. This contradicts the fact that the a's execution does not change the tables in $tables(\phi)$ for any run $r' \in [\![r^i]\!]_{P,\cong}$.
- $secure_{P,\cong}(r, i+1 \vdash_u \phi)$ holds and $secure_{P,\cong}(r, i \vdash_u \phi)$ does not hold. We have already shown that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r]\!]_{P,\cong}$. From $\neg secure_{P,\cong}(r, i \vdash_u \phi)$, it follows that there is $r' \in [\![r^i]\!]_{P,\cong}$ such that $[\phi]^{last(r^i).db} \neq [\phi]^{last(r').db}$. Let b the bijection showing that a preserves the equivalence class with respect to r, P, and $\cong$. Since $r' \in [\![r^i]\!]_{P,u}$, then let $r'' = b(r') = e(r', a)$. From $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r^i]\!]_{P,\cong}$, it follows that $[\phi]^{last(r^i).db} \neq [\phi]^{last(e(r',a)).db}$. From this, $e(r^i, a) = r^{i+1}$, and the fact that $[\phi]^{last(r').db} = [\phi]^{last(e(r',a)).db}$ holds for any $r' \in [\![r]\!]_{P,\cong}$, it follows that $[\phi]^{last(r^{i+1}).db} \neq [\phi]^{last(e(r',a)).db}$. From this and $e(r', a) \in [\![r^{i+1}]\!]_{P,\cong}$, it follows $\neg secure_{P,\cong}(r, i+1 \vdash_u \phi)$. This contradicts $secure_{P,\cong}(r, i+1 \vdash_u \phi)$.

This completes the proof. □

C.4.7 Auxiliary results about *getInfoV* and *getInfoS*

Proposition C.10 states that the *getInfoS* and *getInfoV* functions correctly capture the information an attacker may learn by an integrity exception (or its absence).

Proposition C.10. *Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, $a \in \mathcal{A}_{D,u}$ be an* `INSERT` *or* `DELETE` *action, and r be a run such that (1) $extend(r, a)$ is defined and (2) $secEx(last(extend(r, a))) = \bot$. For any constraint γ in $Dep(\Gamma, a)$, the following statements hold:*

- $[getInfoS(\gamma, a)]^{last(r).db} = \top$ *iff* $\gamma \notin Ex(last(extend(r, a)))$, *and*
- $[getInfoV(\gamma, a)]^{last(r).db} = \top$ *iff* $\gamma \in Ex(last(extend(r, a)))$.

Proof. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, $a \in \mathcal{A}_{D,u}$ be an `INSERT` or `DELETE` action, and and r be a run such that (1) $extend(r, a)$ is defined and (2) $secEx(last(extend(r, a))) = \bot$. Furthermore, let γ be a constraint in $Dep(\Gamma, a)$. We first note that $getInfoS(\gamma, a) = \neg getInfoV(\gamma, a)$. From this, it follows trivially that proving one of the two claims is enough. We thus prove that $[getInfoS(\gamma, a)]^{last(r).db} = \top$ iff $\gamma \notin Ex(last(extend(r, a)))$. There are two cases:

1. $a = \langle u, \texttt{INSERT}, R, \bar{t}\rangle$. There are two cases depending on γ:
   (a) γ is of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'.\, ((R(\overline{x}, \overline{y}, \overline{z}) \wedge R(\overline{x}, \overline{y}', \overline{z}')) \Rightarrow \overline{y} = \overline{y}')$. Let $\bar{t}$ be $(\overline{v}, \overline{w}, \overline{q})$, db be the state $last(r).db$, and db' be the state $db[R \oplus \bar{t}]$.

   (⇒). Assume that $[getInfoS(\gamma, a)]^{last(r).db} = \top$. From this and $getInfoS(\gamma, a)$'s definition, it follows that for all tuples $(\overline{v}, \overline{w}', \overline{q}') \in db(R)$, then $\overline{w}' = \overline{w}$. From a's definition and the LTS semantics, it follows that $db'(R) = db(R) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$. From this and the fact that for all tuples $(\overline{v}, \overline{w}', \overline{q}') \in db(R)$, then $\overline{w}' = \overline{w}$, it follows that for all tuples $(\overline{v}, \overline{w}', \overline{q}') \in db'(R)$, then $\overline{w}' = \overline{w}$. Furthermore, since $db \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{v}', \overline{w}', \overline{q}'), (\overline{v}'', \overline{w}'', \overline{q}'') \in db'(R)$, if $\overline{v}' = \overline{v}''$ and $\overline{v}' \neq \overline{v}$, then $\overline{w}' = \overline{w}$. Therefore, it follows that for all tuples $(\overline{v}', \overline{w}', \overline{q}')$, $(\overline{v}'', \overline{w}'', \overline{q}'') \in db'(R)$, if $\overline{v}' = \overline{v}''$, then $\overline{w}' = \overline{w}$. Therefore, $[\gamma]^{db'} = \top$. From this and the LTS semantics, it follows that $\gamma \notin Ex(last(extend(r, a)))$.

   (⇐). Assume that $\gamma \notin Ex(last(extend(r, a))))$. From this and the LTS semantics, it follows that $[\gamma]^{db'} = \top$. Therefore, for any two tuples $(\overline{v}', \overline{w}', \overline{q}')$ and $(\overline{v}'', \overline{w}'', \overline{q}'') \in db'(R)$, if $\overline{v}' = \overline{v}''$, then $\overline{w}' = \overline{w}$. Assume, for contradiction's sake, that $[getInfoS(\gamma, a)]^{db} = \bot$. This means that there is a tuple $(\overline{v}, \overline{w}', \overline{q}')$ in $db(R)$ such that $\overline{w}' \neq \overline{w}$. From $db' = db[R \oplus (\overline{v}, \overline{w}, \overline{q})]$ and the LTS semantics, it follows that both $(\overline{v}, \overline{w}', \overline{q}')$ and $(\overline{v}, \overline{w}, \overline{q})$ are in $db'(R)$. From this and $\overline{w}' \neq \overline{w}$, it follows that there are two tuples $(\overline{v}, \overline{w}, \overline{q})$ and $(\overline{v}, \overline{w}', \overline{q}')$ in $db(R)$ such that $\overline{w}' \neq \overline{w}$. From this and the relational calculus semantics, it follows that $[\gamma]^{db} = \bot$. This is in contradiction with $[\gamma]^{db'} = \top$.
   (b) γ is of the form $\forall \overline{x}, \overline{z}.\, R(\overline{x}, \overline{z}) \Rightarrow \exists \overline{w}.\, S(\overline{x}, \overline{w})$. Let $\bar{t}$ be $(\overline{v}, \overline{w})$, db be the state $last(r).db$, and db' be the state $db[R \oplus \bar{t}]$.

   (⇒). Assume that $[getInfoS(\gamma, a)]^{db} = \top$. From this and $getInfoS(\gamma, a)$'s definition, it follows that there is a tuple $(\overline{v}, \overline{y})$ in $db(S)$. From a's definition and the LTS semantics, it follows that $db'(S) = db(S)$. From this, it follows that there is a tuple $(\overline{v}, \overline{y})$ in $db'(S)$. Furthermore, since $db \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{v}', \overline{w}') \in db(R)$, if $\overline{v}' \neq \overline{v}$, there is a tuple $(\overline{v}', \overline{y}') \in db(S)$. From this and $\overline{db}' = db[R \oplus (\overline{v}, \overline{w})]$, it follows that for all tuples $(\overline{v}', \overline{w}') \in db'(R)$, there is a tuple $(\overline{v}', \overline{y}') \in db'(S)$. Therefore, $[\gamma]^{db'} = \top$. From this and the LTS semantics, it follows that $\gamma \notin Ex(last(extend(r, a)))$.

   (⇐). Assume that $\gamma \notin Ex(last(extend(r, a)))$. From this and the LTS semantics, it follows that $[\gamma]^{db'} = \top$. Therefore, for any tuple $(\overline{v}', \overline{w}') \in db'(R)$, there is a tuple $(\overline{v}', \overline{y}') \in db'(S)$. Assume, for contradiction's sake, that $[getInfoS(\gamma, a)]^{db} = \bot$. This means that for any tuple $(\overline{v}', \overline{y}')$ in $db(S)$, $\overline{v}' \neq \overline{v}$. From $db'(S) = db(S)$, it follows that for any tuple $(\overline{v}', \overline{y}')$ in $db'(S)$, $\overline{v}' \neq \overline{v}$. From $db' = db[R \oplus (\overline{v}, \overline{w})]$, it follows that there is a tuple $(\overline{v}, \overline{w})$ in $db'(R)$ such that there is no tuple $(\overline{v}, \overline{y}')$ in $db'(S)$. From this and the relational calculus semantics, it follows that $[\gamma]^{db} = \bot$. This is in contradiction with $[\gamma]^{db'} = \top$.
2. $a = \langle u, \texttt{DELETE}, R, \bar{t}\rangle$. In this case, γ is of the form $\forall \overline{x}, \overline{z}.\, S(\overline{x}, \overline{z}) \Rightarrow \exists \overline{w}.\, R(\overline{x}, \overline{w})$. Let $\bar{t}$ be $(\overline{v}, \overline{w})$, db be the state $last(r).db$, and db' be the state $db[R \ominus \bar{t}]$.

   (⇒). Assume that $[getInfoS(\gamma, a)]^{db} = \top$. From this and $getInfoS(\gamma, a)$'s definition, it follows that either there is no tuple $(\overline{v}, \overline{y})$ in $db(S)$ or there is a tuple $(\overline{v}, \overline{w}')$ in $db(R)$ such that $\overline{w}' \neq \overline{w}$. There are two cases:
   (a) there is no tuple $(\overline{v}, \overline{y})$ in $db(S)$. From this, a's definition, and the LTS semantics, it follows that there is no tuple $(\overline{v}, \overline{y})$ in $db'(S)$. From $db \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{v}', \overline{y}')$ in $db(S)$ such that $\overline{v}' \neq \overline{v}$, there is a tuple $(\overline{v}', \overline{w}')$ in $db(R)$. From this, $db'(R) = db(R) \setminus \{(\overline{v}, \overline{w})\}$, $db'(S) = db(S)$, and there is no tuple $(\overline{v}, \overline{y})$ in $db'(S)$, it follows that for all tuples

$(\overline{v}', \overline{y}')$ in $db(S)$, there is a tuple $(\overline{v}', \overline{w}')$ in $db(R)$. Therefore, $[\gamma]^{db'} = \top$. From this and the LTS semantics, it follows that $\gamma \notin Ex(last(extend(r, a)))$.

(b) there is a tuple $(\overline{v}, \overline{w}')$ in $db(R)$ such that $\overline{w}' \neq \overline{w}$. From this, a's definition, and the LTS semantics, it follows that there is a tuple $(\overline{v}, \overline{w}')$ in $db'(R)$ such that $\overline{w}' \neq \overline{w}$. From $db \in \Omega_D^\Gamma$, it follows that for all tuples $(\overline{v}', \overline{y}')$ in $db(S)$ such that $\overline{v}' \neq \overline{v}$, there is a tuple $(\overline{v}', \overline{w}'')$ in $db(R)$. From this, $db'(R) = db(R) \setminus \{(\overline{v}, \overline{w})\}$, $db'(S) = db(S)$, and there is a tuple $(\overline{v}, \overline{w}')$ in $db'(R)$ such that $\overline{w}' \neq \overline{w}$, it follows that for all tuples $(\overline{v}', \overline{y}')$ in $db(S)$, there is a tuple $(\overline{v}', \overline{w}')$ in $db(R)$. Therefore, $[\gamma]^{db'} = \top$. From this and the LTS semantics, it follows that $\gamma \notin Ex(last(extend(r, a)))$.

($\Leftarrow$). Assume that $\gamma \notin Ex(last(extend(r, a)))$. From this and the LTS semantics, it follows that $[\gamma]^{db'} = \top$. Therefore, for any tuple $(\overline{v}', \overline{y}') \in db'(S)$, there is a tuple $(\overline{v}', \overline{w}') \in db'(R)$. Assume, for contradiction's sake, that $[getInfoS(\gamma, a)]^{db} = \bot$. Therefore, there is a tuple $(\overline{v}, \overline{y})$ in $db(S)$ and for all tuples $(\overline{v}, \overline{w}'')$ in $db(R)$, $\overline{w}'' = \overline{w}$. From this, $db'(S) = db(S)$, and $db' = db[R \ominus (\overline{v}, \overline{w})]$, it follows that there is a tuple $(\overline{v}, \overline{y})$ in $db'(S)$ and for all tuples $(\overline{v}'', \overline{w}'')$ in $db'(R)$, $\overline{v}'' \neq \overline{v}$. From this and the relational calculus semantics, it follows that $[\gamma]^{db} = \bot$. This is in contradiction with $[\gamma]^{db'} = \top$.

This completes the proof. □

Proposition C.11 extends Proposition C.10 to triggers.

Proposition C.11. *Let $P = \langle M, f \rangle$ be an extended configuration, where $M = \langle D, \Gamma \rangle$ is a system configuration and f is an M-PDP, L be the P-LTS, $t \in \mathcal{TRIGGER}_D$ be a trigger whose action is an `INSERT` or `DELETE` command, and r be a run such that (1) $extend(r, t)$ is defined, (2) the trigger t is enabled, and (3) $secEx(last(extend(r, t))) = \bot$. Furthermore, we denote by a the actual action performed by the trigger t. For any constraint γ in $Dep(\Gamma, a)$, the following statements hold:*

- $[getInfoS(\gamma, a)]^{last(r).db} = \top$ *iff* $\gamma \notin Ex(last(extend(r, t)))$, *and*
- $[getInfoV(\gamma, a)]^{last(r).db} = \top$ *iff* $\gamma \in Ex(last(extend(r, t)))$.

Proof. The proof of this statement is almost identical to that of Proposition C.10. □

C.4.8 Auxiliary results about f

Lemma C.22 states that the PDP f returns the same result in any two data-indistinguishable states.

Lemma C.22. *Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s, s' \in \Omega_M$ be two M-states such that $sysState(s) \cong^{data}_{M,user(a,s)} sysState(s')$, $tuple(s) = tuple(s')$, $invoker(s) = invoker(s')$, and $trigger(s) = trigger(s')$, and f be the PDP as above. The following conditions hold:*

1. *If $trigger(s) = \epsilon$, then $f(s, a) = \top$ iff $f(s', a) = \top$ for any action a in $\mathcal{A}_{D,\mathcal{U}}$.*
2. *If $trigger(s) \in \mathcal{TRIGGER}_D$, then $f(s, trigCond(s)) = \top$ iff $f(s', trigCond(s)) = \top$.*
3. *If $trigger(s) \in \mathcal{TRIGGER}_D$, $trigCond(s) = \langle u, \texttt{SELECT}, \psi \rangle$, and $[\psi]^{s.db} = [\psi]^{s'.db} = \top$, then $f(s, trigAct(s)) = \top$ iff $f(s', trigAct(s')) = \top$.*

Proof. We prove our three claims by contradiction.

1. Assume, for contradiction's sake, that there are two states s and s' and an action a such that $trigger(s) = trigger(s') = \epsilon$, $sysState(s) \cong^{data}_{M,user(a,s)} sysState(s')$, $f(s, a) = \top$, and $f(s', a) = \bot$. From f's definition, $f(s, a) = \top$, $f(s', a) = \bot$, and Lemma C.17, it follows that $f_{int}(s, a) = \top$, $f_{int}(s', a) = \bot$, and $f_{conf}^{user(a,s)}(s, a) = f_{conf}^{user(a,s')}(s', a) = \top$. From $f_{int}(s', a) = \bot$, it follows that $s' \not\rightsquigarrow^{approx}_{auth} a$. From $f_{int}(s, a) = \top$, it follows $s \rightsquigarrow^{approx}_{auth} a$. From this, $a \in \mathcal{A}_{D,\mathcal{U}}$, and Lemma C.1, it follows $s' \rightsquigarrow^{approx}_{auth} a$, which contradicts $s' \not\rightsquigarrow^{approx}_{auth} a$. This completes the proof for the first claim.
2. Assume, for contradiction's sake, that there are two states s and s' such that $trigger(s) = trigger(s')$, $trigger(s) \neq \epsilon$, $sysState(s) \cong^{data}_{M,user(a,s)} sysState(s')$, $f(s, a) = \top$, and $f(s', a) = \bot$, where $trigCond(s) = trigCond(s') = a$ (this follows from $tuple(s) = tuple(s')$, $invoker(s) = invoker(s')$, and $trigger(s) = trigger(s')$). From f's definition, $f(s, a) = \top$, $f(s', a) = \bot$, and Lemma C.17, it follows that $f_{int}(s, a) = \top$, $f_{int}(s', a) = \bot$, and $f_{conf}^{user(a,s)}(s, a) = f_{conf}^{user(a,s')}(s', a) = \top$. From f_{int}'s definition, $trigger(s') \neq \epsilon$, and $a = trigCond(s')$, it follows that $f_{int}(s', a) = \top$, which contradicts $f_{int}(s', a) = \bot$. This completes the proof for the second claim.
3. Assume, for contradiction's sake, that there are two states s and s' such that $trigger(s) = trigger(s') = t$, $trigger(s) \neq \epsilon$, $sysState(s) \cong^{data}_{M,user(a,s)} sysState(s')$, $[\psi]^{s.db} = [\psi]^{s'.db} = \top$, $f(s, a) = \top$, and $f(s', a) = \bot$, where $a = trigAct(s) = trigAct(s')$ (this follows from $tuple(s) = tuple(s')$, $invoker(s) = invoker(s')$, and $trigger(s) = trigger(s')$). From f's definition, $f(s, a) = \top$, $f(s', a) = \bot$, and Lemma C.17, it follows that $f_{conf}^{user(a,s)}(s, a) = f_{conf}^{user(a,s')}(s', a) = \top$,

$f_{int}(s,a) = \top$, and $f_{int}(s',a) = \bot$. From this, it follows that $s' \not\rightsquigarrow^{approx}_{auth} t$. From $f_{int}(s,a) = \top$, it follows $s \rightsquigarrow^{approx}_{auth} t$. There are two cases depending on t's security mode:

(a) $mode(t) = A$. From this and $s \rightsquigarrow^{approx}_{auth} t$, it follows that $s \rightsquigarrow^{approx}_{auth} a$ and $s \rightsquigarrow^{approx}_{auth} a'$, where $a' = action(statement(t), owner(t), tuple(s))$ is the trigger's action associated with the trigger's owner. Note that s and s' are data indistinguishable. From this, $a, a' \in \mathcal{A}_{D,\mathcal{U}}$, and Lemma C.1, it follows that $s' \rightsquigarrow^{approx}_{auth} a$ and $s' \rightsquigarrow^{approx}_{auth} a'$. From $s' \rightsquigarrow^{approx}_{auth} a$, $s' \rightsquigarrow^{approx}_{auth} a'$, $[\psi]^{s'.db} = \top$, and the rule `EXECUTE TRIGGER - 2`, it follows that $s' \rightsquigarrow^{approx}_{auth} t$, which contradicts $s' \not\rightsquigarrow^{approx}_{auth} t$.

(b) $mode(t) = O$. From this and $s \rightsquigarrow^{approx}_{auth} t$, it follows that $s \rightsquigarrow^{approx}_{auth} a$. Note that s and s' are data indistinguishable. From this, $a, a' \in \mathcal{A}_{D,\mathcal{U}}$, and Lemma C.1, it follows that $s' \rightsquigarrow^{approx}_{auth} a$. From this, $[\psi]^{s'.db} = \top$, and the rule `EXECUTE TRIGGER - 1`, it follows that $s' \rightsquigarrow^{approx}_{auth} t$, which contradicts $s' \not\rightsquigarrow^{approx}_{auth} t$.

This completes the proof for the third claim.

This completes the proof. □

C.4.9 f preserves the equivalence class

In Lemmas C.23 and C.24, we prove that, whenever we use f as PDP, actions and triggers preserve the equivalence class.

Lemma C.23. *Let u be a user in $\mathcal{U}$, $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is the PDP from Section 6.8, and L be the P-LTS. For any run $r \in traces(L)$ and any action $a \in \mathcal{A}_{D,u}$, if $extend(r,a)$ is defined, then a preserves the equivalence class for r, P, and $\cong_{P,u}$.*

Proof. Let u be a user in $\mathcal{U}$, $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is the PDP from Section 6.8, and L be the P-LTS. In the following, we use e to refer to the *extend* function. We prove our claim by contradiction. Assume, for contradiction's sake, that there is a run $r \in traces(L)$ and an action $a \in \mathcal{A}_{D,u}$ such that $e(r,a)$ is defined and a does not preserve the equivalence class for r, P, and $\cong_{P,u}$. According to the LTS semantics, the fact that $e(r,a)$ is defined implies that $triggers(last(r)) = \epsilon$. Therefore, $triggers(last(r')) = \epsilon$ holds as well for any for any $r' \in [\![r]\!]_{P,u}$ (because r and r' are indistinguishable and, therefore, their projections are consistent), and, thus, $e(r',a)$ is defined as well for any $r' \in [\![r]\!]_{P,u}$. There are a number of cases depending on a:

1. $a = \langle u, \texttt{SELECT}, q\rangle$. There are two cases:
   (a) $secEx(last(e(r,a))) = \bot$. From the LTS rules and $secEx(last(e(r,a))) = \bot$, it follows that $f(last(r), a) = \top$. From this and Lemma C.22, it follows that $f(last(r'), a) = \top$ for any $r' \in [\![r]\!]_{P,u}$. From this and the LTS rules, it follows $secEx(last(e(r',a))) = \bot$ for any $r' \in [\![r]\!]_{P,u}$. From $f(last(r'), a) = \top$ for any $r' \in [\![r]\!]_{P,u}$, it follows $f^{user(last(r'),a)}_{conf}(last(r'), a) = \top$ for any $r' \in [\![r]\!]_{P,u}$. Note that $user(last(r'), a) = u$ for any $r' \in [\![r]\!]_{P,u}$ because $trigger(last(r')) = \epsilon$ and $u \in \mathcal{A}_{D,u}$. From this, $f^{user(last(r'),a)}_{conf}(last(r'), a) = \top$ for any $r' \in [\![r]\!]_{P,u}$, and f^{u}_{conf}'s definition, it follows that $secure(u, q, last(r')) = \top$ for any $r' \in [\![r]\!]_{P,u}$. From this and Proposition C.9, it follows that $[q]^{last(r').db} = [q]^{last(r).db}$ for all $r' \in [\![r]\!]_{P,u}$. Furthermore, it follows trivially from the LTS rule *`SELECT` Success*, that the state after a's execution is data indistinguishable from $last(r)$. It is also easy to see that $e(r',a)$ is well-defined for any $r' \in [\![r]\!]_{P,u}$. From the considerations above and $r' \in [\![r]\!]_{P,u}$, it follows trivially that $e(r',a) \in [\![e(r,a)]\!]_{P,u}$. The bijection b is trivially $b(r') = e(r',a)$. This leads to a contradiction.
   (b) $secEx(last(e(r,a))) = \top$. From the LTS rules and $secEx(last(e(r,a))) = \top$, it follows that $f(last(r), a) = \bot$. From this and Lemma C.22, it follows that $f(last(r'), a) = \bot$ for any $r' \in [\![r]\!]_{P,u}$. From this and the LTS rules, it follows $secEx(last(e(r',a))) = \top$ for any $r' \in [\![r]\!]_{P,u}$. The data indistinguishability between $last(e(r',a))$ and $last(e(r,a))$ follows trivially from the data indistinguishability between $last(r')$ and $last(r)$. Therefore, for any run $r' \in [\![r]\!]_{P,u}$, there is exactly one run $e(r',a)$. From the considerations above, it follows trivially that $e(r',a) \in [\![e(r,a)]\!]_{P,u}$. The bijection b is trivially $b(r') = e(r',a)$. This leads to a contradiction.

   Both cases leads to a contradiction. This completes the proof for $a = \langle u, \texttt{SELECT}, q\rangle$.
2. $a = \langle u, \texttt{INSERT}, R, \bar{t}\rangle$. In the following, we denote by gI the function *getInfo*, by gS the function *getInfoS*, and by gV the function *getInfoV*. There are three cases:
   (a) $secEx(last(e(r,a))) = \bot$ and $Ex(last(e(r,a))) = \emptyset$. From the LTS rules and $secEx(last(e(r,a))) = \bot$, it follows that $f(last(r), a) = \top$. From this and Lemma C.22, it follows that $f(last(r'), a) = \top$ for any $r' \in [\![r]\!]_{P,u}$. From this and the LTS rules, it follows that $secEx(last(e(r',a))) = \bot$ for any $r' \in [\![r]\!]_{P,u}$. From $f(last(r), a) = \top$, it follows

that $f^u_{conf}(last(r), a) = \top$ because $user(last(r), a) = u$ since $trigger(last(r), a) = \epsilon$ and $a \in \mathcal{A}_{D,u}$. From this and f^u_{conf}'s definition, it follows that $secure(u, gS(\gamma, act), last(r)) = \top$ for any integrity constraint γ in $Dep(\Gamma, a)$. From $Ex(last(e(r, a))) = \emptyset$ and Proposition C.10, it follows $[gS(\gamma, act)]^{last(r).db} = \top$. From this, $secure(u, gS(\gamma, act), last(r))$, and Proposition C.9, it follows that $[gS(\gamma, act)]^{last(r').db} = \top$ for any $r' \in [\![r]\!]_{P,u}$. From this and Proposition C.10, it follows that $Ex(last(e(r', a))) = \emptyset$ for any $r' \in [\![r]\!]_{P,u}$. We claim that, for any $r' \in [\![r]\!]_{P,u}$, $last(e(r, a))$ and $last(e(r', a))$ are data indistinguishable. From this and the above considerations, it follows trivially that $e(r', a) \in [\![e(r, a)]\!]_{P,u}$. The bijection b is trivially $b(r') = e(r', a)$. This leads to a contradiction.

We now prove our claim that for any $r' \in [\![r]\!]_{P,u}$, $last(e(r, a))$ and $last(e(r', a))$ are data indistinguishable. We prove the claim by contradiction. Let $s_2 = \langle db_2, U_2, sec_2, T_2, V_2\rangle$ be $sysState(last(e(r, a)))$, $s'_2 = \langle db'_2, U'_2, sec'_2, T'_2, V'_2\rangle$ be $sysState(last(e(r', a)))$, $s_1 = \langle db_1, U_1, sec_1, T_1, V_1\rangle$ be $sysState(last(r))$, and $s'_1 = \langle db'_1, U'_1, sec'_1, T'_1, V'_1\rangle$ be $sysState(last(r'))$. In the following, we denote the *permissions* function by p. Furthermore, note that s_1 and s'_1 are data-indistinguishable because $r' \in [\![r]\!]_{P,u}$. There are a number of cases:

i. $U_2 \neq U'_2$. Since a is an `INSERT` operation, it follows that $U_1 = U_2$ and $U'_1 = U'_2$. Furthermore, from $s_1 \cong^{data}_{M,u} s'_1$, it follows that $U_1 = U'_1$. Therefore, $U_2 = U'_2$ leading to a contradiction.

ii. $sec_2 \neq sec'_2$. The proof is similar to the case $U_2 \neq U'_2$.

iii. $T_2 \neq T'_2$. The proof is similar to the case $U_2 \neq U'_2$.

iv. $V_2 \neq V'_2$. The proof is similar to the case $U_2 \neq U'_2$.

v. there is a table R' for which $\langle \oplus, \texttt{SELECT}, R\rangle \in p(s_2, u)$ and $db_2(R') \neq db'_2(R')$. Note that $p(s_2, u) = p(s_1, u)$. There are two cases:
   - $R = R'$. From $s_1 \cong^{data}_{M,u} s'_1$ and $\langle \oplus, \texttt{SELECT}, R\rangle \in p(s_2, u)$, it follows that $db_1(R') = db'_1(R')$. From this and the fact that a has been executed successfully both in $e(r, a)$ and $e(r', a)$, it follows that $db_2(R') = db_1(R') \cup \{\bar{t}\}$ and $db'_2(R') = db'_1(R') \cup \{\bar{t}\}$. From this and $db_1(R') = db'_1(R')$, it follows that $db_2(R') = db'_2(R')$ leading to a contradiction.
   - $R \neq R'$. From $s_1 \cong^{data}_{M,u} s'_1$ and $\langle \oplus, \texttt{SELECT}, R\rangle \in p(s_2, u)$, it follows that $db_1(R') = db'_1(R')$. From this and the fact that a does not modify R', it follows that $db_1(R') = db_2(R')$ and $db'_1(R') = db'_2(R')$. From this and $db_1(R') = db'_1(R')$, it follows that $db_2(R') = db'_2(R')$ leading to a contradiction.

vi. there is a view v for which $\langle \oplus, \texttt{SELECT}, v\rangle \in p(s_2, u)$ and $db_2(v) \neq db'_2(v)$. Note that $p(s_2, u) = p(s_1, u)$. Since a has been successfully executed in both states, we know that $leak(s_1, a, u)$ holds. There are two cases:
   - $R \notin tDet(v, s, M)$. Then, $v(s_1) = v(s_2)$ and $v(s'_1) = v(s'_2)$ (because R's content does not determine v's materialization). From $s_1 \cong^{data}_{M,u} s'_1$ and the fact that a modifies only R, it follows that $v(db_2) = v(db'_2)$ leading to a contradiction.
   - $R \in tDet(v, s, M)$ and for all $o \in tDet(v, s, M)$, $\langle \oplus, \texttt{SELECT}, o\rangle \in p(s_1, u)$. From this and $s_1 \cong^{data}_{M,u} s'_1$, it follows that, for all $o \in tDet(v, s, M)$, $o(s_1) = o(s'_1)$. If $o \neq R$, $o(s_1) = o(s'_1) = o(s_2) = o(s'_2)$. From $\langle \oplus, \texttt{SELECT}, R\rangle \in p(s_1, u)$ and $s_1 \cong^{data}_{M,u} s'_1$, it follows that $db_1(R) = db'_1(R)$. From this and the fact that a has been executed successfully both in $e(r, a)$ and $e(r', a)$, it follows that $db_2(R) = db_1(R) \cup \{\bar{t}\}$ and $db'_2(R) = db'_1(R) \cup \{\bar{t}\}$. From this and $db_1(R) = db'_1(R)$, it follows that $db_2(R) = db'_2(R)$. From this and for all $o \in tDet(v, s, M)$ such that $o \neq R$, $o(s_2) = o(s'_2)$, it follows that for all $o \in tDet(v, s, M)$, $o(s_2) = o(s'_2)$. Since the content of all tables determining v is the same in s_2 and s'_2, it follows that $db_2(v) = db'_2(v)$ leading to a contradiction.

All the cases lead to a contradiction.

(b) $secEx(last(e(r, a))) = \bot$ and $Ex(last(e(r, a))) \neq \emptyset$. From the LTS rules and $secEx(last(e(r, a))) = \bot$, it follows that $f(last(r), a) = \top$. From this and Lemma C.22, it follows that $f(last(r'), a) = \top$ for any $r' \in [\![r]\!]_{P,u}$. From this and the LTS rules, it follows that $secEx(last(e(r', a))) = \bot$ for any $r' \in [\![r]\!]_{P,u}$. Assume that the exception has been caused by the constraint γ, i.e., $\gamma \in Ex(last(e(r, a)))$. From this and Proposition C.10, it follows that $gV(\gamma, a)$ holds in $last(r).db$. From $f(last(r), a) = \top$ and f's definition, it follows that $f^u_{conf}(last(r), a) = \top$ because $user(last(r), a) = u$ since $trigger(last(r)) = \epsilon$ and $a \in \mathcal{A}_{D,u}$. From this and f^u_{conf}'s definition, it follows that $secure(u, gV(\gamma, a), last(r))$ holds. From this, Proposition C.9, and $[gV(\gamma, a)]^{last(r).db} = \top$, it follows that also $[gV(\gamma, act)]^{last(r').db} = \top$ for any $r' \in [\![r]\!]_{P,u}$. From this and Proposition C.10, it follows that $\gamma \in Ex(last(e(r', a)))$ for any $r' \in [\![r]\!]_{P,u}$. The data indistinguishability between $last(e(r, a))$ and $last(e(r', a))$ follows trivially from the data indistinguishability between $last(r)$ and $last(r')$ for any $r' \in [\![r]\!]_{P,u}$. Therefore, for any run $r' \in [\![r]\!]_{P,u}$, there is exactly one run $e(r', a)$. From

the considerations above, it follows trivially that $e(r', a) \in [\![e(r, a)]\!]_{P,u}$. The bijection b is trivially $b(r') = e(r', a)$. This leads to a contradiction.

(c) $secEx(last(e(r, a))) = \top$. From the LTS rules and $secEx(last(e(r, a))) = \top$, it follows that $f(last(r), a) = \bot$. From this and Lemma C.22, it follows that $f(last(r'), a) = \bot$ for any $r' \in [\![r]\!]_{P,u}$. From this and the LTS rules, it follows $secEx(last(e(r', a))) = \top$ for any $r' \in [\![r]\!]_{P,u}$. The data indistinguishability between $last(e(r, a))$ and $last(e(r', a))$ follows trivially from that between $last(r)$ and $last(r')$ for any $r' \in [\![r]\!]_{P,u}$. Therefore, for any run $r' \in [\![r]\!]_{P,u}$, there is exactly one run $e(r', a)$. From the considerations above, it follows trivially that $e(r', a) \in [\![e(r, a)]\!]_{P,u}$. The bijection b is trivially $b(r') = e(r', a)$. This leads to a contradiction.

All cases lead to a contradiction. This completes the proof for $a = \langle u, \texttt{INSERT}, R, \bar{t}\rangle$.

3. $a = \langle u, \texttt{DELETE}, R, \bar{t}\rangle$. The proof is similar to that for $a = \langle u, \texttt{INSERT}, R, \bar{t}\rangle$.
4. $a = \langle \oplus, u', p, u\rangle$. There are two cases:
   (a) $secEx(last(e(r, a))) = \bot$. We assume that $p = \langle \texttt{SELECT}, O\rangle$ for some $O \in D \cup V$. If this is not the case, the proof is trivial. Furthermore, we also assume that $u' = u$, otherwise the proof is, again, trivial since the new permission does not influence u's permissions. From the LTS rules and $secEx(last(e(r, a))) = \bot$, it follows that $f(last(r), a) = \top$. From this and Lemma C.22, it follows that $f(last(r'), a) = \top$ for any $r' \in [\![r]\!]_{P,u}$. From this and the LTS rules, it follows that $secEx(last(e(r', a))) = \bot$ for any $r' \in [\![r]\!]_{P,u}$. From $secEx(last(e(r', a))) = \bot$ and f^{u}_{conf}'s definition, it follows that $last(r').sec = last(e(r', a)).sec$. Therefore, since $last(r)$ and $last(r')$ are data indistinguishable, for any $r' \in [\![r]\!]_{P,u}$, then also $last(e(r, a))$ and $last(e(r', a))$ are data indistinguishable. Therefore, for any run $r' \in [\![r]\!]_{P,u}$, there is exactly one run $e(r', a)$. From the considerations above, it follows trivially that $e(r', a) \in [\![e(r, a)]\!]_{P,u}$. The bijection b is trivially $b(r') = e(r', a)$. This leads to a contradiction.
   (b) $secEx(last(e(r, a))) = \top$. From the LTS rules and $secEx(last(e(r, a))) = \top$, it follows $f(last(r), a) = \bot$. From this and Lemma C.22, it follows that $f(last(r'), a) = \bot$ for any $r' \in [\![r]\!]_{P,u}$. From this and the LTS rules, it follows $secEx(last(e(r', a))) = \top$ for any $r' \in [\![r]\!]_{P,u}$. The data indistinguishability between $last(e(r', a))$ and $last(e(r, a))$ follows trivially from the data indistinguishability between $last(r')$ and $last(r)$. Therefore, for any run $r' \in [\![r]\!]_{P,u}$, there is exactly one run $e(r', a)$. From the considerations above, it follows trivially $e(r', a) \in [\![e(r, a)]\!]_{P,u}$. The bijection b is trivially $b(r') = e(r', a)$. This leads to a contradiction.

   Both cases lead to a contradiction. This completes the proof for $a = \langle \oplus, u', p, u\rangle$.
5. $a = \langle \oplus^*, u', p, u\rangle$. The proof is similar to that for $a = \langle \oplus, u', p, u\rangle$.
6. $a = \langle \ominus, u', p, u\rangle$. The proof is similar to that for $a = \langle u, \texttt{SELECT}, q\rangle$. The only difference is in proving that for any $r' \in [\![r]\!]_{P,u}$, $last(e(r, a))$ and $last(e(r', a))$ are data indistinguishable. Assume, for contradiction's sake, that this is not the case. Let $s_2 = \langle db_2, U_2, sec_2, T_2, V_2\rangle$ be $sysState(last(e(r, a)))$ and $s_2' = \langle db_2', U_2', sec_2', T_2', V_2'\rangle$ be $sysState(last(e(r', a)))$. Furthermore, let $s_1 = \langle db_1, U_1, sec_1, T_1, V_1\rangle$ be $sysState(last(r))$ and $s_1' = \langle db_1', U_1', sec_1', T_1', V_1'\rangle$ be $sysState(last(r'))$. In the following, we denote the *permissions* function by p. Furthermore, note that s_1 and s_1' are data-indistinguishable because $r' \in [\![r]\!]_{P,u}$. There are a number of cases:
   (a) $U_2 \neq U_2'$. Since a is an `REVOKE` operation, it follows that $U_1 = U_2$ and $U_1' = U_2'$. Furthermore, from $s_1 \cong^{data}_{M,u} s_1'$, it follows that $U_1 = U_1'$. Therefore, $U_2 = U_2'$ leading to a contradiction.
   (b) $sec_2 \neq sec_2'$. From $s_1 \cong^{data}_{M,u} s_1'$, it follows that $sec_1 = sec_1'$. From a's definition and the LTS rules, it follows that $sec_2 = revoke(sec_1, u', p, u)$ and $sec_2' = revoke(sec_1', u', p, u)$. From this and $sec_1 = sec_1'$, it follows that $sec_2 = sec_2'$ leading to a contradiction.
   (c) $T_2 \neq T_2'$. The proof is similar to the case $U_2 \neq U_2'$.
   (d) $V_2 \neq V_2'$. The proof is similar to the case $U_2 \neq U_2'$.
   (e) there is a table R for which $\langle \oplus, \texttt{SELECT}, R\rangle \in p(s_2, u)$ and $db_2(R) \neq db_2'(R)$. Since a is an `REVOKE` operation, it follows that $db_1 = db_2$ and $db_1' = db_2'$. Furthermore, from $s_1 \cong^{data}_{M,u} s_1'$, it follows that $db_1(R) = db_1'(R)$. From this, $db_1 = db_2$, and $db_1' = db_2'$, it follows that $db_2(R) = db_2'(R)$ leading to a contradiction.
   (f) there a view v for which $\langle \oplus, \texttt{SELECT}, v\rangle \in p(s_2, u)$ and $db_2(v) \neq db_2'(v)$. Since a is an `REVOKE` operation, it follows that $db_1 = db_2$ and $db_1' = db_2'$. Furthermore, from $s_1 \cong^{data}_{M,u} s_1'$, it follows that $db_1(v) = db_1'(v)$. From this, $db_1 = db_2$, and $db_1' = db_2'$, it follows that $db_2(v) = db_2'(v)$ leading to a contradiction.

   All the cases lead to a contradiction.
7. $a = \langle u, \texttt{CREATE}, o\rangle$. The proof is similar to that for $a = \langle \ominus, u', p, u\rangle$.
8. $a = \langle u, \texttt{ADD\_USER}, u'\rangle$. The proof is similar to that for $a = \langle \ominus, u', p, u\rangle$.

This completes the proof. □

Lemma C.24. *Let u be a user in $\mathcal{U}$, $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is the PDP from Section 6.8, and L be the P-LTS. For any run $r \in traces(L)$ such that $invoker(last(r)) = u$ and any trigger $t \in \mathcal{TRIGGER}_D$, if $extend(r, t)$ is defined, then t preserves the equivalence class for r, M, and $\cong_{P,u}$.*

Proof. Let u be a user in $\mathcal{U}$, $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is the PDP from Section 6.8, and L be the P-LTS. In the following, we use e to refer to the *extend* function. The proof in cases where the trigger t is not enabled is similar to the proof of the `SELECT` case of Lemma C.23. In the following, we therefore assume that the trigger t is enabled. We also assume that its `WHEN` condition is secure. We handle the case of triggers with an insecure `WHEN` condition separately. We prove our claim by contradiction. Assume, for contradiction's sake, that there is a run $r \in traces(L)$ such that $invoker(last(r)) = u$ and a trigger t such that $e(r, t)$ is defined and t does not preserve the equivalence class for r, P, and $\cong_{P,u}$. Since $invoker(last(r)) = u$ and $e(r, t)$ is defined, then $e(r', t)$ is defined as well for any $r' \in [\![r]\!]_{P,u}$ (indeed, from $invoker(last(r)) = u$, it follows that the last action in r is either an action issued by u or a trigger invoked by u. From this, the fact that $e(r, t)$ is defined, and the fact that r and r' are indistinguishable, it follows that $trigger(last(r)) = trigger(last(r')) = t$). Let a be t's action and $w = \langle u', \texttt{SELECT}, q\rangle$ be the `SELECT` command associated with t's `WHEN` condition. Let s be the state $last(r)$, s' be the state obtained just after the execution of the `WHEN` condition, and s'' be the state $last(e(r, t))$. There are a number of cases depending on t's action a:

1. $a = \langle u', \texttt{INSERT}, R, \overline{t}\rangle$. There are three cases:
   (a) $secEx(last(e(r, a))) = \bot$ and $Ex(last(e(r, a))) = \emptyset$. The proof of this case is similar to that of the corresponding case in Lemma C.23.
   (b) $secEx(last(e(r, a))) = \bot$ and $Ex(last(e(r, a))) \neq \emptyset$. The only difference between the proof of this case with respect to the corresponding case in Lemma C.23 is that we have to establish again the data indistinguishability between $last(e(r, t))$ and $last(e(r', t))$. Indeed, for triggers the roll-back state is, in general, different from the one immediately before the trigger's execution, i.e., it may be that $sysState(last(e(r, t))) \neq sysState(last(r))$. We now prove that $last(e(r, t))$ and $last(e(r', t))$ are data indistinguishable. From the LTS semantics, it follows that $r = p{\cdot}s_0{\cdot}\langle invoker(last(r)), op, R', \overline{v}\rangle{\cdot}s_1{\cdot}t_1{\cdot}\ldots{\cdot}s_{n-1}{\cdot}t_n{\cdot}s_n$, where $p \in traces(L)$ and $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$. Similarly, $r' = p'{\cdot}s'_0{\cdot}\langle invoker(last(r)), op, R', \overline{v}\rangle{\cdot}s'_1{\cdot}t_1{\cdot}\ldots{\cdot}s'_{n-1}{\cdot}t_n{\cdot}s'_n$, where $p' \in traces(L)$, $p \cong_{P,u} p'$, and all states s_i and s'_i are data indistinguishable. Then, the roll-back states are, respectively, s_0 and s'_0, which are data indistinguishable. From the LTS rules, $last(e(r, a)) = s_0$ and $last(e(r', a)) = s'_0$. Therefore, the data indistinguishability between $last(e(r, a))$ and $last(e(r', a))$ follows trivially for any $r' \in [\![r]\!]_{P,u}$.
   (c) $secEx(e(r, a)) = \top$. The proof is similar to the previous case.
   All cases lead to a contradiction. This completes the proof for $a = \langle u', \texttt{INSERT}, R, \overline{t}\rangle$.
2. $a = \langle u', \texttt{DELETE}, R, \overline{t}\rangle$. The proof is similar to that for $a = \langle u', \texttt{INSERT}, R, \overline{t}\rangle$.
3. $a = \langle \oplus, u'', p, u'\rangle$. There are two cases:
   (a) $secEx(last(e(r, a))) = \bot$. In this case, the proof is similar to the corresponding case in Lemma C.23.
   (b) $secEx(last(e(r, a))) = \top$. The proof is similar to the $secEx(last(e(r, a))) = \top$ case of $a = \langle u', \texttt{INSERT}, R, \overline{t}\rangle$.
   Both cases lead to a contradiction. This completes the proof for $a = \langle \oplus, u'', p, u'\rangle$.
4. $a = \langle \oplus^*, u'', p, u'\rangle$. The proof is similar to that for $a = \langle \oplus, u'', p, u'\rangle$.
5. $a = \langle \ominus, u'', p, u'\rangle$. The proof is similar to that for $a = \langle u', \texttt{INSERT}, R, \overline{t}\rangle$.

This completes the proof.

We now consider the case of a trigger whose `WHEN` condition is not secure. From this, it follows that f blocks the trigger's execution and throws a security exception. Observe that this happens for any $r' \in [\![r]\!]_{P,u}$ (see Lemma C.22). We thus only need to prove that the roll-back states are data indistinguishable for any $r' \in [\![r]\!]_{P,u}$. We already proved this in the case 1.(b) above. This completes the proof. □

C.4.10 f provides Data Confidentiality

In Theorem C.2, we finally prove the main result of this section, namely that f provides data confidentiality.

Theorem C.2. *Let M be a system configuration, f be the PDP from Section 6.8, and $P = \langle M, f\rangle$ be an extended configuration. For any user $u \in \mathcal{U}$, the PDP f provides data confidentiality with respect to P, u, $\mathcal{ATK}_u$, and $\cong_{P,u}$.*

Proof. Let u be a user in $\mathcal{U}$, $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ is a system configuration and f is the PDP from Section 6.8, and L be the P-LTS. Furthermore, let r be a run in $traces(L)$, i be an integer such that $1 \leq i \leq |r|$, and ϕ be a sentence such that $r, i \vdash_u \phi$ holds. We claim that $secure_{P,u}(r, i \vdash_u \phi)$ holds. The theorem follows trivially from the claim.

We now show that for all $r \in traces(L)$, all i such that $1 \leq i \leq |r|$, and all sentences ϕ such that $r, i \vdash_u \phi$ holds, then $secure_{P,u}(r, i \vdash_u \phi)$ holds as well. We prove our claim by induction on the length of the derivation $r, i \vdash_u \phi$. In the following, we denote by e the function *extend*.

Base Case: Assume that $|r, i \vdash_u \phi| = 1$. There are a number of cases depending on the rule used to obtain $r, i \vdash_u \phi$.

1. *SELECT Success - 1.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{SELECT}, \phi\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = s'$, and $s' = \langle db, U, sec, T, V, c'\rangle$. From the rules, it follows that $f(s', \langle u, \texttt{SELECT}, \phi\rangle) = \top$. From this and f's definition, it follows that $f_{int}(s', \langle u, \texttt{SELECT}, \phi\rangle) = \top$ and $f^u_{conf}(s', \langle u, \texttt{SELECT}, \phi\rangle) = \top$, because $user(s', \langle u, \texttt{SELECT}, \phi\rangle) = u$. From $f^u_{conf}(s', \langle u, \texttt{SELECT}, \phi\rangle) = \top$, it follows $secure(u, \phi, s') = \top$. From this, Lemma C.14, and $sysState(s) = sysState(s')$, it follows $secure(u, \phi, s) = \top$. From this, Proposition C.9, and $last(r^i) = s$, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds.
2. *SELECT Success - 2.* The proof for this case is similar to that of *SELECT Success - 1.*
3. *INSERT Success.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \bar{t}\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $R(\bar{t})$. Then, $secure_{P,u}(r, i \vdash_u R(\bar{t}))$ holds. Indeed, in all runs r' (P, u)-indistinguishable from r^i the last action is $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$. Furthermore, the action has been executed successfully. Therefore, according to the LTS rules, $\bar{t} \in last(r').db(R)$ for all runs $r' \in [\![r^i]\!]_{P,u}$. From this and the relational calculus semantics, it follows that $[R(\bar{t})]^{last(r').db} = \top$ for all runs $r' \in [\![r^i]\!]_{P,u}$. Hence, $secure_{P,u}(r, i \vdash_u R(\bar{t}))$ holds.
4. *INSERT Success - FD.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\neg\exists\overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(s', \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle) = \top$. From this and f's definition, it follows that $f^u_{conf}(s', \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle) = \top$, because $user(s', \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle) = u$. From this and f^u_{conf}'s definition, it follows that $secure(u, \phi, last(r^{i-1})) = \top$ holds because ϕ is equivalent to $getInfoS(\gamma, a)$ for some $\gamma \in Dep(\Gamma, a)$, where $a = \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. We claim that $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds. From this and Proposition C.7, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds.
   We now prove our claim that $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds. Let s' be the state $last(r^{i-1})$. Note that, for brevity's sake, in the following we omit the *sysState* function where needed. For instance, with a slight abuse of notation, we write $[\![s']\!]^{data}_{M,u}$ instead of $[\![sysState(s')]\!]^{data}_{M,u}$. There are two cases:
   (a) the INSERT command has caused an integrity constraint violation, i.e., $Ex(s) \neq \emptyset$. From $secure(u, \phi, s') = \top$ and Proposition C.9, it follows that $secure^{data}_{P,u}(r, i-1 \vdash_u \phi)$ holds. From this, it follows that $[\phi]^v = [\phi]^{s'}$ for any $v \in [\![s']\!]^{data}_{M,u}$. From this and the fact that the INSERT command caused an exception (i.e., $s' = s$), it follows that $[\phi]^v = [\phi]^s$ for any $v \in [\![s]\!]^{data}_{M,u}$. From this, it follows that $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds.
   (b) the INSERT command has not caused exceptions, i.e., $Ex(s) = \emptyset$. From $secure(u, \phi, s') = \top$ and Proposition C.9, it follows that $secure^{data}_{P,u}(r, i-1 \vdash_u \phi)$ holds. From this, it follows that $[\phi]^v = [\phi]^{s'}$ for any $v \in [\![s']\!]^{data}_{M,u}$. Furthermore, from Proposition C.10 and $Ex(s) = \emptyset$, it follows that ϕ holds in s'. Let $A_{s',R,\bar{t}}$ be the set $\{\langle db[R \oplus \bar{t}], U, sec, T, V\rangle \in \Pi_M \mid \exists db' \in \Omega_D.\, \langle db', U, sec, T, V\rangle \in [\![s']\!]^{data}_{M,u}\}$. It is easy to see that $[\![s]\!]^{data}_{M,u} \subseteq A_{s',R,\bar{t}}$. We now show that ϕ holds for any $z \in A_{s',R,\bar{t}}$. Let $z_1 \in [\![s']\!]^{data}_{M,u}$. From $[\phi]^v = [\phi]^{s'}$ for any $v \in [\![s']\!]^{data}_{M,u}$ and the fact that ϕ holds in s', it follows that $[\phi]^{z_1} = \top$. Therefore, for any $(\overline{k}_1, \overline{k}_2, \overline{k}_3) \in R(z_1)$ such that $|\overline{k}_1| = |\overline{v}|$, $|\overline{k}_2| = |\overline{w}|$, and $|\overline{k}_3| = |\overline{z}|$, if $k_1 = \overline{v}$, then $k_2 = \overline{w}$. Then, for any $(\overline{k}_1, \overline{k}_2, \overline{k}_3) \in R(z_1) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$ such that $|\overline{k}_1| = |\overline{v}|$, $|\overline{k}_2| = |\overline{w}|$, and $|\overline{k}_3| = |\overline{z}|$, if $k_1 = \overline{v}$, then $k_2 = \overline{w}$. Therefore, ϕ holds also in $z_1[R \oplus \bar{t}] \in A_{sysState(s'),R,\bar{t}}$. Hence, $[\phi]^z = \top$ for any $z \in A_{s',R,\bar{t}}$. From this and $[\![s]\!]^{data}_{M,u} \subseteq A_{s',R,\bar{t}}$, it follows that $[\phi]^z = \top$ for any $z \in [\![s]\!]^{data}_{M,u}$. From this, it follows that $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds.
5. *INSERT Success - ID.* The proof of this case is similar to that for the *INSERT Success - FD.*
6. *DELETE Success.* The proof for this case is similar to that of *INSERT Success.*
7. *DELETE Success - ID.* The proof of this case is similar to that for the *INSERT Success - FD.*
8. *INSERT Exception.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSER}, R, \bar{t}\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\neg R(\bar{t})$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(s', \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top$. From this and f's definition, it follows that $f^u_{conf}(s', \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top$, because $user(s', \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = u$. From this and f^u_{conf}'s definition, it follows that $secure(u, \phi, last(r^{i-1})) = \top$

holds because $\phi = getInfo(\langle u, \texttt{INSERT}, R, \overline{t}\rangle)$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. From the LTS semantics, it follows that $sysState(s) \cong^{data}_{M,u} sysState(last(r^{i-1}))$. From this, $secure(u, \phi, last(r^{i-1})) = \top$, and Lemma C.14, it follows that $secure(u, \phi, last(r^i)) = \top$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds.

9. *`DELETE` Exception.* The proof for this case is similar to that of *`INSERT` Exception.*
10. *`INSERT` FD Exception.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(s', \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle) = \top$. From this and f's definition, it follows that $f^u_{conf}(s', \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle) = \top$, because $user(s', \langle u, \texttt{INSERT}, R, \overline{t}\rangle) = u$. From this and f^u_{conf}'s definition, it follows that $secure(u, \phi, last(r^{i-1})) = \top$ because $\phi = getInfoV(\gamma, \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle)$ for some constraint $\gamma \in Dep(\Gamma, \langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle)$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. From the LTS semantics, it follows that $sysState(s) \cong^{data}_{M,u} sysState(last(r^{i-1}))$. From this, Lemma C.14, and $secure(u, \phi, last(r^{i-1})) = \top$, it follows that $secure(u, \phi, last(r^i)) = \top$. From this and Proposition C.9, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds.
11. *`INSERT` ID Exception.* The proof for this case is similar to that of *`INSERT` FD Exception.*
12. *`DELETE` FD Exception.* The proof for this case is similar to that of *`INSERT` FD Exception.*
13. *Integrity Constraint.* The proof of this case follows trivially from the fact that for any state $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and any $\gamma \in \Gamma$, $[\gamma]^{db} = \top$ by definition.
14. *Learn `GRANT/REVOKE` Backward.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c'\rangle$, and t be a trigger whose `WHEN` condition is ϕ and whose action is either a `GRANT` or a `REVOKE`. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$, where u' is either the trigger's owner or the trigger's invoker depending on the security mode. From this and f's definition, it follows $f^u_{conf}(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$, because $user(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = u$ because t's invoker is u according to the rules. From this and f^u_{conf}'s definition, it follows $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds.
15. *Trigger `GRANT` Disabled Backward.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c'\rangle$, t be a trigger whose `WHEN` condition is ψ, and ϕ be $\neg\psi$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$, where u' is either the trigger's owner or the trigger's invoker depending on the security mode. From this and f's definition, it follows $f^u_{conf}(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$, as $user(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = u$ because t's invoker is u according to the rules. From this and f^u_{conf}'s definition, it follows that also $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds.
16. *Trigger `REVOKE` Disabled Backward.* The proof for this case is similar to that of *Trigger `GRANT` Disabled Backward.*
17. *Trigger `INSERT` FD Exception.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c'\rangle$, and t be a trigger whose `WHEN` condition is ϕ and whose action act is a `INSERT` statement $\langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle$. Furthermore, let ϕ be $\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(last(r^{i-1}), act) = \top$. From this and f's definition, it follows that $f^u_{conf}(last(r^{i-1}), act) = \top$, because $user(last(r^{i-1}), act) = u$ because t's invoker is u according to the rules. From this and f^u_{conf}'s definition, it follows that $secure(u, \phi, last(r^{i-1})) = \top$ because $\phi = getInfoV(\gamma, act)$ for some constraint $\gamma \in Dep(\Gamma, act)$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds.
18. *Trigger `INSERT` ID Exception.* The proof for this case is similar to that of *Trigger `INSERT` ID Exception.*
19. *Trigger `DELETE` ID Exception.* The proof for this case is similar to that of *Trigger `DELETE` ID Exception.*
20. *Trigger Exception.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c'\rangle$, and t be a trigger whose `WHEN` condition is ϕ and whose action is act. From the rule's definition, it follows that $f(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$, where u' is either the trigger's owner or the trigger's invoker depending on the security mode. From this and f's definition, it follows $f^u_{conf}(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$, because $user(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = u$ since t's invoker is u according to the rules. From this and f^u_{conf}'s definition, it follows that $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds.
21. *Trigger `INSERT` Exception.* The proof for this case is similar to that of *`INSERT` Exception.*
22. *Trigger `DELETE` Exception.* The proof for this case is similar to that of *`DELETE` Exception.*

23. *Trigger Rollback* `INSERT`. Let i be such that $r^i = r^{i-n-1}{\cdot}\langle u, \texttt{INSERT}, R, \bar{t}\rangle{\cdot}s_1{\cdot}t_1{\cdot}s_2{\cdot}\ldots{\cdot}t_n{\cdot}s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$ and $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and ϕ be $\neg R(\bar{t})$. Furthermore, let $last(r^{i-n-1}) = \langle db', U', sec', T', V', c'\rangle$ and s_n be $\langle db, U, sec, T, V, c\rangle$. From the rule's definition, it follows that $secEx(s_1) = \bot$. From this, it follows that $f(last(r^{i-n-1}), \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top$. From this and f's definition, it follows $f^u_{conf}(last(r^{i-n-1}), \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top$ since $user(last(r^{i-n-1}), \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = u$. From this and f^u_{conf}'s definition, it follows $secure(u, \phi, last(r^{i-n-1})) = \top$ because $\phi = getInfo(\langle u, \texttt{INSERT}, R, \bar{t}\rangle)$. From the LTS semantics, it follows that $last(r^{i-n-1}) \cong^{data}_{M,u} s_n$ because $sysState(last(r^{i-n-1})) = sysState(s_n)$. From this, Lemma C.14, and $secure(u, \phi, last(r^{i-n-1})) = \top$, it follows $secure(u, \phi, s_n) = \top$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds.
24. *Trigger Rollback* `DELETE`. The proof for this case is similar to that of *Trigger Rollback* `INSERT`.
25. *Learn from deny – actions.* We prove this case by contradiction. Assume that we can derive $r, i-1 \vdash_u \phi$ using the rule *Learn from deny – actions*. There are $r, r', r'' \in traces(L)$, $1 < i \leq |r|$, $a \in \mathcal{A}_{D,u}$, $s, s' \in \Omega_M$, and ϕ such that: $r^i = r^{i-1}{\cdot}a{\cdot}s$, $r' = r''{\cdot}a{\cdot}s'$, $r^{i-1} \cong_{P,u} r''$, $secEx(s') \neq secEx(s)$, $[\phi]^{last(r^{i-1}).db} = \top$, and $[\phi]^{last(r'').db} = \bot$. From $r^i = r^{i-1}{\cdot}a{\cdot}s$ and $r' = r''{\cdot}a{\cdot}s'$, it follows that $extend(r^{i-1}, a)$ and $extend(r'', a)$ are well-defined. From this, Lemma C.23, and $a \in \mathcal{A}_{D,u}$, a preserves the equivalence class for r^{i-1}, P, and $\cong_{P,u}$. From this, $r^{i-1} \cong_{P,u} r''$, $extend(r^{i-1}, a) = r^i$, and $extend(r'', a) = r'$, it follows that $r^i \cong_{P,u} r'$. This, however, contradicts $secEx(s') \neq secEx(s)$.
26. *Learn from deny – triggers.* The proof is similar to that of the case *Learn from deny – actions*. However, we use Lemma C.24 instead of Lemma C.23.

This completes the proof of the base step.

Induction Step: Assume that the claim hold for any derivation of $r, j \vdash_u \psi$ such that $|r, j \vdash_u \psi| < |r, i \vdash_u \phi|$. We now prove that the claim also holds for $r, i \vdash_u \phi$. There are a number of cases depending on the rule used to obtain $r, i \vdash_u \phi$.

1. *View.* The proof of this case follows trivially from the semantics of the relational calculus extended over views.
2. *Propagate Forward* `SELECT`. Let i be such that $r^{i+1} = r^i{\cdot}\langle u, \texttt{SELECT}, \psi\rangle{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^i) = \langle db', U', sec', T', V', c'\rangle$. From the rule, it follows that $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, \texttt{SELECT}, \psi\rangle$ preserves the equivalence class with respect to r^i, P, and $\cong_{P,u}$. From this, Lemma C.19, and $secure_{P,u}(r, i \vdash_u \phi)$, it follows that also $secure_{P,u}(r, i+1 \vdash_u \phi)$ holds.
3. *Propagate Forward* `GRANT/REVOKE`. Let i be such that $r^{i+1} = r^i{\cdot}\langle op, u', p, u\rangle{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^i) = \langle db', U', sec', T', V', c'\rangle$. From the rule, it follows that $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle op, u', p, u\rangle$ preserves the equivalence class with respect to r^i, P, and $\cong_{P,u}$. From this, Lemma C.20, and $secure_{P,u}(r, i \vdash_u \phi)$, it follows that also $secure_{P,u}(r, i+1 \vdash_u \phi)$ holds.
4. *Propagate Forward* `CREATE`. The proof for this case is similar to that of *Propagate Forward* `SELECT`.
5. *Propagate Backward* `SELECT`. Let i be such that $r^{i+1} = r^i{\cdot}\langle u, \texttt{SELECT}, \psi\rangle{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$ and $last(r^i) = \langle db, U, sec, T, V, c\rangle$. From the rule, it follows that $r, i+1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i+1 \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, \texttt{SELECT}, \psi\rangle$ preserves the equivalence class with respect to r^i, P, and $\cong_{P,u}$. From this, Lemma C.19, and $secure_{P,u}(r, i+1 \vdash_u \phi)$, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds.
6. *Propagate Backward* `GRANT/REVOKE`. Let i be such that $r^{i+1} = r^i{\cdot}\langle op, u', p, u\rangle{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$ and $last(r^i) = \langle db, U, sec, T, V, c\rangle$. From the rule, it follows that $r, i+1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i+1 \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle op, u', p, u\rangle$ preserves the equivalence class with respect to r^i, P, and $\cong_{P,u}$. From this, Lemma C.20, and $secure_{P,u}(r, i+1 \vdash_u \phi)$, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds.
7. *Propagate Backward* `CREATE TRIGGER`. The proof for this case is similar to that of *Propagate Backward* `SELECT`.
8. *Propagate Backward* `CREATE VIEW`. Note that the formulae ψ and $replace(\psi, o)$ are semantically equivalent. This is the only difference between the proof for this case and the one for the *Propagate Backward* `SELECT` case.
9. *Rollback Backward - 1.* Let i be such that $r^i = r^{i-n-1}{\cdot}\langle u, op, R, \bar{t}\rangle{\cdot}s_1{\cdot}t_1{\cdot}s_2{\cdot}\ldots{\cdot}t_n{\cdot}s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. Furthermore, let s_n be $\langle db', U', sec', T', V', c'\rangle$ and $last(r^{i-n-1})$ be $\langle db, U, sec, T, V, c\rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds.

From Lemma C.24, the trigger t_j preserves the equivalence class with respect to $r^{i-n-1+j}$, P, and $\cong_{P,u}$ for any $1 \leq j \leq n$. Therefore, for any $v \in [\![r^{i-1}]\!]_{P,u}$, the run $e(v, t_n)$ contains the roll-back. Therefore, for any $v \in [\![r^{i-1}]\!]_{P,u}$, the state $last(e(v, t_n))$ is the state just before the action $\langle u, op, R, \bar{t}\rangle$. Let A be the set of system states associated with the roll-back states. It is easy to see that A is the same as $\{sysState(last(t')) \mid t' \in [\![r^{i-n-1}]\!]_{P,u}\}$. From $secure_{P,u}(r, i \vdash_u \phi)$, it follows that ϕ has the same result over all states in A. From this and $A = \{sysState(last(t')) \mid t' \in [\![r^{i-n-1}]\!]_{P,u}\}$, it follows that ϕ has the same result over all states in $\{sysState(last(t')) \mid t' \in [\![r^{i-n-1}]\!]_{P,u}\}$. From this, it follows that $secure_{P,u}(r, i-n-1 \vdash_u \phi)$ holds.

10. *Rollback Backward - 2.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t}\rangle \cdot s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. From this, Lemma C.18, the fact that the action does not modify the database state, and $secure_{P,u}(r, i \vdash_u \phi)$, it follows $secure_{P,u}(r, i-1 \vdash_u \phi)$.
11. *Rollback Forward - 1.* Let i be such that $r^i = r^{i-n-1} \cdot \langle u, op, R, \bar{t}\rangle \cdot s_1 \cdot t_1 \cdot s_2 \cdot \ldots \cdot t_n \cdot s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. Furthermore, let s_n be $\langle db, U, sec, T, V, c\rangle$ and $last(r^{i-n-1})$ be $\langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-n-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i-n-1 \vdash_u \phi)$ holds. From Lemma C.24, the trigger t_j preserves the equivalence class with respect to $r^{i-n-1+j}$, P, and $\cong_{P,u}$ for any $1 \leq j \leq n$. Independently on the cause of the roll-back (either a security exception or an integrity constraint violation), we claim that the set A of roll-back system states is $\{sysState(last(t')) \mid t' \in [\![r^{i-n-1}]\!]_{P,u}\}$. From $secure_{P,u}(r, i-n-1 \vdash_u \phi)$, the result of ϕ is the same for all states in A. From this and $A = \{sysState(last(t')) \mid t' \in [\![r^{i-n-1}]\!]_{P,u}\}$, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds.
We now prove our claim. It is trivial to see (from the LTS semantics) that the set of rollback states is a subset of $\{last(v) \mid v \in [\![r^{i-n-1}]\!]_{P,u}\}$. Assume, for contradiction's sake, that there is a state in $\{last(v) \mid v \in [\![r^{i-n-1}]\!]_{P,u}\}$ that is not a rollback state for the runs in $[\![r^i]\!]_{P,u}$. This is impossible since all triggers $t_1, \ldots, t_n$ preserve the equivalence class.
12. *Rollback Forward - 2.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t}\rangle \cdot s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. From this, Lemma C.18, the fact that the action does not modify the database state, and $secure_{P,u}(r, i-1 \vdash_u \phi)$, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds.
13. *Propagate Forward INSERT/DELETE Success.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t}\rangle \cdot s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. From $revise(r^{i-1}, \phi, r^i)$, it follows that the execution of $\langle u, op, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]_{P,u}$. From this, Lemma C.18, and $secure_{P,u}(r, i-1 \vdash_u \phi)$, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds.
14. *Propagate Forward INSERT Success - 1.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t}\rangle \cdot s$, where op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. We claim that the execution of $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$. From this, Lemma C.18, and $secure_{P,u}(r, i-1 \vdash_u \phi)$, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds.
We now prove our claim that the execution of $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$. From the rule's definition, it follows that $r, i-1 \vdash_u R(\bar{t})$ holds. From this and Proposition C.1, it follows that $[R(\bar{t})]^{last(r^{i-1}).db} = \top$. From $r, i-1 \vdash_u R(\bar{t})$ and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u R(\bar{t}))$ holds. From this and $[R(\bar{t})]^{last(r^{i-1}).db} = \top$, it follows that $[R(\bar{t})]^{last(v).db} = \top$ for any $v \in [\![r^{i-1}]\!]_{P,u}$. From this and the relational calculus semantics, it follows that the execution of $\langle u, op, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]_{P,u}$.
15. *Propagate Forward DELETE Success - 1.* The proof for this case is similar to that of *Propagate Forward INSERT Success - 1.*
16. *Propagate Backward INSERT/DELETE Success.* Let i be such that $r^i = r^{i-1} \cdot \langle u, op, R, \bar{t}\rangle \cdot s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence

class with respect to r^{i-1}, P, and $\cong_{P,u}$. From $revise(r^{i-1}, \phi, r^i)$, it follows that the execution of $\langle u, op, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]_{P,u}$. From this, Lemma C.18, and $secure_{P,u}(r, i \vdash_u \phi)$, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds.

17. *Propagate Backward* `INSERT` *Success - 1.* Let i be such that $r^i = r^{i-1}{\cdot}\langle u, op, R, \bar{t}\rangle{\cdot}s$, where op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds. From Lemma C.23, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. We claim that the execution of $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]_{P,u}$ (the proof of this claim is in the proof of the *Propagate Forward* `INSERT` *Success - 1* case). From this, Lemma C.18, and $secure_{P,u}(r, i \vdash_u \phi)$, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds.
18. *Propagate Backward* `DELETE` *Success - 1.* The proof for this case is similar to that of *Propagate Forward* `DELETE` *Success - 1.*
19. *Reasoning.* Let Δ be a subset of $\{\delta \mid r, i \vdash_u \delta\}$ and $last(r^i) = \langle db, U, sec, T, V, c\rangle$. From the induction hypothesis, it follows that $secure_{P,u}(r, i \vdash_u \delta)$ holds for any $\delta \in \Delta$. Note that, given any $\delta \in \Delta$, from $r, i \vdash_u \delta$ and Proposition C.1, it follows that δ holds in $last(r^i)$. From this, $secure_{P,u}(r, i \vdash_u \delta)$ holds for any $\delta \in \Delta$, $\Delta \models_{fin} \phi$, and Lemma C.16, it follows that $secure_{P,u}(r, i \vdash_u \phi)$ holds.
20. *Learn* `INSERT` *Backward - 3.* Let i be such that $r^i = r^{i-1}{\cdot}\langle u, \texttt{INSERT}, R, \bar{t}\rangle{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and ϕ be $\neg R(\bar{t})$. From the rule's definition, $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(last(r^{i-1}), \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top$. From this and f's definition, it follows that $f^u_{conf}(last(r^{i-1}), \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top$ because $user(last(r^{i-1}), \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = u$. From this and f^u_{conf}'s definition, it follows $secure(u, \phi, last(r^{i-1})) = \top$ because $\phi = getInfo(\langle u, \texttt{INSERT}, R, \bar{t}\rangle)$. From this and Proposition C.9, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds.
21. *Learn* `DELETE` *Backward - 3.* The proof is similar to that of *Learn* `INSERT` *Backward - 3.*
22. *Propagate Forward Disabled Trigger.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and t be a trigger. Furthermore, let ψ be t's condition where all free variables are replaced with $tpl(last(r^{i-1}))$. From the rule, it follows that $r, i-1 \vdash_u \phi$. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. Furthermore, from Lemma C.24, it follows that t preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. If the trigger's action is an `INSERT` or a `DELETE` operation, we claim that the operation does not change the content of any table in $tables(\phi)$ for any run $v \in [\![r^{i-1}]\!]_{P,u}$. We also claim that executing the trigger t in any run $v \in [\![r^{i-1}]\!]_{P,u}$ does not generate security or integrity exceptions. From this, the fact that t preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$, Lemma C.21, and $secure_{P,u}(r, i-1 \vdash_u \phi)$, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds. We now prove our claim. Assume that t's action is an `INSERT` or a `DELETE` operation. From the rule, it follows that $r, i-1 \vdash_u \neg\psi$. From this and Proposition C.1, $[\psi]^{last(r^{i-1})} = \bot$. From $r, i-1 \vdash_u \neg\psi$ and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \psi)$ holds. From this and $[\psi]^{last(r^{i-1}).db} = \bot$, it follows that $[\psi]^{v.db} = \bot$ for any run $v \in [\![r^{i-1}]\!]_{P,u}$. Therefore, the trigger t is disabled in any run $v \in [\![r^{i-1}]\!]_{P,u}$. From this and the LTS semantics, it follows that t's execution does not change the content of any table in $tables(\phi)$ for any run $v \in [\![r^{i-1}]\!]_{P,u}$. We now show that executing the trigger t in any run $v \in [\![r^{i-1}]\!]_{P,u}$ does not generate security or integrity exceptions. From the rule, it follows that executing the trigger t in r^{i-1} does not generate security and integrity exceptions. From this and the fact that t preserves the equivalence class (see Lemma C.24), it immediately follows that executing the trigger t in v does not generate security and integrity exceptions for any $v \in [\![r^{i-1}]\!]_{P,u}$.
23. *Propagate Backward Disabled Trigger.* The proof for this case is similar to that of *Propagate Forward Disabled Trigger.*
24. *Learn* `INSERT` *Forward.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and t be a trigger, and ϕ be $R(\bar{t})$. Furthermore, let ψ be t's condition where all free variables are replaced with $tpl(last(r^{i-1}))$. From the rule's definition, it follows that t's action is $\langle u', \texttt{INSERT}, R, \bar{t}\rangle$ and that $r, i-1 \vdash_u \psi$ holds. From Proposition C.1 and $r, i-1 \vdash_u \psi$, it follows that $[\psi]^{last(r^{i-1}).db} = \top$. From this, $secEx(s) = \bot$, and $Ex(s) = \emptyset$, it follows that t's action has been executed successfully. From this, it follows that $\bar{t} \in s.db(R)$. From $r, i-1 \vdash_u \psi$ and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \psi)$. From this and $[\psi]^{last(r^{i-1}).db} = \top$, it follows that $[\psi]^{last(v).db} = \top$ for any $v \in [\![r^{i-1}]\!]_{P,u}$. From this, it follows that the trigger t is enabled in any run $v \in [\![r^{i-1}]\!]_{P,u}$. From Lemma C.24, it follows that t preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. From this, $secEx(s) = \bot$, $Ex(s) = \emptyset$, and the fact that the trigger t is enabled in any run $v \in [\![r^{i-1}]\!]_{P,u}$, it follows that t's action is executed successfully in any run $e(v, t)$, where $v \in [\![r^{i-1}]\!]_{P,u}$. From this, it follows that $db''(R)$, where $db'' = \bar{t} \in last(e(v, t)).db$, for any $v \in [\![r^{i-1}]\!]_{P,u}$. Therefore, $secure_{P,u}(r, i \vdash_u \phi)$

holds.

25. *Learn* ***INSERT*** *- FD.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be $\neg\exists\overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. Furthermore, let ψ be t's condition where all free variables are replaced with the values in $tpl(last(r^{i-1}))$ and $\langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle$ be t's actual action. From the rule, it follows that $r, i{-}1 \vdash_u \psi$. From this and Proposition C.1, it follows that $[\psi]^{last(r^{i-1}).db} = \top$. From this, $Ex(s) = \emptyset$, and $secEx(s) = \bot$, it follows that $f(s', \langle u', \texttt{INSERT}, R, \overline{t}\rangle) = \top$, where s' is the state just after the execution of the `SELECT` statement associated with t's `WHEN` clause. From this and f's definition, it follows that $f^u_{conf}(s', \langle u', \texttt{INSERT}, R, \overline{t}\rangle) = \top$ because $user(s', \langle u', \texttt{INSERT}, R, \overline{t}\rangle) = u$ since u is t's invoker. From this and f^u_{conf}'s definition, it follows that $secure(u, \phi, s') = \top$. From this, $sysState(s') = sysState(last(r^{i-1}))$, and Lemma C.14, it follows $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows $secure_{P,u}(r, i{-}1 \vdash_u \phi)$. We claim that $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds. From this and Proposition C.7, it follows that also $secure_{P,u}(r, i \vdash_u \phi)$ holds.
We now prove our claim that $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds. Let s' be the state just after the execution of the `SELECT` statement associated with t's `WHEN` clause and s'' be the state $last(r^{i-1})$. Furthermore, for brevity's sake, in the following we omit the *sysState* function where needed. For instance, with a slight abuse of notation, we write $[\![s']\!]^{data}_{M,u}$ instead of $[\![sysState(s')]\!]^{data}_{M,u}$. From $secure(u, \phi, s') = \top$, $s' \cong^{data}_{M,u} s''$, Lemma C.14, and Proposition C.9, it follows that $secure^{data}_{P,u}(r, i - 1 \vdash_u \phi)$ holds. From this, it follows that $[\phi]^v = [\phi]^{s''}$ for any $v \in [\![s'']\!]^{data}_{M,u}$. Furthermore, from Proposition C.10 and $Ex(s) = \emptyset$, it follows that ϕ holds in s''. Let $A_{s'',R,\overline{t}}$ be the set $\{\langle db'[R \oplus \overline{t}], U, sec, T, V\rangle \in \Pi_M \mid \langle db', U, sec, T, V\rangle \in [\![s'']\!]^{data}_{M,u}\}$. It is easy to see that $[\![s]\!]^{data}_{M,u} \subseteq A_{s'',R,\overline{t}}$. We now show that ϕ holds for any $z \in A_{s'',R,\overline{t}}$. Let $z_1 \in [\![s'']\!]^{data}_{M,u}$. From $[\phi]^v = [\phi]^{s''}$ for any $v \in [\![s'']\!]^{data}_{M,u}$ and the fact that ϕ holds in s'', it follows that $[\phi]^{z_1} = \top$. Therefore, for any $(\overline{k}_1, \overline{k}_2, \overline{k}_3) \in R(z_1)$ such that $|\overline{k}_1| = |\overline{v}|$, $|\overline{k}_2| = |\overline{w}|$, and $|\overline{k}_3| = |\overline{q}|$, if $k_1 = \overline{v}$, then $k_2 = \overline{w}$. Then, for any $(\overline{k}_1, \overline{k}_2, \overline{k}_3) \in R(z_1) \cup \{(\overline{v}, \overline{w}, \overline{q})\}$ such that $|\overline{k}_1| = |\overline{v}|$, $|\overline{k}_2| = |\overline{w}|$, and $|\overline{k}_3| = |\overline{q}|$, if $k_1 = \overline{v}$, then $k_2 = \overline{w}$. Therefore, ϕ holds also in $z_1[R \oplus \overline{t}] \in A_{sysState(s''),R,\overline{t}}$. Hence, $[\phi]^z = \top$ for any $z \in A_{s'',R,\overline{t}}$. From this and $[\![s]\!]^{data}_{M,u} \subseteq A_{s'',R,\overline{t}}$, it follows that $[\phi]^z = \top$ for any $z \in [\![s]\!]^{data}_{M,u}$. From this, it follows that $secure^{data}_{P,u}(r, i \vdash_u \phi)$ holds.
26. *Learn* ***INSERT*** *- FD - 1.* The proof of this case is similar to that of *Learn* ***INSERT*** *- FD.*
27. *Learn* ***INSERT*** *- ID.* The proof of this case is similar to that of *Learn* ***INSERT*** *- FD.* See also the proof of ***INSERT*** *Success - ID.*
28. *Learn* ***INSERT*** *- ID - 1.* The proof of this case is similar to that of *Learn* ***INSERT*** *- ID.*
29. *Learn* ***INSERT*** *Backward - 1.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be t's actual `WHEN` condition, where all free variables are replaced with the values in $tpl(last(r^{i-1}))$. From the rule's definition, it follows that $secEx(s) = \top$. From this, the LTS semantics, and $secEx(s) = \top$, it follows that $f(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$. From this and f's definition, it follows $f^u_{conf}(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$ because $user(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = u$ since u is t's invoker. From this and f^u_{conf}'s definition, it follows that $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that also $secure_{P,u}(r, i - 1 \vdash_u \phi)$ holds.
30. *Learn* ***INSERT*** *Backward - 2.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be $\neg R(\overline{t})$. Furthermore, let $act = \langle u', \texttt{INSERT}, R, \overline{t}\rangle$ be t's actual action and γ be t's actual `WHEN` condition obtained by replacing all free variables with the values in $tpl(last(r^{i-1}))$. From the rule's definition, it follows $secEx(s) = \top$ and there is a ψ such that $r, i - 1 \vdash_u \psi$ and $r, i \vdash_u \neg\psi$. We claim that $[\gamma]^{db} = \top$. From this and $secEx(s) = \top$, it follows that $f(s', \langle u', \texttt{INSERT}, R, \overline{t}\rangle) = \top$, where s' is the state obtained after the evaluation of t's `WHEN` condition. From this and f's definition, it follows $f^u_{conf}(s', \langle u', \texttt{INSERT}, R, \overline{t}\rangle) = \top$ as $user(s', \langle u', \texttt{INSERT}, R, \overline{t}\rangle) = u$ because u is t's invoker. From this and f^u_{conf}'s definition, it follows $secure(u, \phi, s') = \top$ since ϕ is equivalent to $getInfo(\langle u', \texttt{INSERT}, R, \overline{t}\rangle)$. From this, Lemma C.14, and $sysState(s') = sysState(last(r^{i-1}))$, it follows $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows $secure_{P,u}(r, i - 1 \vdash_u \phi)$.
We now prove our claim that $[\gamma]^{db} = \top$. Assume, for contradiction's sake, that this is not the case. From this and the LTS rules, it follows that $db = db'$. From the rule's definition, it follows that there is a ψ such that $r, i - 1 \vdash_u \psi$ and $r, i \vdash_u \neg\psi$. From this, Proposition C.1, $s = \langle db', U', sec', T', V', c'\rangle$, and $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, it follows that $[\psi]^{db} = \top$ and $[\neg\psi]^{db'} = \top$. Therefore, $[\psi]^{db} = \top$ and $[\psi]^{db'} = \bot$. Hence, $db \neq db'$, which contradicts $db = db'$.
31. *Learn* ***DELETE*** *Forward.* The proof of this case is similar to that of *Learn* ***INSERT*** *Forward.*
32. *Learn* ***DELETE*** *- ID.* The proof of this case is similar to that of *Learn* ***INSERT*** *- FD.* See also the proof of ***DELETE*** *Success - ID.*
33. *Learn* ***DELETE*** *- ID - 1.* The proof of this case is similar to that of *Learn* ***DELETE*** *- ID.*
34. *Learn* ***DELETE*** *Backward - 1.* The proof is similar to that of *Learn* ***INSERT*** *Backward - 1.*

35. *Learn* `DELETE` *Backward - 2.* The proof is similar to that of *Learn* `INSERT` *Backward - 2.*
36. *Propagate Forward Trigger Action.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where t is a trigger, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. From Lemma C.24, the trigger t preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. We claim that (1) the execution of t does not alter the content of the tables in $tables(\phi)$, and (2) executing the trigger in v does not throw integrity or security exceptions for any $v \in [\![r^{i-1}]\!]_{P,u}$. From this, Lemma C.21, and $secure_{P,u}(r, i-1 \vdash_u \phi)$, it follows $secure_{P,u}(r, i \vdash_u \phi)$.
    We now prove our claim that the execution of t does not alter the content of the tables in $tables(\phi)$. If the trigger is not enabled, the claim is trivial. In the following, we assume the trigger is enabled. There are four cases:
    - t's action is an `INSERT` statement. This case amount to claiming that the `INSERT` statement $\langle u', \texttt{INSERT}, R, \overline{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ in case $revise(r^{i-1}, \phi, r^i) = \top$. We proved the claim in the *Propagate Forward* `INSERT/DELETE` *Success* case.
    - t's action is an `DELETE` statement. The proof is similar to that of the `INSERT` case.
    - t's action is an `GRANT` statement. In this case, the action does not alter the database state and the claim follows trivially.
    - t's action is an `REVOKE` statement. The proof is similar to that of the `GRANT` case.

    We now show that executing the trigger t in any run $v \in [\![r^{i-1}]\!]_{P,u}$ does not generate security or integrity exceptions. From the rule, it follows that executing the trigger t in r^{i-1} does not generate security and integrity exceptions. From this and the fact that t preserves the equivalence class (see Lemma C.24), it immediately follows that executing the trigger t in v does not generate security and integrity exceptions for any $v \in [\![r^{i-1}]\!]_{P,u}$.
37. *Propagate Backward Trigger Action.* The proof of this case is similar to *Propagate Backward Trigger Action.*
38. *Propagate Forward* `INSERT` *Trigger Action.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where t is a trigger, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{P,u}(r, i-1 \vdash_u \phi)$ holds. From Lemma C.24, the trigger t preserves the equivalence class with respect to r^{i-1}, P, and $\cong_{P,u}$. We claim that (1) the execution of t does not alter the content of the tables in $tables(\phi)$, and (2) executing the trigger in v does not throw integrity or security exceptions for all $v \in [\![r^{i-1}]\!]_{P,u}$. From this, Lemma C.21, and $secure_{P,u}(r, i-1 \vdash_u \phi)$, it follows $secure_{P,u}(r, i \vdash_u \phi)$.
    We now prove our claim that the execution of t does not alter the content of the tables in $tables(\phi)$. If the trigger is not enabled, the claim is trivial. In the following, we assume the trigger is enabled. Then, t's action is an `INSERT` statement. This case amount to claiming that the `INSERT` statement $\langle u', \texttt{INSERT}, R, \overline{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ in case $r, i-1 \vdash_u R(\overline{t})$ holds. We proved the claim in the *Propagate Forward* `INSERT` *Success - 1* case. We now show that executing the trigger t in any run $v \in [\![r^{i-1}]\!]_{P,u}$ does not generate security or integrity exceptions. From the rule, it follows that executing the trigger t in r^{i-1} does not generate security and integrity exceptions. From this and the fact that t preserves the equivalence class (see Lemma C.24), it immediately follows that executing the trigger t in v does not generate security and integrity exceptions for all $v \in [\![r^{i-1}]\!]_{P,u}$.
39. *Propagate Forward* `DELETE` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `INSERT` *Trigger Action.*
40. *Propagate Backward* `INSERT` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `INSERT` *Trigger Action.*
41. *Propagate Backward* `DELETE` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `INSERT` *Trigger Action.*
42. *Trigger FD* `INSERT` *Disabled Backward.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$, $t \in \mathcal{TRIGGER}_D$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and ϕ be t's actual `WHEN` condition obtained by replacing all free variables with the values in $tpl(last(r^{i-1}))$. Furthermore, let $act = \langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle$ be t's actual action and α be $\exists \overline{y}, \overline{z}.R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. From the rule's definition, it follows that $secEx(s) = \bot$. From this, it follows that $f(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$. From this and f's definition, it follows $f^u_{conf}(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = \top$ since $user(last(r^{i-1}), \langle u', \texttt{SELECT}, \phi\rangle) = u$ since u is t's invoker. From this and f^u_{conf}'s definition, it follows that $secure(u, \neg\phi, last(r^{i-1})) = \top$. From this, it follows that $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows $secure_{P,u}(r, i-1 \vdash_u \phi)$.
43. *Trigger ID* `INSERT` *Disabled Backward.* The proof of this case is similar to that of *Trigger FD* `INSERT` *Disabled Backward.*
44. *Trigger ID* `DELETE` *Disabled Backward.* The proof of this case is similar to that of *Trigger FD* `INSERT` *Disabled Backward.*

This completes the proof of the induction step.

This completes the proof. □

C.5 Complexity Proofs

Here we prove the complexity of our enforcement mechanism.

C.5.1 Complexity of f_{int}

Theorem C.3 states that f_{int} runs in constant time in terms of data complexity.

Theorem C.3. *The data complexity of f_{int} is $O(1)$.*

Proof. Let $M = \langle D, \Gamma \rangle$ be some fixed system configuration, $a \in \mathcal{A}_{D,U}$ be some fixed action, $u \in \mathcal{U}$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $sec \in \Omega^{sec}_{U,D}$ be some fixed policy, T be some fixed set of triggers over D whose owners are in U, V be some fixed set of views over D whose owners are in U, and c be some fixed context. Furthermore, let $db \in \Omega^{\Gamma}_{D}$ be a database state such that $\langle db, U, sec, T, V, c \rangle \in \Omega_M$. We denote by s the state $\langle db, U, sec, T, V, c \rangle$. By analyzing f_{int}'s definition in Section 6.8.2, it is immediate to see that the complexity of f_{int} directly depends on the complexity of computing $\rightsquigarrow^{appr}_{auth}$. Observe that for all actions $a \in \mathcal{A}_{D,\mathcal{U}}$, we can compute $s \rightsquigarrow^{appr}_{auth} a$ in constant time in terms of data complexity. Observe also that f_{int} only uses checks statements of the form $s \rightsquigarrow^{appr}_{auth} a$, where $a \in \mathcal{A}_{D,\mathcal{U}}$. As a result, the overall data complexity is $O(1)$. □

Lemma C.25 shows that the data complexity of the *apprDet* is $O(1)$ as well.

Lemma C.25. *The data complexity of apprDet is $O(1)$.*

Proof. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $T \subseteq D$ be a set of tables, $V \subseteq \mathcal{VIEW}^{owner}_{D}$ be a set of views over D, ϕ be a formula over D, and s be an M-state. An algorithm that computes $apprDet(T, V, \phi, s, M)$ is as follows:

1. Compute the set $extend(M, s, V)$.
2. Compute the set S of all sub-formulae of ϕ, i.e., $S = subF(\phi)$. Note that $\phi \in subF(\phi)$.
3. Sort by length the set of sub-formulae in such a way that the shortest formula is the first one.
4. Let $S' := \emptyset$.
5. For each sub-formula ψ in the sequence:
   (a) Check whether there is a view $v \in extend(M, s, V)$ such that ψ is v's definition. If this is the case, let $S' = S' \cup subF(\psi)$.
   (b) Perform a case distinction on ψ:
      i. If $\psi := R(\overline{x})$ and $R \in T$, $S' = S' \cup subF(\psi)$.
      ii. If $\psi := V(\overline{x})$ and $\langle V, u, q, O \rangle \in V$, $S' = S' \cup subF(\psi)$.
      iii. If $\psi := \alpha \wedge \beta$ and $\alpha, \beta \in S'$, then $S' = S' \cup subF(\psi)$.
      iv. If $\psi := \alpha \vee \beta$ and $\alpha, \beta \in S'$, then $S' = S' \cup subF(\psi)$.
      v. If $\psi := \neg\alpha$ and $\alpha \in S'$, then $S' = S' \cup subF(\psi)$.
      vi. If $\psi := \exists x.\alpha$ and $\alpha \in S'$, then $S' = S' \cup subF(\psi)$.
      vii. If $\psi := \forall x.\alpha$ and $\alpha \in S'$, then $S' = S' \cup subF(\psi)$.
6. $apprDet(T, V, \phi, s, M) = \top$ iff $S = S'$.

Observe that none of the above steps involve the database state db. Therefore, *apprDet* can be executed in constant time in terms of data complexity. □

C.5.2 Complexity of f^{u}_{conf}

In this section, we prove that data complexity of f^{u}_{conf} is AC^0. Note that the complexity class AC^0 identifies those problems that can be solved using constant-depth, polynomial-size boolean circuits with AND, OR, and NOT gates with unbounded fan-in [10]. Note also that, in the following, with AC^0 we usually refer to *uniform-AC^0* [10]. Given a database schema D and a database state $db \in \Omega^{\Gamma}_{D}$, the *size of db*, denoted also as $|db|$, is $|db| = \Sigma_{R \in D} \Sigma_{\overline{t} \in db(R)} |\overline{t}|$, where the size $|\overline{t}|$ of a tuple $\overline{t}$ is just its cardinality. Similarly, the *the size of the schema D*, denoted $|D|$, is $\Sigma_{R \in D} |R|$. Finally, given a set of views V over D, the *size of the extended vocabulary extVocabulary*(D, V), denoted $|extVoc(D, V)|$, is $\Sigma_{o \in R \cup V} \Sigma_{0 \leq i < |o|} \frac{|o|!}{(|o| - i)! \cdot i!}$. Note that, given a view V, we denote by $|V|$ its cardinality.

Furthermore, given a *RC*-formula ϕ, the *size of* ϕ, denoted as $|\phi|$, is defined as follows:

$$|\phi| = \begin{cases} 1 + |\overline{x}| & \text{if } \phi := R(\overline{x}) \\ 1 & \text{if } \phi := \top \\ 1 & \text{if } \phi := \bot \\ 3 & \text{if } \phi := x = y \\ 1 + |\psi| + |\gamma| & \text{if } \phi := \psi \; O \; \gamma \text{ and } O \in \{\vee, \wedge\} \\ 1 + |\psi| & \text{if } \phi := \neg\psi \\ 2 + |\psi| & \text{if } \phi := Q\, x.\, \psi \text{ and } Q \in \{\exists, \forall\} \end{cases}$$

Lemma C.27 shows that the rewritten formula $\phi^v_{s,u}$, for some $v \in \{\top, \bot\}$, is linear in the size of the original formula ϕ.

Lemma C.26. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be a partial M-state, $u \in U$ be a user, and ϕ be a D-formula. For all formulae ϕ and all $v \in \{\top, \bot\}$, $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be a partial M-state, and $u \in U$ be a user. Let ϕ be an arbitrary formula over $D \cup V$ and v be an arbitrary value in $\{\top, \bot\}$. We now prove that $|\phi^v_{s,u}| \leq m\cdot|\phi|$ by induction over the structure of the formula ϕ.

Base Case There are four cases:

1. $\phi := x = y$. In this case, $\phi^v_{s,u} = \phi$. From this, $|\phi^v_{s,u}| = |\phi|$. From this, it follows trivially that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$.
2. $\phi := \top$. The proof of this case is similar to that of $\phi := x = y$.
3. $\phi := \bot$. The proof of this case is similar to that of $\phi := x = y$.
4. $\phi := R(\overline{x})$. Without loss of generality, we assume that $v = \top$. From this, it follows that $\phi^\top_{s,u} := \bigvee_{S \in R^\top_{s,u}} S(\overline{x})$. From this, it follows that $|\phi^\top_{s,u}| = (|R^\top_{s,u}| - 1) + \Sigma_{S \in R^\top_{s,u}} |S(\overline{x})|$. From this and $|S(\overline{x})| = 1 + |\overline{x}|$, it follows that $|\phi^\top_{s,u}| = (|R^\top_{s,u}| - 1) + \Sigma_{S \in R^\top_{s,u}} (1 + |\overline{x}|)$. From this, it follows that $|\phi^\top_{s,u}| = (|R^\top_{s,u}| - 1) + |R^\top_{s,u}|\cdot(1 + |\overline{x}|)$. From $\phi := R(\overline{x})$, it follows that $|\phi| = 1 + |\overline{x}|$. From this and $|\phi^\top_{s,u}| = (|R^\top_{s,u}| - 1) + |R^\top_{s,u}|\cdot(1 + |\overline{x}|)$, it follows that $|\phi^\top_{s,u}| = |R^\top_{s,u}|\cdot|\phi| + (|R^\top_{s,u}| - 1)$. We claim that $|R^\top_{s,u}| \leq |extVoc(D, V)|$. From this and $|\phi^\top_{s,u}| = |R^\top_{s,u}|\cdot|\phi| + (|R^\top_{s,u}| - 1)$, it follows that $|\phi^\top_{s,u}| \leq |extVoc(D, V)|\cdot|\phi| + |extVoc(D, V)|$. From this, it follows that $|\phi^\top_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$.
   We now prove our claim that $|R^\top_{s,u}| \leq |extVoc(D, V)|$. The set $R^\top_{s,u}$ is a subset of $extVocabulary(D, V)$ by construction. The set $extVocabulary(D, V)$ contains any possible projection of tables in D and views in V. It is easy to check that the cardinality of $extVocabulary(D, V)$ is, indeed, $|extVoc(D, V)|$.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all sub-formulae of ϕ. We now show that our claim holds also for ϕ. There are a number of cases depending on ϕ's structure.

1. $\phi := \psi \wedge \gamma$. From this, it follows that $\phi^v_{s,u} := \psi^v_{s,u} \wedge \gamma^v_{s,u}$. From this, it follows that $|\phi^v_{s,u}| = 1 + |\psi^v_{s,u}| + |\gamma^v_{s,u}|$. From the induction hypothesis, it follows that $|\psi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\psi|$ and $|\gamma^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\gamma|$. From this and $|\phi^v_{s,u}| = 1 + |\psi^v_{s,u}| + |\gamma^v_{s,u}|$, it follows that $|\phi^v_{s,u}| \leq 1 + (|extVoc(D, V)| + 1)\cdot|\psi| + (|extVoc(D, V)| + 1)\cdot|\gamma|$. From this and $|extVoc(D, V)| \geq 0$, it follows that $|\phi^v_{s,u}| \leq |extVoc(D, V)| + 1 + (|extVoc(D, V)| + 1)\cdot|\psi| + (|extVoc(D, V)| + 1)\cdot|\gamma|$. From this, it follows that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot(1 + |\psi| + |\gamma|)$. From this and $|\phi| = 1 + |\psi| + |\gamma|$, it follows that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$.
2. $\phi := \psi \vee \gamma$. The proof of this case is similar to that of $\phi := \psi \wedge \gamma$.
3. $\phi := \neg\psi$. From this, it follows that $\phi^v_{s,u} := \neg\psi^{\neg v}_{s,u}$. From this, it follows that $|\phi^v_{s,u}| = 1 + |\psi^{\neg v}_{s,u}|$. From the induction hypothesis, it follows that $|\psi^{\neg v}_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\psi|$. From this and $|\phi^v_{s,u}| = 1 + |\psi^v_{s,u}|$, it follows that $|\phi^v_{s,u}| \leq 1 + (|extVoc(D, V)| + 1)\cdot|\psi|$. From this and $|extVoc(D, V)| \geq 0$, it follows that $|\phi^v_{s,u}| \leq |extVoc(D, V)| + 1 + (|extVoc(D, V)| + 1)\cdot|\psi|$. From this, it follows that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot(1 + |\psi|)$. From this and $|\phi| = 1 + |\psi|$, it follows that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$.
4. $\phi := \exists x.\, \psi$. If $\phi^v_{s,u}$ is $\neg v$, then the claim holds trivially since $|\phi^v_{s,u}| = 1$. In the following, we assume that $\phi^v_{s,u} := \exists x.\, \psi^v_{s,u}$. From this, it follows that $|\phi^v_{s,u}| = 2 + |\psi^v_{s,u}|$. From the induction hypothesis, it follows that $|\psi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\psi|$. From this and $|\phi^v_{s,u}| = 2 + |\psi^v_{s,u}|$, it follows that $|\phi^v_{s,u}| \leq 2 + (|extVoc(D, V)| + 1)\cdot|\psi|$. From this and $|extVoc(D, V)| \geq 0$, it follows that $|\phi^v_{s,u}| \leq 2\cdot|extVoc(D, V)| + 2 + (|extVoc(D, V)| + 1)\cdot|\psi|$. From this, it follows that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot(2 + |\psi|)$. From this and $|\phi| = 2 + |\psi|$, it follows that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$.
5. $\phi := \forall x.\, \psi$. The proof of this case is similar to that of $\phi := \exists x.\, \psi$.

This completes the proof of the induction step.

This completes the proof of our claim. □

Lemma C.27 states that our rewriting has size that is linear in the original formula's size.

Lemma C.27. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be a partial M-state, $u \in U$ be a user, and ϕ be a D-formula. For all sentences ϕ and all $v \in \{\top, \bot\}$, $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$ and $|\neg\phi^\top_{s,u} \wedge \phi^\bot_{s,u}| \leq 2(|extVoc(D, V)| + 1)\cdot|\phi|$.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, $s = \langle db, U, sec, T, V\rangle$ be a partial M-state, $u \in U$ be a user, and ϕ be a D-formula. Furthermore, let ϕ be a sentence and v be a value in $\{\top, \bot\}$. The fact that $|\phi^v_{s,u}| \leq (|extVoc(D, V)| + 1)\cdot|\phi|$ follows trivially from Lemma C.26. Let ψ be the formula $\neg\phi^\top_{s,u} \wedge \phi^\bot_{s,u}$. The size of ψ is $2 + |\phi^\top_{s,u}| + |\phi^\bot_{s,u}|$. From this and Lemma C.26, it follows that $|\psi| \leq 2 + (|extVoc(D, V)| + 1)\cdot|\phi| + (|extVoc(D, V)| + 1)\cdot|\phi|$. From this, it follows that $|\psi| \leq 2(|extVoc(D, V)| + 1)\cdot|\phi|$. This completes the proof. □

In the following, we study the data complexity of our PDP. Note that, given a PDP f, the *data complexity of f* is the data complexity of the following decision problem:

Definition C.2. Let $M = \langle D, \Gamma\rangle$ be some fixed system configuration, $a \in \mathcal{A}_{D,U}$ be some fixed action, $u \in \mathcal{U}$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $sec \in \Omega^{sec}_{U,D}$ be some fixed policy, T be some fixed set of triggers over D whose owners are in U, V be some fixed set of views over D whose owners are in U, and c be some fixed context.
Input: A database state db such that $\langle db, U, sec, T, V, c\rangle \in \Omega_M$.
Question: Is $f(\langle db, U, sec, T, V, c\rangle, a) = \top$? □

We define in a similar way the data complexity of the *secure* procedure, which we analyze in Lemma C.28.

Lemma C.28. *The data complexity of the secure procedure is AC^0.*

Proof. Let $M = \langle D, \Gamma\rangle$ be some fixed system configuration, ϕ be some fixed sentence, $u \in \mathcal{U}$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $sec \in \Omega^{sec}_{U,D}$ be some fixed policy, T be some fixed set of triggers over D whose owners are in U, V be some fixed set of views over D whose owners are in U, and c be some fixed context. Furthermore, let $db \in \Omega^\Gamma_D$ be a database state such that $\langle db, U, sec, T, V, c\rangle \in \Omega_M$. We denote by s the state $\langle db, U, sec, T, V, c\rangle$. We can check whether $secure(u, \phi, \langle db, U, sec, T, V, c\rangle) = \top$ as follows:

1. Compute the formula $\phi^{rw}_{s,u}$.
2. Compute $[\phi^{rw}_{s,u}]^{db}$.
3. $secure(u, \phi, \langle db, U, sec, T, V, c\rangle) = \top$ iff $[\phi^{rw}_{s,u}]^{db} = \bot$.

We claim that the first step can be done in constant time in terms of data complexity. It is well-known that the data complexity of query execution is AC^0 [10]. From this, it follows that the data complexity of *secure* is also AC^0.

We now prove our claim that computing the formula $\phi^{rw}_{s,u}$ can be done in constant time in terms of data complexity. The extended vocabulary $extVocabulary(D, V)$ does not depend on the database state. From this and the definition of R^v_s, where R is a predicate symbol and $v \in \{\top, \bot\}$, the set R^v_s (and the time needed to compute it) depends just on the database schema D and the set of views V. The set $AUTH_{s,u}$ and the time needed to compute it depend just on the size of the policy sec. Furthermore, the time needed to compute $AUTH^*_{s,u}$ depends just on the size of the policy sec and of the extended vocabulary. Therefore, for any predicate R, the set R^v_s can be computed in constant time in terms of database size. The computation of the formula ϕ', obtained by replacing sub-formulae of the form $\exists\overline{x}.R(\overline{x}, \overline{y})$ with the corresponding predicates in the extended vocabulary, can be done in linear time in terms of $|\phi|$ and in constant time in terms of $|db|$. Note that the size of the resulting formula is linear in $|\phi|$. It is easy to see that also computing $\phi^\top_{s,u}$ and $\phi^\bot_{s,u}$ can be done in linear time in terms of $|\phi|$ and in constant time in terms of $|db|$. As shown in Lemma C.27, the size of the resulting formula is linear in $|\phi|$. Finally, we can replace the predicates in the extended vocabulary with the corresponding sub-formulae again in linear time in terms of $|\phi|$. Note that, again, the size of the resulting formula is linear in $|\phi|$. Therefore, the overall rewriting process can be done in linear time in the size of ϕ and in constant time in the size of db. □

Lemma C.29 states that the data complexity of the $f^u_{conf,\mathtt{I},\mathtt{D}}$ function is AC^0.

Lemma C.29. *The data complexity of $f^u_{conf,I,D}$ is AC^0.*

Proof. Let $M = \langle D, \Gamma \rangle$ be some fixed system configuration, $a \in \mathcal{A}_{D,U}$ be some fixed `INSERT` or `DELETE` action, $u \in \mathcal{U}$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $sec \in \Omega^{sec}_{U,D}$ be some fixed policy, T be some fixed set of triggers over D whose owners are in U, V be some fixed set of views over D whose owners are in U, and c be some fixed context. Furthermore, let $db \in \Omega^{\Gamma}_{D}$ be a database state such that $\langle db, U, sec, T, V, c\rangle \in \Omega_M$. We can check whether $f^{u}_{conf,\mathtt{I,D}}(\langle db, U, sec, T, V, c\rangle, a) = \top$ as follows:

1. If $trigger(s) = \epsilon$ and $a \notin \mathcal{A}_{D,u}$, return $\top$.
2. If $trigger(s) \neq \epsilon$ and $invoker(s) \neq u$, return $\top$.
3. Compute the result of $noLeak(s, a, u)$. If $noLeak(s, a, u) = \bot$, then returns $\bot$.
4. Compute the set $Dep(\Gamma, a)$.
5. Compute $secure(u, getInfo(a), s)$. If its result is $\bot$, return $\bot$.
6. For each $\gamma \in Dep(\Gamma, a)$, compute $secure(u, getInfoV(a, \gamma), s)$. If its result is $\bot$, return $\bot$.
7. For each $\gamma \in Dep(\Gamma, a)$, compute $secure(u, getInfoS(a, \gamma), s)$. If its result is $\bot$, return $\bot$.
8. Return $\top$.

The data complexity of the steps 1 and 2 is $O(1)$. We claim that also the data complexity of the third step is $O(1)$. The complexity of the fourth step is $O(|\Gamma|)$ and therefore its data complexity is again $O(1)$. From the definition of *getInfo*, the resulting formula is constant in the size of the database. Furthermore, also constructing the formula can be done in constant time in the size of the database. From this and Lemma C.28, it follows that the data complexity of the fifth step is AC^0. For a similar reason, the data complexity of the sixth and seventh steps is also AC^0. Therefore, the overall data complexity of the $f^{u}_{conf,\mathtt{I,D}}$ procedure is AC^0.

We now prove our claim that the data complexity of the *noLeak* procedure is $O(1)$. An algorithm implementing the *noLeak* procedure is as follows: for each view $v \in V$, for each grant $g \in sec$, if $g = \langle op, u, \langle \mathtt{SELECT}, v\rangle, u'\rangle$, then (1) compute the set $tDet(v, s, M)$. (2) if $R \in tDet(v, s, M)$, for each $o \in tDet(v, s, M)$, check whether $\langle op, u, \langle \mathtt{SELECT}, o\rangle, u''\rangle \in sec$. The size of the set $tDet(v, s, M)$ is at most $|D|$ and its computation depends just on *sec*. From this, it follows that the complexity of the step 1.(b) is $O(1)$. From Lemma C.25 and the definition of *tDet*, the data complexity of computing $tDet(v, s, M)$ is $O(1)$, and the rest of the computation does not depend on *db* The overall data complexity is, therefore, $O(1)$. □

Lemma C.30 states that the data complexity of the $f^{u}_{conf,\mathtt{G}}$ function is $O(1)$.

Lemma C.30. *The data complexity of $f^{u}_{conf,\mathtt{G}}$ is $O(1)$.*

Proof. Let $M = \langle D, \Gamma \rangle$ be some fixed system configuration, $a \in \mathcal{A}_{D,U}$ be some fixed `GRANT` action, $u \in \mathcal{U}$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $sec \in \Omega^{sec}_{U,D}$ be some fixed policy, T be some fixed set of triggers over D whose owners are in U, V be some fixed set of views over D whose owners are in U, and c be some fixed context. Furthermore, let $db \in \Omega^{\Gamma}_{D}$ be a database state such that $\langle db, U, sec, T, V, c\rangle \in \Omega_M$. We can check whether $f^{u}_{conf,\mathtt{G}}(\langle db, U, sec, T, V, c\rangle, \langle op, u'', p, u'\rangle)) = \top$ as follows.

1. If $trigger(s) = \epsilon$ and $a \notin \mathcal{A}_{D,u}$, return $\top$.
2. If $trigger(s) \neq \epsilon$ and $invoker(s) \neq u$, return $\top$.
3. If p is not a `SELECT` privilege, return $\top$.
4. If $u'' \neq u$, return $\top$.
5. For each $g \in sec$, if $g = \langle op, u, p, u'\rangle$, return $\top$.
6. Return $\bot$.

The complexity of the fifth step is $O(|sec|)$, whereas the complexity of the other steps is $O(1)$. Therefore, the overall complexity of the $f^{u}_{conf,\mathtt{G}}$ procedure is $O(|sec|)$. From this, it follows that the data complexity of $f^{u}_{conf,\mathtt{G}}$ procedure is $O(1)$. □

Lemma C.31 states that the data complexity of the $f^{u}_{conf,\mathtt{S}}$ function is AC^0.

Lemma C.31. *The data complexity of $f^{u}_{conf,\mathtt{S}}$ is AC^0.*

Proof. Let $M = \langle D, \Gamma \rangle$ be some fixed system configuration, $a \in \mathcal{A}_{D,U}$ be some fixed `SELECT` action $\langle u', \mathtt{SELECT}, \phi\rangle$, $u \in \mathcal{U}$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $sec \in \Omega^{sec}_{U,D}$ be some fixed policy, T be some fixed set of triggers over D whose owners are in U, V be some fixed set of views over D whose owners are in U, and c be some fixed context. Furthermore, let $db \in \Omega^{\Gamma}_{D}$ be a database state such that $\langle db, U, sec, T, V, c\rangle \in \Omega_M$. We can check whether $f^{u}_{conf,\mathtt{S}}(\langle db, U, sec, T, V, c\rangle, a)) = \top$ as follows.

1. If $trigger(s) = \epsilon$ and $a \notin \mathcal{A}_{D,u}$, return $\top$.
2. If $trigger(s) \neq \epsilon$ and $invoker(s) \neq u$, return $\top$.
3. Compute $secure(u, \phi, s)$ and return its result.

The complexity of the first and second steps is $O(1)$. From Lemma C.28, it follows that the data complexity of the third step is AC^0. From this, it follows that the data complexity of $f^u_{conf,\mathtt{S}}$ procedure is AC^0. □

Finally, Theorem C.4 states that the data complexity of f^u_{conf} is AC^0.

Theorem C.4. *The data complexity of f^u_{conf} is AC^0.*

Proof. Let $M = \langle D, \Gamma \rangle$ be some fixed system configuration, $a \in \mathcal{A}_{D,U}$ be some fixed action, $u \in \mathcal{U}$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $sec \in \Omega^{sec}_{U,D}$ be some fixed policy, T be some fixed set of triggers over D whose owners are in U, V be some fixed set of views over D whose owners are in U, and c be some fixed context. The data complexity of f^u_{conf} is the maximum of the data complexities of $f^u_{conf,\mathtt{I},\mathtt{D}}$, $f^u_{conf,\mathtt{G}}$, and $f^u_{conf,\mathtt{S}}$. From Lemmas C.29–C.31, it follows that: (a) the data complexity of $f^u_{conf,\mathtt{I},\mathtt{D}}$ is AC^0, (b) the data complexity of $f^u_{conf,\mathtt{S}}$ is AC^0, and (c) the data complexity of $f^u_{conf,\mathtt{G}}$ is $O(1)$. From this, it follows that the data complexity of f^u_{conf} is $max(AC^0, O(1))$. Hence, the data complexity of f^u_{conf} is AC^0. □

C.5.3 Complexity of the overall algorithm

Here we prove the data complexity of the PDP f.

Theorem C.5. *The data complexity of f is AC^0.*

Proof. From f's definition, it follows that f's data complexity is the maximum complexity between f^u_{conf}'s complexity and f_{int}'s complexity. From this, Theorem C.3, and Theorem C.4, it follows that the data complexity of f is AC^0. □

Appendix D

Proofs for Chapter 7

We remark that in the following we consider only sequences of queries, WhileSql programs, and configurations that are well-defined, i.e., that do not get stuck according to the WhileSql's semantics. Observe that this requires, for instance, that all views are used only after their declaration.

D.1 From Database Access Control to Information-flow Control

Here, we prove the main results from Section 7.5, namely Theorems 7.1 and 7.2. In this section, we refer to terminology, notation, and auxiliary functions taken from Chapters 5 and 6.

D.1.1 Weak indistinguishability

We now introduce a new notion of indistinguishability among database runs. This notion plays a key role in our proofs. Observe that this indistinguishability notion is strictly weaker than the one given in Chapter 6.

Definition D.1. Let M be a system configuration, f be an M-PDP, L be the $\langle M, f\rangle$-LTS, and u be a user. We say that two M-system states $s = \langle db, U, sec, T, V\rangle$ and $s' = \langle db', U', sec', T', V'\rangle$ are *configuration indistinguishable*, written $s \cong^{conf}_{M,u} s'$, iff $U = U'$, $sec = sec'$, $T = T'$, and $V = V'$. Moreover, two runs r and r' in $traces(L)$ are $(\langle M, f\rangle, u)$-*weakly indistinguishable*, written $r \cong^{\mathbf{W}}_{u,\langle M,f\rangle} r'$, iff

1. $r|_u$ and $r'|_u$ are consistent.
2. If $r|^2_u = s'{\cdot}a{\cdot}s$ and $a \neq *$, then $sysState(last(r|^1_u))$ and $sysState(last(r'|^1_u))$ are (M, u)-data indistinguishable.
3. If $r|^{|r|_u|}_u = r|^{|r|_u|-1}_u{\cdot} * {\cdot}s$, then $sysState(last(r))$ and $sysState(last(r'))$ are (M, u)-data indistinguishable.
4. For all i such that $1 \leq i \leq |r|_u| - 2$, if $r|^{i+2}_u = r|^i_u{\cdot} * {\cdot}s'{\cdot}a{\cdot}s$ and $a \neq *$, then $sysState(last(r|^{i+1}_u))$ and $sysState(last(r'|^{i+1}_u))$ are (M, u)-data indistinguishable.
5. For all i such that $1 \leq i \leq |r|_u|-1$, if $r|^{i+1}_u = r|^i_u{\cdot}a{\cdot}s$, $a \neq *$, and $s \in \Omega_M$, then $sysState(last(r|^i_u))$ and $sysState(last(r'|^i_u))$ are configuration indistinguishable.
6. $sysState(last(r))$ and $sysState(last(r'))$ are configuration indistinguishable. □

We denote by $[\![r]\!]^{\mathbf{W}}_{P,u}$ the equivalence class defined by r with respect to $\cong^{\mathbf{W}}_{P,u}$.

D.1.2 Auxiliary notation

Here, we introduce some notation and machinery for (1) moving between WhileSql runs and database runs from Chapter 5, and (2) moving between a set of initial states (at a certain point in the run) and the final states obtained by executing a given sequence of queries.

Extension. Let $M = \langle D, \Gamma\rangle$ be a system configuration, f be an M-PDP, L be the $\langle M, f\rangle$-LTS, and r be a run in $traces(L)$. Furthermore, let $\langle u, q\rangle$ be a query. We denote by $extend(r, \langle u, q\rangle)$ the run obtained by extending r with $\langle u, q\rangle$ (if such a run exists). Furthermore, we denote by $extend^*(r, \langle u, q\rangle)$ the run obtained by extending r with the action $\langle u, q\rangle$ and all triggers executed in response to $\langle u, q\rangle$. Similarly, given a trigger t, we denote by $extend(r, t)$ the run obtained by extending r with t (if such a run exists). We refer the reader to Appendix C for a formal definition of *extend*. For conciseness, in the following we use e and e^* instead of *extend* and *extend**.

Notation for database access control. Let $M = \langle D, \Gamma\rangle$ be a system configuration, f be an M-PDP, and L be the $\langle M, f\rangle$-LTS. The *trigger-free projection of a run* r, denoted $\mu(r)$, is defined as follows: $\mu(\epsilon) = \epsilon$, $\mu(s) = s$ where $s \in \Omega_M$,

$$\mu(r) = \begin{cases} \mu(r'){\cdot}a{\cdot}s & r' \in traces(L) \wedge s \in \Omega_M \wedge a \in \mathcal{A}_{D,\mathcal{U}} \wedge r = r'{\cdot}a{\cdot}s \wedge \neg ID(a) \\ \mu(r'){\cdot}a{\cdot}s & r' \in traces(L) \wedge s, s_1, \ldots, s_n \in \Omega_M \wedge t_1, \ldots, t_n \in \mathcal{TRIGGER} \wedge a \in \mathcal{A}_{D,\mathcal{U}} \wedge \\ & r = r'{\cdot}a{\cdot}s_1{\cdot}t_1{\cdot}s_2 \ldots {\cdot}s_n{\cdot}t_n{\cdot}s \wedge ID(a) \end{cases}$$

where $ID(a)$ returns $\top$ iff a is an `INSERT` or `DELETE` action.

Notation for triggers. Given a run $r \in traces(L)$, we denote by $triggers(r, i)$ the sequence of triggers executed in r in response to the i-th query.

Notation for Reduction 7.1. We denote by $code(ctx, q, u)$ (respectively $\mathcal{S}(ctx, q, u)$ and $mem(ctx, q, u)$) the programs $c_1 \cdot \ldots \cdot c_n$ (respectively the scheduler $\mathcal{S}$ and the memories $m_1 \cdot \ldots \cdot m_n$) constructed according to Reduction 7.1.

Let $M = \langle D, \Gamma \rangle$ be a system configuration, *ifEnf* be an enforcement mechanism for WhileSql programs that is sound and stable (as required by Theorem 7.1), u be a user, and *dbEnf* be the database access control mechanism constructed using Reduction 7.1. Furthermore, let P be $\langle M, dbEnf \rangle$ be an extended configuration, L be the P-LTS, and $\langle s, ctx \rangle$ be the last state of r, i.e, $\langle s, ctx \rangle = last(r)$. In the following, we use f to refer to *dbEnf*. With a slight abuse of notation, given a run $r \in traces(L)$ and an action $\langle u, q \rangle$, we write $[r, \langle u, q \rangle]^{ifc}$ to denote the run generated by the WhileSql programs produced by our reduction and starting from the state and scheduler resulting from Reduction 7.1. Observe that each program produced by our reduction is processed by the WhileSql's semantics in two steps, namely one application of the M-Eval-Step rule and one application of the M-Eval-End rule. Abusing notation, in the following we ignore the steps associated with the M-Eval-End rule. Therefore, the i-th statement executed in the run $[r, \langle u, q \rangle]^{ifc}$ corresponds to the i-th command executed in $\mu(e(r, \langle u, q \rangle)$ (and $|[r, \langle u, q \rangle]^{ifc}| = |\mu(e(r, \langle u, q \rangle)|$).

Notation for epochs. Let ep be a user-based epoch (as defined by the predicates given in Section 7.5) in $[r, \langle u, q \rangle]^{ifc}$ and i be a natural number such that $1 \leq i \leq |ep|$. We denote by $start(ep)$ the first state in the epoch and by $end(ep)$ the last state in the epoch. Since our reduction does not use the memory, with a slight abuse of notation, we ignore memories. For instance, we assume that $start(ep)$ and $end(ep)$ consist just of database states. Similarly, we write $ifEnf(C, \mathcal{S}, u, s)$ instead of $ifEnf(C, M, \mathcal{S}, u, s)$. Furthermore, we denote by $code(ep)$ the initial WhileSql program associated with the epoch ep, and $\mathcal{S}(ep)$ the initial scheduler in ep. We also denote by $trace(ep)$ the trace associated with ep and by $trace(ep, i)$ the trace obtained after executing i queries in ep. Finally, we denote by $PK_u(ep, i)$ the knowledge $PK_u(start(ep), code(ep), \mathcal{S}(ep), trace(ep, i))$.

Given a set of runs R and a sequence of queries $\overline{q} = \langle u_1, q_1 \rangle \cdot \ldots \cdot \langle u_n, q_n \rangle$, we denote by $\diamond(R, \overline{q})$ the set of states obtained by repeatedly extending all runs in R with the commands in $\overline{q}$. Furthermore, we denote by $[db(R)]$ the set $\{last(r) \mid r \in R\}$.

D.1.3 Proofs about weak indistinguishability

We now prove that the indistinguishability notion from Chapter 6 implies weak indistinguishability (or, equivalently, that the equivalence class defined by the indistinguishability notion is included in the one defined by weak indistinguishability).

Proposition D.1. *Let M be a system configuration, f be an M-PDP, L be the $\langle M, f \rangle$-LTS, and u be a user. Furthermore, let r, r' be two runs in $traces(L)$. The following facts hold:*

1. *If $r \cong_{u,\langle M,f \rangle} r'$, then $r \cong^{\mathbf{W}}_{u,\langle M,f \rangle} r'$.*
2. *$[\![r]\!]_{u,\langle M,f \rangle} \subseteq \{r' \in traces(L) \mid r \cong^{\mathbf{W}}_{u,\langle M,f \rangle} r'\}$.*

Proof. We first show that the second claim follows from the first one. Assume, for contradiction's sake, that $[\![r]\!]_{u,\langle M,f \rangle} \not\subseteq \{r' \in traces(L) \mid r \cong^{\mathbf{W}}_{u,\langle M,f \rangle} r'\}$. Therefore, there is a run $r'' \in [\![r]\!]_{u,\langle M,f \rangle}$ such that $r'' \notin \{r' \in traces(L) \mid r \cong^{\mathbf{W}}_{u,\langle M,f \rangle} r'\}$. From this, it follows that $r \cong_{u,\langle M,f \rangle} r'$ and $r \ncong^{\mathbf{W}}_{u,\langle M,f \rangle} r'$, which contradicts the first claim.

We now prove that if $r \cong_{u,\langle M,f \rangle} r'$, then $r \cong^{\mathbf{W}}_{u,\langle M,f \rangle} r'$. Let M be a system configuration, f be an M-PDP, L be the $\langle M, f \rangle$-LTS, and u be a user. Furthermore, let r, r' be two runs in $traces(L)$ such that $r \cong_{u,\langle M,f \rangle} r'$. From this, it follows:

(a) $r|_u$ and $r'|_u$ are consistent, and

(b) $sysState(last(r))$ and $sysState(last(r'))$ are (M, u)-data indistinguishable, and

(c) for all i such that $1 \leq i \leq |r|_u| - 1$, if $r|^{i+1}_u = r|^i_u \cdot a \cdot s$, $a \neq *$, and $s \in \Omega_M$, then $sysState(last(r|^i_u))$ and $sysState(last(r'|^i_u))$ are (M, u)-data indistinguishable.

From (b), it follows that $sysState(last(r))$ and $sysState(last(r'))$ are configuration indistinguishable (since data indistinguishability implies configuration indistinguishability). From (c), it follows that (1) if $r|^2_u = s' \cdot a \cdot s$ and $a \neq *$, then $sysState(last(r|^1_u))$ and $sysState(last(r'|^1_u))$ are (M, u)-data indistinguishable, (2) if $r|^{|r|_u|}_u = r|^{|r|_u|-1}_u \cdot * \cdot s$, then $sysState(last(r))$ and $sysState(last(r'))$ are (M, u)-data indistinguishable, and (3) for all i such that $1 \leq i \leq |r|_u| - 2$, if $r|^{i+2}_u = r|^i_u \cdot * \cdot s' \cdot a \cdot s$ and $a \neq *$, then $sysState(last(r|^{i+1}_u))$ and $sysState(last(r'|^{i+1}_u))$ are (M, u)-data indistinguishable, and (4) for all i such that $1 \leq i \leq |r|_u| - 1$, if $r|^{i+1}_u = r|^i_u \cdot a \cdot s$, $a \neq *$, and $s \in \Omega_M$, then $sysState(last(r|^i_u))$ and $sysState(last(r'|^i_u))$ are data indistinguishable, which implies that $sysState(last(r|^i_u))$ and $sysState(last(r'|^i_u))$ are configuration indistinguishable. Therefore, it follows that $r \cong^{\mathbf{W}}_{u,\langle M,f \rangle} r'$. □

Using Proposition D.1, we now show that judgment security with respect to weak indistinguishability implies judgment security with respect to Chapter 6's indistinguishability. The same applies to data confidentiality.

Proposition D.2. *Let M be a system configuration, f be an M-PDP, $P = \langle M, f\rangle$, L be the P-LTS, A be an attacker model, and u be a user. The following facts hold:*
(1) $\mathit{secure}_{P,\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ implies $\mathit{secure}_{P,\cong_{P,u}}(r, i \vdash_u \phi)$.
(2) If f provides data confidentiality with respect to P, u, A, and $\cong^{\mathbf{W}}_{P,u}$, then f provides data confidentiality with respect to P, u, A, and $\cong_{P,u}$.

Proof. Let M be a system configuration, f be an M-PDP, $P = \langle M, f\rangle$, L be the P-LTS, A be an attacker model, and u be a user. We first prove that $\mathit{secure}_{P,\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ implies $\mathit{secure}_{P,\cong_{P,u}}(r, i \vdash_u \phi)$. Assume, for contradiction's sake, that this is not the case. From $\mathit{secure}_{P,\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$, it follows that for all $r' \in \mathit{traces}(L)$ such that $r^i \cong^{\mathbf{W}}_{P,u} r'$, it holds that $[\phi]^{db} = [\phi]^{db'}$, where $\mathit{last}(r^i) = \langle db, U, S, T, V, c\rangle$ and $\mathit{last}(r') = \langle db', U', S', T', V', c'\rangle$. From the fact that $\mathit{secure}_{P,\cong_{P,u}}(r, i \vdash_u \phi)$ does not hold, it follows that there is a run $r' \in \mathit{traces}(L)$ such that $r^i \cong_{P,u} r'$ and $[\phi]^{db} \neq [\phi]^{db'}$, where $\mathit{last}(r^i) = \langle db, U, S, T, V, c\rangle$ and $\mathit{last}(r') = \langle db', U', S', T', V', c'\rangle$. From this and Proposition D.1, it follows that there is a run $r' \in \mathit{traces}(L)$ such that $r^i \cong^{\mathbf{W}}_{P,u} r'$ and $[\phi]^{db} \neq [\phi]^{db'}$, where $\mathit{last}(r^i) = \langle db, U, S, T, V, c\rangle$ and $\mathit{last}(r') = \langle db', U', S', T', V', c'\rangle$. This contradicts $\mathit{secure}_{P,\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$.

We now prove the second claim. Assume that f provides data confidentiality with respect to P, u, A, and $\cong^{\mathbf{W}}_{P,u}$. From this, it follows that $\mathit{secure}_{P,\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds for all judgments $r, i \vdash_u \phi$ that hold in A. From this and the first claim, it follows that $\mathit{secure}_{P,\cong_{P,u}}(r, i \vdash_u \phi)$ for all judgments $r, i \vdash_u \phi$ that hold in A. From this, it follows that f provides data confidentiality with respect to P, u, A, and $\cong_{P,u}$. □

D.1.4 Correctness of Reduction 7.1

Here we prove some facts about Reduction 7.1 and how it relates to the database runs.

Proposition D.3. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, ifEnf be an enforcement mechanism for* WhileSql *programs, u be a user, and dbEnf be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = \langle M, \mathit{dbEnf}\rangle$ be the extended configuration and L be the P-LTS. Let r be a run in $\mathit{traces}(L)$ and $\langle u, q\rangle$ be a database command such that $e(r, \langle u, q\rangle)$ is defined. The run $r_1 = [r, \langle u, q\rangle]^{ifc}$ produced by executing $\mathit{code}(ctx, q, u)$ starting from $\langle \mathit{mem}(ctx, q, u), \mathit{init}\langle s, ctx\rangle\rangle$ with scheduler $\mathcal{S}(ctx, q, u)$, where ctx is the context in the last state of r and s is the last state in r, satisfies the following conditions:*

1. *For $1 \leq i < |\mu(e(r, \langle u, q\rangle))| - 1$, the i-th query executed in $\mu(e(r, \langle u, q\rangle))$ is $\langle u_i, q_i\rangle$ iff the i-th statement executed in r_1 is $\langle u_i, x \leftarrow q_i\rangle$ if $\mathit{secEx}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) = \bot$ and **skip** otherwise.*
2. *The $(|\mu(e(r, \langle u, q\rangle))| - 1)$-th query executed in $\mu(e(r, \langle u, q\rangle))$ is $\langle u, q\rangle$ iff the $(|\mu(e(r, \langle u, q\rangle))| - 1)$-th statement executed in r_1 is $\langle u, x \leftarrow q\rangle$.*
3. *For all $1 \leq i \leq |\mu(e(r, \langle u, q\rangle))|$, (1) $\mathit{last}(\mu(e^*(r, \langle u, q\rangle))^i) = \langle s, ctx\rangle$ iff the i-th configuration in r_1 is $\langle C, M, \langle s, ctx'\rangle, \mathcal{S}\rangle$, and (2) $\mathit{inTrigger}(ctx) = \mathit{inTrigger}(ctx') = \bot$.*
4. *For all $1 \leq i < |\mu(e(r, \langle u, q\rangle))| - 1$, if the i-th query $\langle u_i, q_i\rangle$ in $\mu(e(r, \langle u, q\rangle))$ is authorized and no triggers associated with q_i throw integrity or security exceptions (i.e., $\mathit{secEx}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) = \bot$ and $\mathit{Ex}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) = \emptyset$), then the result of $\langle u_i, q_i\rangle$ in $\mu(e(r, \langle u, q\rangle))$ is k iff the i-th step in r_1 is associated with the label $\langle u', q_i, k, \tau\rangle$, where $u' = \mathit{Usr}(u_i, x \leftarrow q_i)$.*
5. *For all $1 \leq i < |\mu(e(r, \langle u, q\rangle))| - 1$, if the i-th query $\langle u_i, q_i\rangle$ in $\mu(e(r, \langle u, q\rangle))$ is authorized (i.e., $\mathit{secEx}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) = \bot$) and the query itself has caused an exception (i.e., $\mathit{Ex}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1}) \neq \emptyset$ and $\mathit{triggers}(e(r, \langle u, q\rangle), i) = \epsilon$), then $\mathit{Ex}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) = K$ iff the i-th step in r_1 is associated with the label $\langle db(u), q_i, \langle \textbf{\textit{IntEx}}, K\rangle, \epsilon\rangle$.*
6. *For all $1 \leq i < |\mu(e(r, \langle u, q\rangle))| - 1$, if the i-th query $\langle u_i, q_i\rangle$ in $\mu(e(r, \langle u, q\rangle))$ is authorized (i.e., $\mathit{secEx}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) = \bot$) and the trigger t executed in response to q has caused an exception (i.e., $\mathit{Ex}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) \neq \emptyset$ and $\mathit{triggers}(e(r, \langle u, q\rangle), i) = \bar{t}{\cdot}t$), then $\mathit{Ex}(\mathit{last}(\mu(e(r, \langle u, q\rangle))^{i+1})) = K$ iff the i-th step in r_1 is associated with the label $\langle db(u), q_i, \langle t, \mathtt{B}, \textbf{\textit{IntEx}}, K\rangle, \tau\rangle$.*
7. *For all $1 \leq i < |\mu(e(r, \langle u, q\rangle))| - 1$, the label $\langle u, q, m, \tau{\cdot}\langle \mathit{public}, t, q\rangle{\cdot}\tau'\rangle$ is associated with the i-th step in r_1 iff the trigger t is successfully executed in response to the i-th query in $\mu(e(r, \langle u, q\rangle))$, its condition is enabled, and it modified the security policy (and it has been the $|\tau| + 1$-th trigger to do so in the i-th step).*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, *ifEnf* be an enforcement mechanism for WhileSql programs, u be a user, and *dbEnf* be the database access control mechanism constructed using

Reduction 7.1. Furthermore, let $P = \langle M, dbEnf \rangle$ be the extended configuration and L be the P-LTS. Let r be a run in $traces(L)$ and $\langle u, q \rangle$ be a database command such that $e(r, \langle u, q \rangle)$ is defined. Finally, let $r_1 = [r, \langle u, q \rangle]^{ifc}$ be the run produced by executing $code(ctx, q, u)$ starting from $\langle mem(ctx, q, u), init\langle s, ctx \rangle \rangle$ with scheduler $\mathcal{S}(ctx, q, u)$, where $\langle s, ctx \rangle$ is the database state in the last state of r.

Proof of (1). The first claim directly follows from the construction of Reduction 7.1. The reduction associates a WhileSql program $c_i = \langle u_i, s_i \rangle$ to each query $\langle u_i, q_i \rangle$ in $\mu(r)$ such that s_i is $x \leftarrow q_i$ if q_i is authorized (i.e., $secEx(last(\mu(r)^{i+1})) = \bot$) and $s_i = \textbf{skip}$ otherwise. Furthermore, the order in which the WhileSql programs are executed is directly dictated by the run $e(r, \langle u, q \rangle)$. In particular, the i-th step in r_1 is associated to the program c_i, i.e., to the i-th query in $\mu(e(r, \langle u, q \rangle))$ (and all the associated triggers). Let i be a value such that $1 \leq i < |\mu(e(r, \langle u, q \rangle))| - 1$. The i-th statement executed in r_1 is exactly the program c_i. From this, it follows that the i-th query executed in $\mu(e(r, \langle u, q \rangle))$ is $\langle u_i, q_i \rangle$ iff the i-th statement executed in r_1 is $(u_i, x \leftarrow q_i)$ if $secEx(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = \bot$ and **skip** otherwise (by construction). This completes the proof of the first claim.

Proof of (2). The proofs of the second claim also directly follows from the construction of Reduction 7.1. From $e(r, \langle u, q \rangle)$ being defined, it follows that $inTrigger(ctx) = \bot$. From this, the last query in L' is not part of the history in ctx. Therefore, the program $c_{|L'|}$ is $(u, x \leftarrow q)$. This together with $|L'| = |\mu(e(r, \langle u, q \rangle)) - 1|$ gives the second claim.

Proof of (3). We now prove, by induction on i, that for all $1 \leq i \leq |\mu(e(r, \langle u, q \rangle))|$, $last(\mu(e(r^*, \langle u, q \rangle))^i) = \langle s, ctx \rangle$ iff the i-th configuration in r_1 is $\langle C, M, \langle s, ctx' \rangle, \mathcal{S} \rangle$, $inTrigger(ctx) = \bot$, and $inTrigger(ctx') = \bot$. The proof of the base case $i = 1$ is trivial. Indeed, the reduction directly extracts the initial system state from s and ctx. From this, the initial configuration in r_1 is $\langle code(ctx, q, u), M, s, \mathcal{S}(ctx, q, u) \rangle$ and $s = last(\mu(e^*(r, \langle u, q \rangle)))^1$ by construction. For the induction's step, assume that $last(\mu(e^*(r, \langle u, q \rangle))^j) = \langle s', ctx \rangle$ and the j-th configuration in r_1 is $\langle C', M', \langle s', ctx' \rangle, \mathcal{S}' \rangle$ and $inTrigger(ctx) = inTrigger(ctx') = \bot$ for any $j < i$. We now prove that $last(\mu(e^*(r, \langle u, q \rangle))^i) = \langle s, ctx \rangle$ and the i-th configuration in r_1 is $\langle C, M, \langle s, ctx' \rangle, \mathcal{S} \rangle$ and $inTrigger(ctx) = inTrigger(ctx') = \bot$. There are two cases:

1. Assume that $c_{i-1} = \langle u_{i-1}, \textbf{skip} \rangle$. From this, it directly follows that the i-th configuration in r_1 is $\langle C, M, \langle s, ctx' \rangle, \mathcal{S}' \rangle$. From this and the E-Skip and M-Eval-Step rules, it follows that $\langle s, ctx' \rangle$ are the same as those in the $(i-1)$-th configuration in r_1. Furthermore, from the first two claims, it follows that the $(i-1)$-th query (or one of the associated triggers) in $\mu(e^*(r, \langle u, q \rangle))$ has not been authorized. Therefore, from the database operational semantics, it follows that (1) s is the same as the $(i-1)$-th database state in $e^*(r, \langle u, q \rangle)$, and (2) $inTrigger(ctx) = \bot$. From these facts, we can derive that $last(\mu(e^*(r, \langle u, q \rangle))^i) = \langle s, ctx \rangle$ and the i-th configuration in r_1 is $\langle C, M, \langle s, ctx' \rangle, \mathcal{S} \rangle$ and $inTrigger(ctx) = inTrigger(ctx') = \bot$ (by applying the induction hypothesis to the $(i-1)$-th states).
2. Assume that $c_{i-1} = \langle u_{i-1}, x \leftarrow q_{i-1} \rangle$. From this and the WhileSql's semantics, it follows that the $\langle s, ctx' \rangle$ is the database state obtained by executing the query $\langle u_{i-1}, q_{i-1} \rangle$ (together with all associated triggers) on the state $\langle s_1, ctx_1 \rangle$ in the $(i-1)$-th configuration in r_1. Similarly, the i-th state $\langle s, ctx \rangle$ in $\mu(e^*(r, \langle u, q \rangle))$ is obtained by executing the query $\langle u_{i-1}, q_{i-1} \rangle$ (and all associated triggers) on the $(i-1)$-th configuration $\langle s_2, ctx_2 \rangle$ in $e(r, \langle u, q \rangle)$. Furthermore, from the first two claims, we know that the query and the associated triggers have all been authorized by $dbEnf$. Finally, since $c_{i-1} \neq \langle u_{i-1}, \textbf{skip} \rangle$, we know that the execution of $x \leftarrow q_{i-1}$ (and of the associated triggers) does not throw security exceptions according to the WhileSql semantics. From the induction hypothesis, it follows that $s_1 = s_2$ and $inTrigger(ctx_1) = inTrigger(ctx_2) = \bot$. From this and the determinism of the database operational semantics, it follows that $s = s'$ and $inTrigger(ctx) = inTrigger(ctx') = \bot$.

This completes the proof of the third claim.

Proof of (4). We now prove our fourth claim. Namely that for all $1 \leq i < |\mu(e(r, \langle u, q \rangle))| - 1$, if $secEx(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = \bot$ and $Ex(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = \emptyset$, then the result of the i-th query $\langle u_i, q_i \rangle$ in $\mu(e(r, \langle u, q \rangle))$ is k iff the i-th step in r_1 is associated with the label $\langle u', q_i, k \rangle$, where $u' = Usr(u_i, x \leftarrow q_i)$. Let i be a value such that $1 \leq i < |\mu(e(r, \langle u, q \rangle))| - 1$, $secEx(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = \bot$, and $Ex(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = \emptyset$. From the third claim, it follows that the i-th database state in $\mu(e(r, \langle u, q \rangle))$ is $\langle s', ctx' \rangle$ and the database state in the i-th configuration of r_1 is $\langle s', ctx'' \rangle$. From this and WhileSql's semantics, it follows that the result of the query q_i is k and $[\![q]\!](\langle s', ctx'' \rangle, u_i) = \langle s'', k, \epsilon, \epsilon \rangle$. From this, $\langle u', q_i, k \rangle$ is one of the labels associated with the i-th step in r_1.

Proof of (5). We now prove that for all $1 \leq i < |\mu(e(r, \langle u, q \rangle))| - 1$, if $secEx(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = \bot$, $Ex(last(\mu(e(r, \langle u, q \rangle))^{i+1})) \neq \emptyset$, and $triggers(r, i) = \epsilon$ then $Ex(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = K$ iff the i-th step in r_1 is associated with the label $\langle db(u), q_i, \langle \textbf{IntEx}, K \rangle, \epsilon \rangle$. Let i be a value such that $1 \leq i < |\mu(e(r, \langle u, q \rangle))| - 1$, $secEx(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = \bot$, $Ex(last(\mu(e(r, \langle u, q \rangle))^{i+1})) \neq \emptyset$, $triggers(r, i) = \epsilon$, and $K \neq \emptyset$. From the third claim, it follows that the i-th database state in $\mu(e(r, \langle u, q \rangle))$ is $\langle s', ctx' \rangle$ and the database state in the i-th configuration of r_1 is $\langle s', ctx'' \rangle$. From this, $Ex(last(\mu(e(r, \langle u, q \rangle))^{i+1})) = K$, $triggers(r, i) = \epsilon$, and WhileSql's semantics, it follows that

$[\![q]\!](\langle s', ctx'\rangle, u_i)$ is $\langle s'', \dagger, \langle \mathbf{IntEx}, K\rangle, \epsilon\rangle$, where $\langle s', ctx'\rangle$ is the i-th database state in r_1. From this, it follows that $\langle db(u), q_i, \langle \mathbf{IntEx}, K\rangle, \epsilon\rangle$ is the label associated with the i-th step in r_1.

Proof of (6). We now prove that for all $1 \leq i < |\mu(e(r, \langle u, q\rangle))|-1$, if $secEx(last(\mu(e(r, \langle u, q\rangle))^{i+1})) = \bot$, $Ex(last(\mu(e(r, \langle u, q\rangle))^{i+1})) \neq \emptyset$, and $triggers(r, i) = \bar{t}{\cdot}t$ then $Ex(last(\mu(e(r, \langle u, q\rangle))^{i+1})) = K$ iff the i-th step in r_1 is associated with the label $\langle db(u), q_i, \langle t, \mathtt{B}, \mathbf{IntEx}, K\rangle, \tau\rangle$. Let i be a value such that $1 \leq i < |\mu(e(r, \langle u, q\rangle))| - 1$, $secEx(last(\mu(e(r, \langle u, q\rangle))^{i+1})) = \bot$, $Ex(last(\mu(e(r, \langle u, q\rangle))^{i+1})) \neq \emptyset$, $triggers(r, i) = \bar{t}{\cdot}t$, and $K \neq \emptyset$. From the third claim, it follows that the i-th database state in $\mu(e(r, \langle u, q\rangle))$ is $\langle s', ctx'\rangle$ and the database state in the i-th configuration of r_1 is $\langle s', ctx''\rangle$. From this, $Ex(last(\mu(e(r, \langle u, q\rangle))^{i+1})) = K$, $triggers(r, i) = \bar{t}{\cdot}t$, and WhileSql's semantics, it follows that $[\![q]\!](\langle s', ctx'\rangle, u_i)$ is $\langle s'', \dagger, \langle t, \mathtt{B}, \mathbf{IntEx}, K\rangle, \tau\rangle$, where $\langle s', ctx'\rangle$ is the i-th database state in r_1. From this, it follows that $\langle db(u), q_i, \langle t, \mathtt{B}, \mathbf{IntEx}, K\rangle, \tau\rangle$ is the label associated with the i-th step in r_1.

Proof of (7). Finally, we prove our claim that for all $1 \leq i < |\mu(e(r, \langle u, q\rangle))| - 1$, the label $\langle u, q, m, \tau{\cdot}\langle public, t, q\rangle{\cdot}\tau'\rangle$ is associated with the i-th step in r_1 iff the trigger t is successfully executed in response to the i-th query in $\mu(e(r, \langle u, q\rangle))$, its condition is enabled, it has modified the security policy, and t was the $|\tau| + 1$ trigger executed in response to the i-th query to modify the policy. From the third claim, it follows that the i-th database state in $\mu(e(r, \langle u, q\rangle))$ is $\langle s', ctx'\rangle$ and the database state in the i-th configuration of r_1 is $\langle s', ctx''\rangle$. From this and WhileSql's semantics (see the definition of $[\![\cdot]\!](\cdot)$ in Figures 7.8–7.9), the label $\langle u, q, m, \tau{\cdot}\langle public, t, q\rangle{\cdot}\tau'\rangle$ is associated with the i-th step in r_1 iff the trigger t is successfully executed, its condition is enabled, t actually modifies the policy, and all and only the triggers in the labels in τ have been executed before t. □

Proposition D.4. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, ifEnf be an enforcement mechanism for* WhileSql *programs, u be a user, and dbEnf be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = \langle M, dbEnf\rangle$ be the extended configuration and L be the P-LTS. Let r be a run in traces(L), t be a trigger such that $e(r, t)$ is defined, and $\langle u, q\rangle$ be a command. Finally, let r' be the run obtained by extending $e(r, t)$ with all scheduled triggers until either all triggers are executed or an exception is thrown. The run $r_1 = [r, \langle u, q\rangle]^{ifc}$ produced by executing code(ctx, q, u) starting from $\langle mem(ctx, q, u), init\langle s, ctx\rangle\rangle$ with scheduler $\mathcal{S}(ctx, q, u)$, where ctx is the context in the last state of r and s is the last state in r, satisfies the following conditions:*

1. *For $1 \leq i < |\mu(r')|$, the i-th query executed in $\mu(r')$ is $\langle u_i, q_i\rangle$ iff the i-th statement executed in r_1 is $\langle u_i, x \leftarrow q_i\rangle$ if $secEx(last(\mu(r')^{i+1})) = \bot$ and **skip** otherwise.*
2. *For all $1 \leq i \leq |\mu(r')|$, $last(\mu(r')^i) = \langle s, ctx\rangle$ iff the i-th configuration in r_1 is $\langle C, M, \langle s, ctx'\rangle, \mathcal{S}\rangle$ and $inTrigger(ctx) = inTrigger(ctx') = \bot$.*
3. *For all $1 \leq i < |\mu(r')|$, if the i-th query $\langle u_i, q_i\rangle$ in $\mu(r')$ is authorized and no triggers associated with q_i throw integrity or security exceptions (i.e., $secEx(last(\mu(r')^{i+1})) = \bot$ and $Ex(last(\mu(r')^{i+1})) = \emptyset$), then the result of $\langle u_i, q_i\rangle$ in $\mu(r')$ is k iff the i-th step in r_1 is associated with the label $\langle u', q_i, k\rangle$, where $u' = Usr(u_i, x \leftarrow q_i)$.*
4. *For all $1 \leq i < |\mu(r')|$, if the i-th query $\langle u_i, q_i\rangle$ in $\mu(r')$ is authorized (i.e., $secEx(last(\mu(r')^{i+1})) = \bot$) and the query itself has caused an exception (i.e., $Ex(last(\mu(r')^{i+1})) \neq \emptyset$ and $triggers(r', i) = \epsilon$), then $Ex(last(\mu(r')^{i+1})) = K$ iff the i-th step in r_1 is associated with the label $\langle db(u), q_i, \langle \mathbf{IntEx}, K\rangle, \epsilon\rangle$.*
5. *For all $1 \leq i < |\mu(r')|$, if the i-th query $\langle u_i, q_i\rangle$ in $\mu(r')$ is authorized (i.e., $secEx(last(r')^{i+1}) = \bot$) and the trigger t executed in response to q has caused an exception (i.e., $Ex(last(\mu(r')^{i+1})) \neq \emptyset$ and $triggers(r', i) = \bar{t}{\cdot}t$), then $Ex(last(\mu(r')^{i+1})) = K$ iff the i-th step in r_1 is associated with the label $\langle db(u), q_i, \langle t, \mathtt{B}, \mathbf{IntEx}, K\rangle, \tau\rangle$.*
6. *For all $1 \leq i < |\mu(r')|$, the label $\langle u, q, m, \tau{\cdot}\langle public, t, q\rangle{\cdot}\tau'\rangle$ is associated with the i-th step in r_1 iff the trigger t is successfully executed in response to the i-th query in $\mu(r')$, its condition is enabled, and it modified the security policy (and it has been the $|\tau|+1$-th trigger to do so in the i-th step).*

Proof. The proof is similar to that of Proposition D.3. The only insight is that Reduction 7.1 does not add the query $\langle u, q\rangle$ to the list of queries L' (this follows from the fact that $e(r, t)$ is defined and therefore $inTrigger(ctx) = \top$). □

D.1.5 Equivalence-class preservation

Here, we prove that under proper conditions the enforcement mechanism constructed following Reduction 7.1 preserves the equivalence classes (as formalized in Chapter C) for weak indistinguishability.

Proposition D.5. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, ifEnf be an enforcement mechanism for* WhileSql *programs, u be a user, and dbEnf be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = \langle M, dbEnf\rangle$ be the extended configuration and L be the*

P-LTS. Let r, r' be two runs in traces(L) and $\langle u, q\rangle$ be a database command such that $r \cong^{\mathbf{W}}_{P,u} r'$ hold and $e(r, \langle u, q\rangle)$ and $e(r', \langle u, q\rangle)$ are defined. The runs $r_1 = [r, \langle u, q\rangle]^{ifc}$ and $r_2 = [r', \langle u, q\rangle]^{ifc}$ produced by executing code(ctx, q, u) starting from $\langle mem(ctx, q, u), init\langle s, ctx\rangle\rangle$ with scheduler $\mathcal{S}(ctx, q, u)$ and code(ctx', q, u) starting from $\langle mem(ctx', q, u), init\langle s', ctx'\rangle\rangle$ with scheduler $\mathcal{S}(ctx', q, u)$, where $\langle s, ctx\rangle = last(r)$ and $\langle s', ctx'\rangle = last(r')$, satisfy the following conditions:

- *r_1 and r_2 terminate,*
- *$f(r_1, ctx)$ and $f(r_2, ctx')$ are u-epoch-equivalent, where $f(r, ctx)$ is the run obtained from r by replacing the observations produced by the execution of the last statement with ϵ if $inTrigger(ctx) = \bot$, and*
- *the last statement executed in both runs is the same.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, *ifEnf* be an enforcement mechanism for WhileSql programs, u be a user, and *dbEnf* be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = \langle M, dbEnf\rangle$ be the extended configuration and L be the P-LTS. Let r, r' be two runs in $traces(L)$ such that $r \cong^{\mathbf{W}}_{P,u} r'$ hold, $\langle u, q\rangle$ be a database command, and M be a sequence of memories. Finally, let r_1 and r_2 be the runs $r_1 = [r, \langle u, q\rangle]^{ifc}$ and $r_2 = [r', \langle u, q\rangle]^{ifc}$ produced by executing $code(ctx, q, u)$ starting from $\langle mem(ctx, q, u), init\langle s, ctx\rangle\rangle$ with scheduler $\mathcal{S}(ctx, q, u)$ and $code(ctx', q, u)$ starting from $\langle mem(ctx', q, u), init\langle s', ctx'\rangle\rangle$ with scheduler $\mathcal{S}(ctx', q, u)$ respectively, where $\langle s, ctx\rangle = last(r)$ and $\langle s', ctx'\rangle = last(r')$, The reduction constructs programs without **while** statements, and therefore r_1 and r_2 trivially terminate. Similarly, the construction ensures that the last statement executed both in r_1 and r_2 is the same. Indeed, the last statement in both runs is $\langle u, x \leftarrow q\rangle$, which corresponds to the query $\langle u, q\rangle$.

We now prove that $f(r_1, ctx)$ and $f(r_2, ctx')$ are u-epoch-equivalent. Given a run, we denote by $idx(e, r)$, where e is an epoch in $[\![E^{user}]\!](r)$, the position of the epoch e in the run, e.g., $idx(e, r) = 3$ if e is the third epoch in r. Furthermore, we denote by $idx(e, r, u)$ the position of the epoch e in r when restricting ourselves to u-epochs, namely $idx(e, r, u) = k$ iff e is the k-th epoch in r where all commands are executed by u. Let ν be the mapping from $[\![E^{user}]\!](f(r_1, ctx))$ to $[\![E^{user}]\!](f(r_2, ctx'))$ defined as follows: $\nu(e) = e'$, where $idx(e, f(r_1, ctx), u) = idx(e', f(r_2, ctx'), u)$. We claim that ν is a bijection from $[\![E^{user}, S^{user}_u]\!](f(r_1, ctx))$ to $[\![E^{user}, S^{user}_u]\!](f(r_2, ctx'))$. Note that ν is order preserving by construction. To show that $f(r_1, ctx)$ and $f(r_2, ctx')$ are u-epoch-equivalent we just have to show that for all $e_1 \in [\![E^{user}, S^{user}_u]\!](f(r_1, ctx))$, the initial configurations in e_1 and $\nu(e_1)$ are indistinguishable for u and the traces in e_1 and $\nu(e_1)$ are the same.

We now prove that for all $e_1 \in [\![E^{user}, S^{user}_u]\!](f(r_1, ctx))$, the traces in e_1 and $\nu(e_1)$ are the same. The equivalence of the traces in e_1 and $\nu(e_1)$ directly follows from Proposition D.3 and $r \cong^{\mathbf{W}}_{P,u} r'$. Assume, for contradiction's sake, that this is not the case. Namely, there is an epoch e_1 such that the traces in e_1 and $\nu(e_1)$ are different. This may happen for two cases:

1. The commands executed in e_1 and $\nu(e_1)$ are different. From the first two points of Proposition D.3, this happens iff either the queries executed in the portions of r and r' corresponding to e_1 and $\nu(e_1)$ are different or the security decisions for a common query is different in the portions of r and r' corresponding to e_1 and $\nu(e_1)$. Both cases, however, contradict $r \cong^{\mathbf{W}}_{P,u} r'$.
2. The commands executed in e_1 and $\nu(e_1)$ are the same but they produce different labels. There are two cases:
   (a) The different labels are caused by a query that is not an `INSERT` or `DELETE`. From this and the fourth and fifth points in Proposition D.3, this implies that the queries behaved differently in r and r'. This directly contradicts $r \cong^{\mathbf{W}}_{P,u} r'$.
   (b) The different labels are caused by an `INSERT` or `DELETE` query. If the different label is directly associated with the `INSERT` or `DELETE` query, this implies that the queries behaved differently in r and r'. This directly contradicts $r \cong^{\mathbf{W}}_{P,u} r'$. If the different label is caused by one of the triggers, this implies that the policy has been modified in a different way in r and r'. This again contradicts $r \cong^{\mathbf{W}}_{P,u} r'$ (since $r \cong^{\mathbf{W}}_{P,u} r'$ implies that the configuration is the same in the u-projections of r and r').

Note that the only case where Proposition D.3 and $r \cong^{\mathbf{W}}_{P,u} r'$ cannot be used to determine that the labels are the same in e_1 and $\nu(e_1)$ is the last statement $\langle u, q\rangle$. In the above proof, this case does not occur since we reason about $f(r_1, ctx)$ and $f(r_2, ctx')$ (which ignore the result of the last statement) instead of r_1 and r_2.

Finally, we prove, by induction on $idx(e_1, r, u)$, that the initial configurations in e_1 and $\nu(e_1)$ are indistinguishable for u. For the base case, let $idx(e_1, r, u) = 1$. Therefore, $idx(\nu(e_1), r, u) = 1$. Let $\langle M_1, \langle s_1, ctx_1\rangle\rangle$ be the global state at the beginning of e_1 and $\langle M_2, \langle s_2, ctx_2\rangle\rangle$ be the global state at the beginning of $\nu(e_1)$. From Proposition D.3, it follows that the corresponding database states in r and r' are, respectively, $\langle s_1, ctx'_1\rangle$ and $\langle s_2, ctx'_2\rangle$. From this and $r \cong^{\mathbf{W}}_{P,u} r'$, it follows that $s_1 \approx_u s_2$. Furthermore, since e_1 and $\nu(e_1)$ are the first epochs for the user u_1, the memories associated with the user u are all set to the (same) initial memory and from $r \cong^{\mathbf{W}}_{P,u} r'$ and Reduction 7.1 it follows that $M_1{\restriction_u} = M_2{\restriction_u}$. Therefore, $M_1 \approx_u M_2$. Thus, the initial global states in e_1 and $\nu(e_1)$ are

indistinguishable. For the induction step, the same argument holds for the database state, so we just have to prove that the memories are u-equivalent. Let M_1 and M_2 be the memories at the beginning of e_i and $\nu(e_i)$, where $i = idx(e_1, r, u) = idx(\nu(e_1), r, u)$. Due to the construction of Reduction 7.1, the memories associated with the programs that still have to be executed are the same in M_1 and M_2 (and trivially u-indistinguishable). The u-equivalence of the memories associated with the programs that have been executed before e_1 and $\nu(e_1)$ directly follows, instead, from the fact that the labels in all previous epochs (the only ones that may modify memories visible for u) are the same in $f(r_1, ctx)$ and $f(r_2, ctx')$. Therefore, the results of the queries is the same and the memories are u-indistinguishable.

We now prove our claim that ν is a bijection from $[\![E^{user}, S_u^{user}]\!](f(r_1, ctx))$ to $[\![E^{user}, S_u^{user}]\!](f(r_2, ctx'))$. The fact that $|[\![E^{user}, S_u^{user}]\!](f(r_1, ctx))| = |[\![E^{user}, S_u^{user}]\!](f(r_2, ctx'))|$ directly follows from the construction of Reduction 7.1 and $r \cong^{\mathbf{W}}_{P,u} r'$. Indeed, from $r \cong^{\mathbf{W}}_{P,u} r'$, it follows that $r|_u$ and $r'|_u$ are consistent. Therefore, the commands issued by u are the same in r and r', they are executed in the same order, and they are interleaved with commands from other users in the same way. From this and the fact that Reduction 7.1 constructs programs that exactly mimic the database runs by associating a single statement to a single query (see Proposition D.3), it follows that each sequence of queries issued by u in, say, r is associated to exactly one epoch in r_1 (the same holds for r' and r_2). Furthermore, $\nu(e) \neq \nu(e')$ whenever $e \neq e'$ for any two $e, e' \in [\![E^{user}, S_u^{user}]\!](f(r_1, ctx))$ by construction. Therefore, ν is a bijection from $[\![E^{user}, S_u^{user}]\!](f(r_1, ctx))$ to $[\![E^{user}, S_u^{user}]\!](f(r_2, ctx'))$. □

Proposition D.6. *Let $M = \langle D, \Gamma\rangle$ be a system configuration, ifEnf be an enforcement mechanism for* WhileSql *programs, u be a user, and dbEnf be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = \langle M, dbEnf\rangle$ be the extended configuration and L be the P-LTS. For any run $r \in traces(L)$ and any action $a \in \mathcal{A}_{D,u}$, if the following conditions hold:*

1. *ifEnf is sound for the progress-sensitive variant of Definition 7.2 given E^{user} and S_u^{user},*
2. *ifEnf is u-stable, and*
3. *extend(r, a) is defined,*

a preserves the equivalence class for r, $\langle M, dbEnf\rangle$, and $\cong^{\mathbf{W}}_{\langle M, dbEnf\rangle, u}$.

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, *ifEnf* be an enforcement mechanism for WhileSql programs that is (1) sound, and (2) stable (as required by the proposition), u be a user, and *dbEnf* be the database access control mechanism constructed using Reduction 7.1.

Furthermore, let $P = \langle M, dbEnf\rangle$ be the extended configuration, L be the P-LTS, and $\langle s, ctx\rangle$ be the last state of r, i.e, $\langle s, ctx\rangle = last(r)$. In the following, we use e to refer to *extend* and f to refer to *dbEnf*. With a slight abuse of notation, given a run $r \in traces(L)$ and an action $\langle u, q\rangle$, we write $[r, \langle u, q\rangle]^{ifc}$ to denote the run generated by the WhileSql programs generated according to our reduction and starting from the state and scheduler generated by the reduction algorithm. Note that the i-th statement executed in the run $[r, \langle u, q\rangle]^{ifc}$ corresponds to the i-th command executed in $\mu(e(r, \langle u, q\rangle)$ (and $|[r, \langle u, q\rangle]^{ifc}| = |\mu(e(r, \langle u, q\rangle)|$).

We prove our claim by contradiction. Assume, by contradiction's sake, that there is a run $r \in traces(L)$ and an action $a \in \mathcal{A}_{D,u}$ such that $e(r, a)$ is defined and a does not preserve the equivalence class for r, $\langle M, dbEnf\rangle$, $\cong^{\mathbf{W}}_{\langle M, dbEnf\rangle, u}$, and u. Since $e(r, a)$ is defined, $triggers(last(r)) = \epsilon$. From this, it follows that $triggers(last(r')) = \epsilon$ also for any $r' \in [\![r]\!]^{\mathbf{W}}_{P,u}$ (because r and r' are indistinguishable and, hence, their projections are consistent). From this, it follows that $e(r', a)$ is defined for any any $r' \in [\![r]\!]^{\mathbf{W}}_{P,u}$. With a slight abuse of notation, we denote the action $a \in \mathcal{A}_{D,u}$ also as $\langle u, q\rangle$, where u is the user executing the action and q is the action without the invoker (it is trivial to move between the two notations).

Let r' be a run such that $r \cong^{\mathbf{W}}_{P,u} r'$ holds. From this and Proposition D.5, it follows that the runs $r_1 = [r, \langle u, q\rangle]^{ifc}$ and $r_2 = [r', \langle u, q\rangle]^{ifc}$ produced by executing $code(ctx, q, u)$ starting from $\langle mem(ctx, q, u), init\langle s, ctx\rangle\rangle$ with scheduler $\mathcal{S}(ctx, q, u)$ and $code(ctx', q, u)$ starting from $\langle mem(ctx', q, u), init\langle s', ctx'\rangle\rangle$ with scheduler $\mathcal{S}(ctx', q, u)$, where $\langle s, ctx\rangle = last(r)$ and $\langle s', ctx'\rangle = last(r')$, satisfy the following conditions: (a) r_1 and r_2 terminate, (b) $f(r_1, ctx)$ and $f(r_2, ctx')$ are u-epoch-equivalent, where $f(r, ctx)$ is the run obtained from r by replacing the observations produced by the execution of the last statement with ϵ if $inTrigger(ctx) = \bot$, and (c) the last statement executed in both runs is the same. From this and the stability of *ifEnf*, it follows that $ifEnf(code(ctx, q, u), \mathcal{S}(ctx, q, u), u, \langle s, \epsilon\rangle) = ifEnf(code(ctx', q, u), \mathcal{S}(ctx', q, u), u, \langle s', \epsilon\rangle)$. We claim that $f(s'', ctx'', \langle u, q\rangle) = f(s, ctx, \langle u, q\rangle)$, where $\langle s'', ctx''\rangle = last(r')$. There are two cases:

1. $f(s, ctx, \langle u, q\rangle) = \top$. Let ep^{ifc} (respectively $ep^{ifc'}$) be the last epoch in the run $[r, \langle u, q\rangle]^{ifc}$ (respectively $[r', \langle u, q\rangle]^{ifc}$). Furthermore, let *init* (respectively *init'*) be the index associated with the beginning of ep^{ifc} in $e(r, \langle u, q\rangle)$ (respectively $ep^{ifc'}$ in $e(r', \langle u, q\rangle)$). Note that *init* (and *init'*) are always well-defined (they correspond to the i-th query in the run r, where i is the number of statements executed in $[r, \langle u, q\rangle]^{ifc}$ up to the beginning of ep^{ifc}). Finally, let *idx* be the index of the last query (u, c) inside the epoch ep^{ifc} and $\overline{q}^i(ep^{ifc})$ be all the queries executed

in ep^{ifc} up to the i-th query (included). Note that idx is also the index of the last query (u, c) inside $ep^{ifc'}$ (this directly follows from our reduction).
We remark that $code(ep^{ifc})$ is the same as $code(ep^{ifc'})$ (it follows from the reduction's definition). Furthermore, the order in which the statements in $code(ep^{ifc})$ and $code(ep^{ifc'})$ are executed (this again directly follows from Reduction 7.1). In the rest of the proofs, we ignore the memories in WhileSql runs as they are never read for programs generated by our reduction. Hence, with a slight abuse of notation, we use the same notation for global states and runtime states.
We now show that executing $\langle u, q\rangle$ produces the same results (and exceptions) both in $\langle s, ctx\rangle$ and $\langle s', ctx'\rangle$. From this and $r \cong^{\mathbf{W}}_{P,u} r'$, it directly follows that $e(r, \langle u, q\rangle) \cong^{\mathbf{W}}_{P,u} e(r', \langle u, q\rangle)$ (leading to a contradiction with $\langle u, q\rangle$ not preserving the equivalence class).
From f's construction, $f(s, ctx, \langle u, q\rangle) = \top$, and *ifEnf* soundness, it follows that $PK_u(ep^{ifc}, idx) = [start(ep^{ifc})]_u$, $PK_u(ep^{ifc}, idx-1) = [start(ep^{ifc})]_u$, $PK_u(ep^{ifc'}, idx) = [start(ep^{ifc'})]_u$, and $PK_u(ep^{ifc'}, idx-1) = [start(ep^{ifc'})]_u$. Furthermore, from our reduction, the notion of user-based epochs, and $r \cong^{\mathbf{W}}_{P,u} r'$, then $start(ep^{ifc})$ and $start(ep^{ifc'})$ are data-indistinguishable for u. From this, $[start(ep^{ifc})]_{\approx_u} = [start(ep^{ifc'})]_{\approx_u}$ (since data indistinguishability is an equivalence relation). We refer to $[start(ep^{ifc})]_{\approx_u}$ and $[start(ep^{ifc'})]_{\approx_u}$ as C and C' in the following. From $PK_u(ep^{ifc}, idx) = PK_u(ep^{ifc}, idx-1) = C$, it follows that for all runs starting from $s \in C$, executing all queries in ep^{ifc} up to the idx-th query produces the same observations τ (due to the fact that PK is progress-sensitive). Similarly, from $PK_u(ep^{ifc'}, idx) = PK_u(ep^{ifc'}, idx-1) = C'$, it follows that for all runs starting from $s \in C'$, executing all queries in ep^{ifc} up to the idx-th query produces the same observations τ'. There are a number of cases depending on what is the idx-th command:

- If the idx-th command in ep^{ifc} is a `SELECT` command $\langle u, \texttt{SELECT}\ \varphi\rangle$, then the only observation is $\langle db(u), q, r\rangle$, where r is the `SELECT`'s query result. From this, it follows that (1) for all database states $s \in \diamond(C, \overline{q}^{idx-1}(ep^{ifc}))$, the result of φ is the same, and (2) for all database states $s' \in \diamond(C', \overline{q}^{idx-1}(ep^{ifc'}))$, the result of φ is the same. From this, $[db([\![r^{init}]\!]^{\mathbf{W}}_{P,u})] \subseteq C$, and the fact that all (and only) the queries in $\overline{q}^{idx-1}$ have been executed in r, it follows that for all database states $\langle s_1, ctx_1\rangle \in [db([\![r]\!]^{\mathbf{W}}_{P,u})]$ the result of φ is the same (since $[db([\![r]\!]^{\mathbf{W}}_{P,u})] \subseteq \diamond(C, \overline{q}^{idx-1}(ep^{ifc}))$). From this, $[db([\![r'^{init'}]\!]^{\mathbf{W}}_{P,u})] \subseteq C'$, and the fact that all (and only) the queries in $\overline{q}^{idx-1}$ have been executed in r, it follows that for all database states $(s'_1, ctx'_1) \in [db([\![r']\!]^{\mathbf{W}}_{P,u})]$ the result of φ is the same (since $[db([\![r']\!]^{\mathbf{W}}_{P,u})] \subseteq \diamond(C', \overline{q}^{idx-1}(ep^{ifc'}))$). From this and $C = C'$, it follows that the result of φ is the same in $\langle s, ctx\rangle$ and $\langle s', ctx'\rangle$.
- If the idx-th command in ep^{ifc} is an `INSERT` or `DELETE` command, then the query produces an observation $\langle db(u), q, r, \tau\rangle$, where r is either $\top$ or an exception and τ is a (possibly empty) sequence of public observations associated with `GRANT` and `REVOKE` commands. For simplicity, we restrict ourselves to integrity exceptions caused by `INSERT` and `DELETE` commands (for exceptions related with `GRANT` and `REVOKE` commands see the next case). From $PK_u(ep^{ifc}, idx) = PK_u(ep^{ifc}, idx-1) = C$, it follows that for all states $s \in \diamond(C, \overline{q}^{idx-1}(ep^{ifc}))$, the `INSERT` or `DELETE` command (and the triggers executed in response to it) produces the same results and exceptions (and public observations τ). Similarly, from $PK_u(ep^{ifc'}, idx) = PK_u(ep^{ifc'}, idx-1) = C'$, it follows that for all states $s \in \diamond(C', \overline{q}^{idx-1}(ep^{ifc}))$, the `INSERT` or `DELETE` command (and the triggers executed in response to it) produces the same results and exceptions (and public observations τ). From this, $C = C'$, $[db([\![r^{init}]\!]^{\mathbf{W}}_{P,u})] \subseteq C$, $[db([\![r'^{init'}]\!]^{\mathbf{W}}_{P,u})] \subseteq C'$, $\langle s, ctx\rangle \in [db([\![r]\!]^{\mathbf{W}}_{P,u})]$, $\langle s', ctx'\rangle \in [db([\![r']\!]^{\mathbf{W}}_{P,u})]$, $[db([\![r]\!]^{\mathbf{W}}_{P,u})] \subseteq \diamond(C, \overline{q}^{idx-1}(ep^{ifc}))$, and $[db([\![r']\!]^{\mathbf{W}}_{P,u})] \subseteq \diamond(C', \overline{q}^{idx-1}(ep^{ifc}))$ it follows that the result (and the exceptions) produced by the query is the same in r and r' (and the same holds for τ).
- If the idx-th command in ep^{ifc} is a command modifying the database configuration (i.e., `GRANT`, `REVOKE`, `ADD_USER`, or `CREATE` command), then the only observation is $\langle public, q, r, \epsilon\rangle$, where r is either the positive query result or an exception. From $r \cong^{\mathbf{W}}_{P,U} r'$, it follows that the database configuration is the same in $\langle s, ctx\rangle$ and $\langle s', ctx'\rangle$. From this, the operational semantics of configuration commands from Chapter 6, and the definition of $[\![\cdot]\!](\cdot,\cdot)$, it follows that the result of $\langle u, q\rangle$ is the same in $\langle s, ctx\rangle$ and $\langle s', ctx'\rangle$ (i.e., either $\langle u, q\rangle$ is successfully executed in both cases or its execution throws an exception both in $\langle s, ctx\rangle$ and $\langle s', ctx'\rangle$).

In all cases, we have shown that the result of the query (as well as possible integrity exceptions and additional observations) is the same for $\langle s, ctx\rangle$ and $\langle s', ctx'\rangle$. We claim that the last states in $e(r, \langle u, q\rangle)$ and $e(r', \langle u, q\rangle)$ are configuration indistinguishable. From this and $r \cong^{\mathbf{W}}_{P,u} r'$, it follows that $e(r, \langle u, q\rangle) \cong^{\mathbf{W}}_{P,u} e(r', \langle u, q\rangle)$. From this, it follows that for any $r' \in [\![r]\!]^{\mathbf{W}}_{P,u}$, $e(r', \langle u, q\rangle) \in [\![e(r, \langle u, q\rangle)]\!]^{\mathbf{W}}_{P,u}$. From this, it follows that $\langle u, q\rangle$ preserves the equivalence class for r,

$\langle M, dbEnf\rangle$, and $\cong^{\mathbf{W}}_{\langle M,dbEnf\rangle,u}$.
We now prove our claim that the last states in $e(r,\langle u,q\rangle)$ and $e(r',\langle u,q\rangle)$ are configuration indistinguishable. Note that from $r \cong^{\mathbf{W}}_{P,u} r'$, the last states in r and r' are configuration indistinguishable. For `SELECT`, `INSERT`, and `DELETE` queries, the claim follows trivially since these statements do not modify the database configuration and the last states in r and r' are configuration indistinguishable. For configuration commands, the claim directly follows from (1) the last states in r and r' are configuration indistinguishable, and (2) the commands would either both succeed or both fail in $e(r,\langle u,q\rangle)$ and $e(r',\langle u,q\rangle)$ (as shown above). If the commands fail, the configuration indistinguishability directly follows from the one of the last states in r and r'. If both commands succeed, then the configurations are modified in the same way and the last states in $e(r,\langle u,q\rangle)$ and $e(r',\langle u,q\rangle)$ are configuration indistinguishable.

2. $f(s,ctx,\langle u,q\rangle) = \bot$. From this, $r \cong^{\mathbf{W}}_{P,u} r'$, and $f(s'',ctx'',\langle u,q\rangle) = f(s,ctx,\langle u,q\rangle)$, it follows that $e(r,\langle u,q\rangle) \cong^{\mathbf{W}}_{P,u} e(r',\langle u,q\rangle)$ (because the action has been denied in both runs, the database state has not been modified, and if the last action in r and r' has not been a command issued by the user u the corresponding database states are data-indistinguishable). From this, it follows that for any $r' \in [\![r]\!]^{\mathbf{W}}_{P,u}$, $e(r',\langle u,q\rangle) \in [\![e(r,\langle u,q\rangle)]\!]^{\mathbf{W}}_{P,u}$. From this, it follows that $\langle u,q\rangle$ preserves the equivalence class for r, $\langle M, dbEnf\rangle$, and $\cong^{\mathbf{W}}_{\langle M,dbEnf\rangle,u}$.

This completes the proof of our claim.

We now prove our claim that $f(s'',ctx'',\langle u,q\rangle) = f(s,ctx,\langle u,q\rangle)$, where $\langle s'',ctx''\rangle = last(r')$. Assume, for contradiction's sake, that this is not the case. From the stability of *ifEnf*, it follows that $ifEnf(code(ctx,q,u),\mathcal{S}(ctx,q,u),u,\langle s,\epsilon\rangle) = ifEnf(code(ctx',q,u),\mathcal{S}(ctx',q,u),u,\langle s',\epsilon\rangle)$. From this, $f(s'',ctx'',\langle u,q\rangle) \neq f(s,ctx,\langle u,q\rangle)$, and *dbEnf*'s definition, it follows that (1) $ifEnf(code(ctx,q,u),\mathcal{S}(ctx,q,u),u,\langle s,\epsilon\rangle) = \top$, and (2) $ifEnf(code(ctx',q,u),\mathcal{S}(ctx',q,u),u,\langle s',\epsilon\rangle) = \top$. There are three cases:

- The run $[r,\langle u,q\rangle]^{ifc}$ terminates and the run $[r',\langle u,q\rangle]^{ifc}$ does not terminate. Since $r \cong^{\mathbf{W}}_{P,u} r'$, the last states in r and r' are configuration indistinguishable. From this and $extend(r,\langle u,q\rangle)$ is defined, it follows that $extend(r',\langle u,q\rangle)$ is defined. From this, $[r',\langle u,q\rangle]^{ifc}$ terminates without getting stuck, leading to a contradiction.
- The run $[r',\langle u,q\rangle]^{ifc}$ terminates and the run $[r,\langle u,q\rangle]^{ifc}$ does not terminate. This contradicts the fact that $extend(r,\langle u,q\rangle)$ is defined.
- Both runs terminate without getting stuck and the last observation in $[r,\langle u,q\rangle]^{ifc}$ and $[r',\langle u,q\rangle]^{ifc}$ are different (one results in a security exception and the other does not). Let ep^{ifc} (respectively $ep^{ifc'}$) be the last epoch in the run $[r,\langle u,q\rangle]^{ifc}$ (respectively $[r',\langle u,q\rangle]^{ifc}$). Observe that the last observations belong to the last epochs ep^{ifc} and $ep^{ifc'}$ respectively. From $r \cong^{\mathbf{W}}_{P,u} r'$, it follows that the states at the beginning of the epochs ep^{ifc} and $ep^{ifc'}$ are data indistinguishable. From this and $ifEnf(code(ctx,q,u),\mathcal{S}(ctx,q,u),u,\langle s,\epsilon\rangle) = \top$, the observations produced in ep^{ifc} and $ep^{ifc'}$ must be the same, leading to a contradiction.

This completes the proof of our claim. □

Proposition D.7. *Let $M = \langle D,\Gamma\rangle$ be a system configuration, ifEnf be an enforcement mechanism for* WhileSql *programs, u be a user, and dbEnf be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = \langle M, dbEnf\rangle$ be the extended configuration and L be the P-LTS. For any run $r \in traces(L)$ and any trigger t, if the following conditions hold:*

1. *ifEnf is sound for the progress-sensitive variant of Definition 7.2 given E^{user} and S^{user}_u,*
2. *ifEnf is u-stable, and*
3. *$extend(r,t)$ is defined,*

t preserves the equivalence class for r, $\langle M, dbEnf\rangle$, and $\cong^{\mathbf{W}}_{\langle M,dbEnf\rangle,u}$.

Proof. Let $r \in traces(L)$ be a run and t be a trigger such that $e(r,t)$ is defined. We claim that Reduction 7.1 always authorizes any command $\langle u,q\rangle$ if $inTrigger(ctx) = \top$. This combined with the database semantics (which ensures that only a trigger's condition and action can be executed while executing the trigger t), implies that the execution of the trigger t never throws security exceptions. Therefore, proving that t preserves the equivalence class for r, $\langle M, dbEnf\rangle$, and $\cong^{\mathbf{W}}_{\langle M,dbEnf\rangle,u}$ amounts to proving (1) if t throws an integrity exception in $e(r,t)$ then it throws an integrity exception also in $e(r',t)$ for any $r' \cong^{\mathbf{W}}_{P,u} r$, and (2) $last(e(r,t))$ and $last(e(r',t))$ are configuration equivalent for any $r' \cong^{\mathbf{W}}_{P,u} r$.

We prove that if t throws an integrity exception in $e(r,t)$ then it throws an integrity exception also in $e(r',t)$ for any $r' \cong^{\mathbf{W}}_{P,u} r$. Assume, for contradiction's sake, that this is not the case. This implies that the labels produced by executing the statement a in the program computed by Reduction 7.1 differ for r and r'. This implies that executing a (and the associated triggers) leaks information. This, therefore, contradicts the fact that a is authorized (see the proof of Proposition D.6).

We now prove that $last(e(r,t))$ and $last(e(r',t))$ are configuration equivalent for any $r' \cong^{\mathbf{W}}_{P,u} r$. From $r' \cong^{\mathbf{W}}_{P,u} r$, it follows that $last(r)$ and $last(r')$ are configuration equivalent. If t's action is an

`INSERT` or `DELETE` command, the configuration equivalence of $last(e(r,t))$ and $last(e(r',t))$ directly follows from the configuration equivalence of $last(r)$ and $last(r')$. Instead, assume that t's action is a `GRANT` or `REVOKE` command influencing the user u. Furthermore, assume, for contradiction's sake, that $last(e(r,t))$ and $last(e(r',t))$ are not configuration equivalent. There are four cases:

1. t is enabled in both $e(r,t)$ and $e(r,t')$. From this and the configuration equivalence of $last(r)$ and $last(r')$, it follows that $last(e(r,t))$ and $last(e(r',t))$ are configuration equivalent as well, leading to a contradiction.
2. t is enabled in $e(r,t)$ but it is not enabled in $e(r',t)$. There are two cases:
   (a) $auth(last(r),u) = auth(last(e(r,t)),u)$. From this, it follows that the database configuration is the same in $last(r)$ and $last(e(r,t))$. Furthermore, from the fact that t is not enabled in $e(r',t)$ it follows that the database configuration is the same in $last(r')$ and $last(e(r',t))$. From this and the configuration equivalence of $last(r)$ and $last(r')$, it directly follows that $last(e(r,t))$ and $last(e(r',t))$ are configuration equivalent, leading to a contradiction.
   (b) $auth(last(r),u) \neq auth(last(e(r,t)),u)$. From this and the fact that we are executing the trigger, it follows that the query a that fired the trigger has been authorized. From this and $auth(last(r),u) \neq auth(last(e(r,t)),u)$, it follows that $\langle public, t, q, \epsilon\rangle$, where q is the `GRANT` or `REVOKE` command associated with t's action, belonged to the labels produced by executing the statement associated with a in the program defined by Reduction 7.1. From this and the fact that a has been authorized, it follows that $\langle public, t, q, \epsilon\rangle$ must be associated to the execution of a for any indistinguishable run. Therefore, $\langle public, t, q, \epsilon\rangle$ should be derived also from the run in $e(r',t)$ (since it was indistinguishable from r when we executed a). Therefore, t must be enabled also in $e(r',t)$, leading to a contradiction.
3. t is not enabled in $e(r,t)$ but it is enabled in $e(r',t)$. The proof is similar to the previous case.
4. t is not enabled in both $e(r,t)$ and $e(r,t')$. From this and the configuration equivalence of $last(r)$ and $last(r')$, it follows that $last(e(r,t))$ and $last(e(r',t))$ are configuration equivalent as well since the database configuration in $last(r)$ and $last(e(r,t))$ is the same (and the same hold for r' and $e(r',t)$), leading to a contradiction.

Finally, we prove our claim that Reduction 7.1 authorizes any command $\langle u, q\rangle$ if $inTrigger(ctx)$. This directly follows from the fact that the program constructed by Reduction 7.1 contain only *secure* commands (since the insecure ones are replaced with **skip** statements). □

D.1.6 Proof of the main result

We are now ready to prove the main result in this section. Namely that given a sound and stable enforcement mechanism for WhileSql programs, our construction can be used to construct a secure database access control mechanism.

Theorem D.1. *Let M be a system configuration, ifEnf be an enforcement mechanism for* WhileSql *programs, u be a user, and dbEnf be the database access control mechanism constructed using Reduction 7.1. If for all $u \in UID$,(1) ifEnf is sound for the progress-sensitive variant of Definition 7.2 given E^{user} and S_u^{user}, and (2) ifEnf is u-stable, then dbEnf provides data confidentiality with respect to $\langle M, dbEnf\rangle$, u, $\mathcal{ATK}_u$, and $\cong_{u,\langle M,dbEnf\rangle}$ for any user $u \in UID$, where $\mathcal{ATK}_u$ is the attacker model from Chapter 6.*

Proof. Let $M = \langle D, \Gamma\rangle$ be a system configuration, *ifEnf* be an enforcement mechanism for WhileSql programs, $u \in UID$ be a user, and *dbEnf* be the database access control mechanism constructed using Reduction 7.1. Furthermore, we denote by f be mechanism *dbEnf*, by P the extended configuration $\langle M, dbEnf\rangle$, by A the attacker model $\mathcal{ATK}_u$, and by L the P-LTS. Let r be a run in $traces(L)$, i be an integer such that $1 \leq i \leq |r|$, and ϕ be a sentence such that $r, i \vdash_u \phi$ holds in A. We claim that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds. The theorem follows trivially from the claim and Proposition D.2.

We now show that for all $r \in traces(L)$, all i such that $1 \leq i \leq |r|$, and all sentences ϕ such that $r, i \vdash_u \phi$ holds, then also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds. We prove our claim by induction on the length of the derivation $r, i \vdash_u \phi$. In the following, we denote by e the function *extend*.

Base Case: Assume that $|r, i \vdash_u \phi| = 1$. There are a number of cases depending on the rule used to obtain $r, i \vdash_u \phi$.

1. *SELECT Success - 1.* Let i be such that $r^i = r^{i-1} \cdot \langle u, \texttt{SELECT}, \phi\rangle \cdot \langle s, c\rangle$, where $s = \langle db, U, sec, T, V\rangle$ and $last(r^{i-1}) = \langle s', c'\rangle$, where $s' = \langle db, U, sec, T, V\rangle$. We claim that the result of ϕ is the same for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

   We now prove our claim that the result of ϕ is the same for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$. Let r' be a run in $[\![r^i]\!]^{\mathbf{W}}_{P,u}$. From $r' \cong^{\mathbf{W}}_{P,u} r^i$, it follows that $r'|_u$ and $r^i|_u$ are consistent. From this, it follows that (1) the last action in $r'|_u$ is $\langle u, \texttt{SELECT}, \phi\rangle$, and (2) $res(last(r'|_u)) = res(last(r^i|_u))$. Therefore, $[\phi]^{db'} = [\phi]^{db}$, where db' is the database state in $last(r')$ and db is the database state in $res(last(r^i))$. This completes the proof of our claim.

2. *SELECT Success - 2.* The proof for this case is similar to that of *SELECT Success - 1.*
3. *INSERT Success.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, \bar{t}\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $R(\bar{t})$. Then, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u R(\bar{t}))$ holds. Indeed, in all runs $r' \cong^{\mathbf{W}}_{P,u} r^i$ the last action is $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$. Furthermore, the action has been executed successfully. Therefore, according to the LTS rules, $\bar{t} \in last(r').db(R)$ for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$. From this and the relational calculus semantics, it follows that $[R(\bar{t})]^{last(r').db} = \top$ for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$. Hence, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u R(\bar{t}))$ holds.
4. *INSERT Success - FD.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\neg\exists\overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. From the rule's definition, it follows that the `INSERT` command has not violated the integrity constraint γ. From this, it follows that for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$ the `INSERT` command has not violated γ. From this and ϕ is the weakest precondition for not violating γ (see Proposition C.10), it follows that ϕ holds in all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$.

   We now prove that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds as well. Let r' be a run in $[\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From $r' \cong^{\mathbf{W}}_{P,u} r^{i-1}$, the fact that the `INSERT` command has been authorized in r^i, the stability of *ifEnf*, and Proposition D.5, it follows that executing the `INSERT` command is authorized also in r'. From this and the soundness of *ifEnf*, it follows that the observations produced by executing the `INSERT` command in r' are the same as those produced by executing the command in r^{i-1}. From this, it follows that the `INSERT` command would not violate the integrity constraint γ in r'. From this and ϕ is the weakest precondition for violating γ (see Proposition C.10), it follows that $[\phi]^{last(r').db} = [\phi]^{last(r^{i-1}).db}$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds.
5. *INSERT Success - ID.* The proof of this case is similar to that for the *INSERT Success - FD.*
6. *DELETE Success.* The proof for this case is similar to that of *INSERT Success.*
7. *DELETE Success - ID.* The proof of this case is similar to that for the *INSERT Success - FD.*
8. *INSERT Exception.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSER}, R, \bar{t}\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\neg R(\bar{t})$. From the rule's definition, it follows that the `INSERT` command has violated the integrity constraint γ. From this and the LTS semantics, it follows that (1) $R(\bar{t})$ does not hold before executing the command (otherwise the command would not have thrown an exception), and (2) $R(t)$ does not hold after executing the command (since the system rolled-back to the state before the `INSERT`). Furthermore, for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$ the `INSERT` command has violated γ. For the same argument, it follows that (1) $R(\bar{t})$ does not hold before executing the command in r' (otherwise the command would not have thrown an exception), and (2) $R(t)$ does not hold after executing the command in r' (since the system rolled-back to the state before the `INSERT`). Therefore, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

   We now prove that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds as well. Let r' be a run in $[\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From $r^{i-1} \cong^{\mathbf{W}}_{P,u} r'$, the fact that the command has been authorized in r^i, the stability of *ifEnf*, and Proposition D.5, it follows that executing the `INSERT` command is authorized also in r'. From this and the soundness of *ifEnf*, it follows that the observations produced by executing the `INSERT` command in r' are the same as those produced by executing the command in r^{i-1}. From this, it follows that executing the `INSERT` command would violate the integrity constraint γ in also in r'. From this and the LTS semantics, it follows that $[\phi]^{last(r').db} = [\phi]^{last(r^{i-1}).db} = \bot$ (otherwise the exception would not have been thrown). From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds.
9. *DELETE Exception.* The proof for this case is similar to that of *INSERT Exception.*
10. *INSERT FD Exception.* Let i be such that $r^i = r^{i-1}\cdot\langle u, \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle\cdot s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U, sec, T, V, c'\rangle$, and ϕ be $\exists\overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. From the rule's definition, it follows that the `INSERT` command is authorized and that it has violated an integrity constraint, which we call γ. From this, it follows that for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$ the last `INSERT` command has violated γ. From this and ϕ is the weakest precondition for violating γ (see Proposition C.10), it follows that ϕ holds in all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$.

    We now prove that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds as well. Let r' be a run in $[\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From $r' \cong^{\mathbf{W}}_{P,u} r^{i-1}$, the fact that the `INSERT` command has been authorized in r^i, the stability of *ifEnf*, and Proposition D.5, it follows that executing the `INSERT` command is authorized also in r'. From this and the soundness of *ifEnf*, it follows that the observations produced by executing the `INSERT` command in r' are the same as those produced by executing the command in r^{i-1}. From this and r^i, it follows that the `INSERT` command would violate the integrity constraint γ also in r'. From this and ϕ is the weakest precondition for violating γ (see Proposition C.10),

it follows that $[\phi]^{r'.db} = [\phi]^{r^i.db}$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds.

11. *INSERT ID Exception.* The proof for this case is similar to that of *INSERT FD Exception.*
12. *DELETE FD Exception.* The proof for this case is similar to that of *INSERT FD Exception.*
13. *Integrity Constraint.* The proof of this case follows trivially from the fact that for any state $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and any $\gamma \in \Gamma$, $[\gamma]^{db} = \top$ by definition.
14. *Learn GRANT/REVOKE Backward.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c'\rangle$, and t be a trigger whose WHEN condition is ϕ and whose action is either a GRANT or a REVOKE. From the rule's definition, the trigger has been scheduled. From this, it follows that there has been a previous INSERT or DELETE command that has been authorized and executed such that this trigger has been executed in response to the command. Let k be the index of this command and $r' \in [\![r^k]\!]^{\mathbf{W}}_{P,u}$ (note that r^k is the run obtained from r by dropping the all the operations associated with the k-th command and the subsequent triggers). From $r' \cong^{\mathbf{W}}_{P,u} r^k$, the fact that the operation has been authorized in r^{k+1}, *ifEnf* stability, and Proposition D.5, it follows that the k-th command (and all his triggers) is authorized also in r'. From this and the soundness of *ifEnf*, it follows that the observations produced by executing the command (and the subsequent triggers) are the same for all $r' \in [\![r^k]\!]^{\mathbf{W}}_{P,u}$. From this, the trigger in r^i effectively modifies the database policy, and the fact that policy changes are public events, it follows that the trigger t modifies the policy in response to the command for any (suitable extension of a) run $r' \in [\![r^k]\!]^{\mathbf{W}}_{P,u}$. From this, it follows that the trigger is enabled for any (suitable extension of a) run $r' \in [\![r^k]\!]^{\mathbf{W}}_{P,u}$ (otherwise the corresponding label would not be part of the observations associated with the k-th command). From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$.
15. *Trigger GRANT Disabled Backward.* The proof is similar to that of *Learn GRANT/REVOKE Backward.*
16. *Trigger REVOKE Disabled Backward.* The proof for this case is similar to that of *Trigger GRANT Disabled Backward.*
17. *Trigger INSERT FD Exception.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c'\rangle$, and t be a trigger whose WHEN condition is ϕ and whose action act is a INSERT statement $\langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle$. Furthermore, let ϕ be $\exists \overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$ (and let γ be the constraint associated with ϕ). From the rule definition, the trigger has been scheduled. From this, it follows that there has been a previous INSERT or DELETE command that has been authorized and executed such that this trigger has been executed in response to the command. Let k be the index of this command and r' be a run in $[\![r^k]\!]^{\mathbf{W}}_{P,u}$. From $r' \cong^{\mathbf{W}}_{P,u} r^k$, the fact that the k-th command has been authorized in r^{k+1}, Proposition D.5, and the stability of *ifEnf*, it follows that the k-th command is authorized also in r'. From this and the soundness of *ifEnf*, it follows that the observations produced by executing the command (and the associated triggers) in r' are the same as those produced by executing the command in r^k. From this and r^i, it follows that the trigger would violate the integrity constraint γ also in any suitable extension of r'. From this and ϕ is the weakest precondition for violating γ (see Proposition C.10), it follows that ϕ holds immediately before executing t in the extension of r'. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds.
18. *Trigger INSERT ID Exception.* The proof for this case is similar to that of *Trigger INSERT ID Exception.*
19. *Trigger DELETE ID Exception.* The proof for this case is similar to that of *Trigger DELETE ID Exception.*
20. *Trigger Exception.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec', T, V, c'\rangle$, and t be a trigger whose WHEN condition is ϕ and whose action is act. From the rule definition, the trigger has been scheduled. From this, it follows that there has been a previous INSERT or DELETE command that has been authorized and executed such that this trigger has been executed in response to the command. Let k be the index of this command and r' be a run in $[\![r^k]\!]^{\mathbf{W}}_{P,u}$. From $r' \cong^{\mathbf{W}}_{P,u} r^k$, the fact that the k-th command has been authorized in r^{k+1}, Proposition D.5, and the stability of *ifEnf*, it follows that the k-th command is authorized also in r'. From this and the soundness of *ifEnf*, it follows that the observations produced by executing the command (and the associated triggers) in r' are the same as those produced by executing the command in r^k. From this and the fact that the trigger has thrown an exception, it follows that the trigger is enabled also in the extension of r'. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds.
21. *Trigger Rollback INSERT.* Let i be such that $r^i = r^{i-n-1}{\cdot}\langle u, \texttt{INSERT}, R, \overline{t}\rangle{\cdot}s_1{\cdot}t_1{\cdot}s_2{\cdot}\ldots{\cdot}t_n{\cdot}s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$ and $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and ϕ be $\neg R(\overline{t})$. Furthermore, let $last(r^{i-n-1}) = \langle db', U', sec', T', V', c'\rangle$ and s_n be $\langle db, U, sec, T, V, c\rangle$. From the rule's definition, it follows that one of the triggers fired by the INSERT command has thrown an exception. From

this and the LTS semantics, it follows that (1) $R(\bar{t})$ does not hold before executing the command (otherwise the command would not have had effect), and (2) $R(t)$ does not hold after executing the command (since the system rolled-back to the state before the `INSERT`). From this, it follows that for all runs $r' \in [\![r^i]\!]^{\mathbf{W}}_{P,u}$ the `INSERT` command has violated γ. Therefore, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

22. *Trigger Rollback* ***DELETE***. The proof for this case is similar to that of *Trigger Rollback* ***INSERT***.
23. *Learn from deny – actions.* We prove this case by contradiction. Assume that we can derive $r, i-1 \vdash_u \phi$ using the rule *Learn from deny – actions*. There are $r, r', r'' \in traces(L)$, $1 < i \leq |r|$, $a \in \mathcal{A}_{D,u}$, $s, s' \in \Omega_M$, and ϕ such that: $r^i = r^{i-1} \cdot a \cdot s$, $r' = r'' \cdot a \cdot s'$, $r^{i-1} \cong_{P,u} r''$, $secEx(s') \neq secEx(s)$, $[\phi]^{last(r^{i-1}).db} = \top$, and $[\phi]^{last(r'').db} = \bot$. From $r^i = r^{i-1} \cdot a \cdot s$ and $r' = r'' \cdot a \cdot s'$, it follows that $extend(r^{i-1}, a)$ and $extend(r'', a)$ are well-defined. From this, *ifEnf*'s soundness and stability, Proposition D.6, and $a \in \mathcal{A}_{D,u}$, a preserves the equivalence class for r^{i-1}, $\langle M, dbEnf \rangle$, and $\cong^{\mathbf{W}}_{P,u}$. From $r^{i-1} \cong_{P,u} r''$ and Proposition D.1, it follows that $r^{i-1} \cong^{\mathbf{W}}_{P,u} r''$. From this, $r^{i-1} \cong^{\mathbf{W}}_{P,u} r''$, $extend(r^{i-1}, a) = r^i$, and $extend(r'', a) = r'$, it follows that $r^i \cong^{\mathbf{W}}_{P,u} r'$. This, however, contradicts $secEx(s') \neq secEx(s)$.
24. *Learn from deny – triggers.* The proof is similar to that of the case *Learn from deny – actions*. However, we use Proposition D.7 instead of Proposition D.6.

This completes the proof of the base step.

Induction Step: Assume that the claim hold for any derivation of $r, j \vdash_u \psi$ such that $|r, j \vdash_u \psi| < |r, i \vdash_u \phi|$. We now prove that the claim also holds for $r, i \vdash_u \phi$. There are a number of cases depending on the rule used to obtain $r, i \vdash_u \phi$.

1. *View.* The proof of this case follows trivially from the semantics of the relational calculus extended over views.
2. *Propagate Forward* ***SELECT***. Let i be such that $r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \psi \rangle \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and $last(r^i) = \langle db', U', sec', T', V', c' \rangle$. From the rule, it follows that $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, \texttt{SELECT}, \psi \rangle$ preserves the equivalence class with respect to r^i, P, and $\cong^{\mathbf{W}}_{P,u}$. From this, Lemma C.19, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$, it follows that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i+1 \vdash_u \phi)$ holds.
3. *Propagate Forward* ***GRANT/REVOKE***. Let i be such that $r^{i+1} = r^i \cdot \langle op, u', p, u \rangle \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and $last(r^i) = \langle db', U', sec', T', V', c' \rangle$. From the rule, it follows that $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle op, u', p, u \rangle$ preserves the equivalence class with respect to r^i, P, and $\cong^{\mathbf{W}}_{P,u}$. From this, Lemma C.20, and $secure_{P,u}(r, i \vdash_u \phi)$, it follows that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i+1 \vdash_u \phi)$ holds.
4. *Propagate Forward* ***CREATE***. The proof for this case is similar to that of *Propagate Forward* ***SELECT***.
5. *Propagate Backward* ***SELECT***. Let i be such that $r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \psi \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$ and $last(r^i) = \langle db, U, sec, T, V, c \rangle$. From the rule, it follows that $r, i+1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i+1 \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, \texttt{SELECT}, \psi \rangle$ preserves the equivalence class with respect to r^i, P, and $\cong^{\mathbf{W}}_{P,u}$. From this, Lemma C.19, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i+1 \vdash_u \phi)$, it follows that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.
6. *Propagate Backward* ***GRANT/REVOKE***. Let i be such that $r^{i+1} = r^i \cdot \langle op, u', p, u \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$ and $last(r^i) = \langle db, U, sec, T, V, c \rangle$. From the rule, it follows that $r, i+1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i+1 \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle op, u', p, u \rangle$ preserves the equivalence class with respect to r^i, P, and $\cong^{\mathbf{W}}_{P,u}$. From this, Lemma C.20, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i+1 \vdash_u \phi)$, it follows that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.
7. *Propagate Backward* ***CREATE TRIGGER***. The proof for this case is similar to that of *Propagate Backward* ***SELECT***.
8. *Propagate Backward* ***CREATE VIEW***. Note that the formulae ψ and $replace(\psi, o)$ are semantically equivalent. This is the only difference between the proof for this case and the one for the *Propagate Backward* ***SELECT*** case.
9. *Rollback Backward - 1.* Let i be such that $r^i = r^{i-n-1} \cdot \langle u, op, R, \bar{t} \rangle \cdot s_1 \cdot t_1 \cdot s_2 \cdot \ldots \cdot t_n \cdot s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. Furthermore, let s_n be $\langle db', U', sec', T', V', c' \rangle$ and $last(r^{i-n-1})$ be $\langle db, U, sec, T, V, c \rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$

holds. From Proposition D.7, the trigger t_j preserves the equivalence class with respect to $r^{i-n-1+j}$, P, and $\cong^{\mathbf{W}}_{P,u}$ for any $1 \leq j \leq n$. Therefore, for any $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$, the run $e(v, t_n)$ contains the roll-back. Therefore, for any $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$, the state $last(e(v, t_n))$ is the state just before the action $\langle u, op, R, \bar{t}\rangle$. Let A be the set of system states associated with the roll-back states. It is easy to see that A is the same as $\{sysState(last(t'))|t' \in [\![r^{i-n-1}]\!]^{\mathbf{W}}_{P,u}\}$. From $secure_{P,u}(r, i \vdash_u \phi)$, it follows that ϕ has the same result over all states in A. From this and $A = \{sysState(last(t'))|t' \in [\![r^{i-n-1}]\!]^{\mathbf{W}}_{P,u}\}$, it follows that ϕ has the same result over all states in $\{sysState(last(t'))|t' \in [\![r^{i-n-1}]\!]^{\mathbf{W}}_{P,u}\}$. From this, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-n-1 \vdash_u \phi)$ holds.

10. *Rollback Backward - 2.* Let i be such that $r^i = r^{i-1}\cdot\langle u, op, R, \bar{t}\rangle\cdot s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. From this, Lemma C.18, the fact that the action does not modify the database state, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$, it follows $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$.
11. *Rollback Forward - 1.* Let i be such that $r^i = r^{i-n-1}\cdot\langle u, op, R, \bar{t}\rangle\cdot s_1\cdot t_1\cdot s_2\cdot \ldots \cdot t_n\cdot s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, $t_1, \ldots, t_n \in \mathcal{TRIGGER}_D$, and op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$. Furthermore, let s_n be $\langle db, U, sec, T, V, c\rangle$ and $last(r^{i-n-1})$ be $\langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-n-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-n-1 \vdash_u \phi)$ holds. From Proposition D.7, the trigger t_j preserves the equivalence class with respect to $r^{i-n-1+j}$, P, and $\cong^{\mathbf{W}}_{P,u}$ for any $1 \leq j \leq n$. Independently on the cause of the roll-back (either a security exception or an integrity constraint violation), we claim that the set A of roll-back system states is $\{sysState(last(t'))|t' \in [\![r^{i-n-1}]\!]^{\mathbf{W}}_{P,u}\}$. From $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-n-1 \vdash_u \phi)$, the result of ϕ is the same for all states in A. From this and $A = \{sysState(last(t'))|t' \in [\![r^{i-n-1}]\!]^{\mathbf{W}}_{P,u}\}$, it follows that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

    We now prove our claim. It is trivial to see (from the LTS's semantics) that the set of rollback's states is a subset of $\{sysState(last(v))|v \in [\![r^{i-n-1}]\!]^{\mathbf{W}}_{P,u}\}$. For the other direction, assume, for contradiction's sake, that there is a state in $\{sysState(last(v))|v \in [\![r^{i-n-1}]\!]^{\mathbf{W}}_{P,u}\}$ that is not a rollback state for the runs in $[\![r^i]\!]^{\mathbf{W}}_{P,u}$. This is impossible since all triggers $t_1, \ldots, t_n$ preserve the equivalence class.
12. *Rollback Forward - 2.* Let i be such that $r^i = r^{i-1}\cdot\langle u, op, R, \bar{t}\rangle\cdot s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. From this, Lemma C.18, the fact that the action does not modify the database state, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$, it follows that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.
13. *Propagate Forward `INSERT/DELETE` Success.* Let i be such that $r^i = r^{i-1}\cdot\langle u, op, R, \bar{t}\rangle\cdot s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. From $revise(r^{i-1}, \phi, r^i)$, it follows that the execution of $\langle u, op, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this, Lemma C.18, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.
14. *Propagate Forward `INSERT` Success - 1.* Let i be such that $r^i = r^{i-1}\cdot\langle u, op, R, \bar{t}\rangle\cdot s$, where op is one if $\{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. We claim that the execution of $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$, and Lemma C.18, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

    We now prove our claim that the execution of $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$. From the rule's definition, it follows that $r, i-1 \vdash_u R(\bar{t})$ holds. From this and the soundness of $\vdash_u$, it follows that $[R(\bar{t})]^{last(r^{i-1}).db} = \top$. From $r, i-1 \vdash_u R(\bar{t})$ and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u R(\bar{t}))$ holds. From this and

$[R(\bar{t})]^{last(r^{i-1}).db} = \top$, it follows that $[R(\bar{t})]^{last(v).db} = \top$ for any $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this and the relational calculus semantics, it follows that the execution of $\langle u, op, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$.

15. *Propagate Forward* `DELETE` *Success - 1.* The proof for this case is similar to that of *Propagate Forward* `INSERT` *Success - 1.*
16. *Propagate Backward* `INSERT/DELETE` *Success.* Let i be such that $r^i = r^{i-1}{\cdot}\langle u, op, R, \bar{t}\rangle{\cdot}s$, where $op \in \{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. From $revise(r^{i-1}, \phi, r^i)$, it follows that the execution of $\langle u, op, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this, Lemma C.18, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds.
17. *Propagate Backward* `INSERT` *Success - 1.* Let i be such that $r^i = r^{i-1}{\cdot}\langle u, op, R, \bar{t}\rangle{\cdot}s$, where op is one of $\{\texttt{INSERT}, \texttt{DELETE}\}$, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds. From Proposition D.6, the action $\langle u, op, R, \bar{t}\rangle$ preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. We claim that the execution of $\langle u, \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ for any $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ (the proof of this claim is in the proof of the *Propagate Forward* `INSERT` *Success - 1* case). From this, Lemma C.18, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds.
18. *Propagate Backward* `DELETE` *Success - 1.* The proof for this case is similar to that of *Propagate Forward* `DELETE` *Success - 1.*
19. *Reasoning.* Let Δ be a subset of $\{\delta \mid r, i \vdash_u \delta\}$ and $last(r^i) = \langle db, U, sec, T, V, c\rangle$. From the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \delta)$ holds for any $\delta \in \Delta$. Note that, given any $\delta \in \Delta$, from $r, i \vdash_u \delta$ and the soundness of $\vdash_u$ (see Proposition C.1), it follows that δ holds in $last(r^i)$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \delta)$ holds for any $\delta \in \Delta$, $\Delta \models_{fin} \phi$, and Lemma C.16, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.
20. *Learn* `INSERT` *Backward - 3.* Let i be such that $r^i = r^{i-1}{\cdot}\langle u, \texttt{INSERT}, R, \bar{t}\rangle{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and ϕ be $\neg R(\bar{t})$. Furthermore, we denote by a the `INSERT` action. From the rule, it follows that $r, i-1 \vdash_u \psi$ and $r, i \vdash_u \neg\psi$ hold. From $r, i-1 \vdash_u \psi$, $r, i \vdash_u \neg\psi$, and $\vdash_u$'s soundness, it follows that $[\phi]^{last(r^{i-1})} = \top$ and $[\phi]^{last(r^i)} = \bot$. From this, $last(r^{i-1}) \neq last(r^i)$. From this, $\neg R(\bar{t})$ holds in r^{i-1}. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \psi)$ and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \psi)$. From this, $[\phi]^{last(r^{i-1})} = \top$, and $[\phi]^{last(r^i)} = \bot$, it follows that $[\phi]^{last(r')} = \top$ for any $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_u$ and $[\phi]^{last(e(r',a)} = \bot$ for any $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_u$. From this, $last(r') \neq last(e(r', a))$ for any $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_u$. From this and $a = \langle u, \texttt{INSERT}, R, \bar{t}\rangle$, it follows that $[\neg R(\bar{t})]^{last(r')} = \top$ for any $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_u$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \neg R(\bar{t}))$.
21. *Learn* `DELETE` *Backward - 3.* The proof for this case is similar to that of *Learn* `INSERT` *Backward - 3.*
22. *Propagate Forward Disabled Trigger.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and t be a trigger. Furthermore, let ψ be t's condition where all free variables are replaced with $tpl(last(r^{i-1}))$. From the rule, it follows that $r, i-1 \vdash_u \phi$. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds. Furthermore, from Proposition D.7, it follows that t preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. If the trigger's action is an `INSERT` or a `DELETE` operation, we claim that the operation does not change the content of any table in $tables(\phi)$ for any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. We also claim that executing the trigger t in any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ does not generate security or integrity exceptions. From these claims, the fact that t preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$, Lemma C.21, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$, it follows that also $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

    We now prove our claim. Assume that t's action in either an `INSERT` or a `DELETE` operation. From the rule, it follows that $r, i-1 \vdash_u \neg\psi$. From this and the soundness of $\vdash_u$ (see Proposition C.1), $[\psi]^{last(r^{i-1})} = \bot$. From $r, i-1 \vdash_u \neg\psi$ and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \psi)$ holds. From this and $[\psi]^{last(r^{i-1}).db} = \bot$, it follows that $[\psi]^{v.db} = \bot$ for any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. Therefore, the trigger t is disabled in any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this and the LTS semantics, it follows that t's execution does not change the content of any

table in $tables(\phi)$ for any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$.
We now prove our claim that executing the trigger t in any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ does not generate security or integrity exceptions. This directly follows from (1) executing t in r^{i-1} does not throw an exception, and (2) t preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$.

23. *Propagate Backward Disabled Trigger.* The proof for this case is similar to that of *Propagate Forward Disabled Trigger*.
24. *Learn* `INSERT` *Forward.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and t be a trigger, and ϕ be $R(\overline{t})$. Furthermore, let ψ be t's condition where all free variables are replaced with $tpl(last(r^{i-1}))$. From the rule's definition, it follows that t's action is $\langle u', \texttt{INSERT}, R, \overline{t}\rangle$ and that $r, i-1 \vdash_u \psi$ holds. From the soundness of $\vdash_u$ (see Proposition C.1) and $r, i-1 \vdash_u \psi$, it follows that $[\psi]^{last(r^{i-1}).db} = \top$. From r, $i-1 \vdash_u \psi$ and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \psi)$. From this and $[\psi]^{last(r^{i-1}).db} = \top$, it follows that $[\psi]^{last(r').db} = \top$ for any $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From the rule definition, the trigger has been scheduled. From this, it follows that there has been a previous `INSERT` or `DELETE` command that has been authorized and executed such that this trigger has been executed in response to the command. Let k be the index of this command and r' be a run in $[\![r^k]\!]^{\mathbf{W}}_{P,u}$. From $r' \cong^{\mathbf{W}}_{P,u} r^k$, the fact that the k-th command has been authorized in r^{k+1}, Proposition D.5, and the stability of *ifEnf*, it follows that the k-th command is authorized also in r'. From this and the soundness of *ifEnf*, it follows that the observations produced by executing the command (and the associated triggers) in r' are the same as those produced by executing the command in r^k. From $[\psi]^{last(r').db} = \top$ for any $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$, it follows that t is enabled in any $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this and the fact that the observations are the same, it follows that extending r' with t does not throw exceptions. From this and the fact that t is enabled, it follows that $R(\overline{t})$ holds in $last(r').db$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u R(\overline{t}))$.
25. *Learn* `INSERT` *- FD.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be $\neg\exists\overline{y}, \overline{z}.\, R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$ (and let γ be the associate integrity constraint). Furthermore, let ψ be t's condition where all free variables are replaced with the values in $tpl(last(r^{i-1}))$ and $\langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q})\rangle$ be t's actual action. From the rule, it follows that $r, i-1 \vdash_u \psi$. From this and the soundness of $\vdash_u$, it follows that $[\psi]^{last(r^{i-1}).db} = \top$. From $r, i-1 \vdash_u \psi$ and the induction hypothesis, it follows $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \psi)$. From this and $[\psi]^{last(r^{i-1}).db} = \top$, it follows $[\psi]^{last(r').db} = \top$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From the rule, it follows that executing t in r^i does not violate γ. From this and the fact that ϕ is the weakest precondition for not violating γ (see Proposition C.10), it follows that $[\phi]^{last(r^{i-1}).db} = \top$. From Proposition D.5, the stability, and the soundness of *ifEnf*, it follows that t is executed and authorized also in $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ without throwing exceptions. From this and $[\psi]^{last(r').db} = \top$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$, it follows that t is enabled in $e(r', t)$. From this, Proposition C.10, and the fact that t does not violate γ, it follows that $[\phi]^{last(r^{i-1}).db} = \top$. From this, $[\phi]^{last(r').db} = \top$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$.
26. *Learn* `INSERT` *- FD - 1.* The proof of this case is similar to that of *Learn* `INSERT` *- FD.*
27. *Learn* `INSERT` *- ID.* The proof of this case is similar to that of *Learn* `INSERT` *- FD.* See also the proof of `INSERT` *Success - ID.*
28. *Learn* `INSERT` *- ID - 1.* The proof of this case is similar to that of *Learn* `INSERT` *- ID.*
29. *Learn* `INSERT` *Backward - 1.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be t's actual `WHEN` condition, where all free variables are replaced with the values in $tpl(last(r^{i-1}))$. From the rule's definition, it follows that t is authorized and it does not throw an exception. From this, *dbEnf*'s definition, Proposition D.5, and the stability and soundness of *ifEnf*, the same holds for t in $e(r', t)$ such that $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From the rule, $\vdash_u$'s soundness (see Proposition C.1), and the induction hypothesis, it follows that $[\psi]^{last(r').db} \neq [\psi]^{last(e(r',t)).db}$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this and the fact that t does not throw exceptions in $e(r', t)$, it follows that t is enabled in $e(r', t)$. From this, $[\phi]^{last(r').db} = \top$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \psi)$.
30. *Learn* `INSERT` *Backward - 2.* Let i be such that $r^i = r^{i-1}{\cdot}t{\cdot}s$, where $s = \langle db', U', sec', T', V', c'\rangle \in \Omega_M$, $last(r^{i-1}) = \langle db, U, sec, T, V, c\rangle$, and $t \in \mathcal{TRIGGER}_D$, and ϕ be $\neg R(\overline{t})$. Furthermore, let $act = \langle u', \texttt{INSERT}, R, \overline{t}\rangle$ be t's actual action and γ be t's actual `WHEN` condition obtained by replacing all free variables with the values in $tpl(last(r^{i-1}))$. From the rule's definition, there is a ψ such that $r, i-1 \vdash_u \psi$ and $r, i \vdash_u \neg\psi$. From the rule's definition, it follows that t is authorized and it does not throw an exception. From this, *dbEnf*'s definition, Proposition D.5, and the stability and soundness of *ifEnf*, the same holds for t in $e(r', t)$ such that $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From the rule, $\vdash_u$'s soundness, and the induction hypothesis, it follows that $[\psi]^{last(r').db} \neq$

$[\psi]^{last(e(r',t)).db}$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this and the fact that t does not throw exceptions in $e(r',t)$, it follows that t is enabled in $e(r',t)$ (otherwise, the database state $last(r').db$ and $last(e(r',t)).db$ would have been different). From this, $[\phi]^{last(r').db} = \top$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. Furthermore, $[R(\bar{t})]^{last(r').db} = \bot$ for all $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ (otherwise, for at least one $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ the database states $last(r').db$ and $last(e(r',t)).db$ would have been identical contradicting $[\psi]^{last(r').db} \neq [\psi]^{last(e(r',t)).db}$). From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \neg R(\bar{t}))$.

31. *Learn* `DELETE` *Forward.* The proof of this case is similar to that of *Learn* `INSERT` *Forward.*
32. *Learn* `DELETE` *- ID.* The proof of this case is similar to that of *Learn* `INSERT` *- FD.* See also the proof of `DELETE` *Success - ID.*
33. *Learn* `DELETE` *- ID - 1.* The proof of this case is similar to that of *Learn* `DELETE` *- ID.*
34. *Learn* `DELETE` *Backward - 1.* The proof is similar to that of *Learn* `INSERT` *Backward - 1.*
35. *Learn* `DELETE` *Backward - 2.* The proof is similar to that of *Learn* `INSERT` *Backward - 2.*
36. *Propagate Forward Trigger Action.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where t is a trigger, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds. From Proposition D.7, the trigger t preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. We claim that (1) the execution of t does not alter the content of the tables in $tables(\phi)$, and (2) executing the trigger t in any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ does not generate security or integrity exceptions. From these claims, Lemma C.21, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$, it follows that also the judgment $r, i \vdash_u \phi$ is secure, i.e., $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

    We now prove our claim that the execution of t does not alter the content of the tables in $tables(\phi)$. If the trigger is not enabled, proving the claim is trivial. In the following, we assume the trigger is enabled. There are four cases:

    - t's action is an `INSERT` statement. This case amount to claiming that the `INSERT` statement $\langle u', \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ in case $revise(r^{i-1}, \phi, r^i) = \top$. We proved the claim above in the *Propagate Forward* `INSERT/DELETE` *Success* case.
    - t's action is a `DELETE` statement. The proof is similar to that of the `INSERT` case.
    - t's action is a `GRANT` statement. In this case, the action does not alter the database state and the claim follows trivially.
    - t's action is a `REVOKE` statement. The proof is similar to that of the `GRANT` case.

    We now prove our claim that executing the trigger t in any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ does not generate security or integrity exceptions. This directly follows from (1) executing t in r^{i-1} does not throw an exception, and (2) t preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$.
37. *Propagate Backward Trigger Action.* The proof of this case is similar to *Propagate Backward Trigger Action.*
38. *Propagate Forward* `INSERT` *Trigger Action.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where t is a trigger, $s = \langle db, U, sec, T, V, c\rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c'\rangle$. From the rule's definition, $r, i-1 \vdash_u \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$ holds. From Proposition D.7, the trigger t preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$. We claim that (1) the execution of t does not alter the content of the tables in $tables(\phi)$, and (2) executing the trigger t in any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ does not generate security or integrity exceptions. From these claims, Lemma C.21, and $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \phi)$, it follows that also the judgment $r, i \vdash_u \phi$ is secure, i.e., $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i \vdash_u \phi)$ holds.

    We now prove our claim that the execution of t does not alter the content of the tables in $tables(\phi)$. If the trigger is not enabled, the claim is trivial. In the following, we assume the trigger is enabled. Then, t's action is an `INSERT` statement. This case amount to claiming that the `INSERT` statement $\langle u', \texttt{INSERT}, R, \bar{t}\rangle$ does not alter the content of the tables in $tables(\phi)$ in case $r, i-1 \vdash_u R(\bar{t})$ holds. We proved the claim above in the *Propagate Forward* `INSERT` *Success - 1* case.

    We now prove our claim that executing the trigger t in any run $v \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ does not generate security or integrity exceptions. This directly follows from (1) executing t in r^{i-1} does not throw an exception, and (2) t preserves the equivalence class with respect to r^{i-1}, P, and $\cong^{\mathbf{W}}_{P,u}$.
39. *Propagate Forward* `DELETE` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `INSERT` *Trigger Action.*
40. *Propagate Backward* `INSERT` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `INSERT` *Trigger Action.*
41. *Propagate Backward* `DELETE` *Trigger Action.* The proof of this case is similar to that of *Propagate Forward* `INSERT` *Trigger Action.*

42. *Trigger FD* `INSERT` *Disabled Backward.* Let i be such that $r^i = r^{i-1} \cdot t \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$, $t \in \mathcal{TRIGGER}_D$, $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, and ϕ be t's actual `WHEN` condition obtained by replacing all free variables with the values in $tpl(last(r^{i-1}))$. Furthermore, let $act = \langle u', \texttt{INSERT}, R, (\overline{v}, \overline{w}, \overline{q}) \rangle$ be t's actual action and ψ be $\exists \overline{y}, \overline{z}.R(\overline{v}, \overline{y}, \overline{z}) \wedge \overline{y} \neq \overline{w}$. From the rule's definition, it follows that (1) the trigger has been authorized, (2) and $r, i-1 \vdash_u \psi$ holds, where ψ is the weakest precondition associated with throwing an exception for one of the constraints γ (see Proposition C.10). From $r, i-1 \vdash_u \psi$ and the induction hypothesis, it follows that $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \psi)$. From this and the soundness of $\vdash_u$, it follows that ψ holds for all runs $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$. From this and the fact that the trigger has been executed, it also follows that the exception is not thrown if we extend $r' \in [\![r^{i-1}]\!]^{\mathbf{W}}_{P,u}$ with t. This happens iff the trigger's actual condition ϕ is not is satisfied. From this, $secure_{\cong^{\mathbf{W}}_{P,u}}(r, i-1 \vdash_u \neg\phi)$.
43. *Trigger ID* `INSERT` *Disabled Backward.* The proof of this case is similar to that of *Trigger FD* `INSERT` *Disabled Backward.*
44. *Trigger ID* `DELETE` *Disabled Backward.* The proof of this case is similar to that of *Trigger FD* `INSERT` *Disabled Backward.*

This completes the proof of the induction step.

This completes the proof. □

D.1.7 Security in the Truman model

We now prove that our construction for the Truman model is secure as well. Note that in the following we restrict our setting to non-boolean `SELECT` queries (the only one for which the Truman model semantics has been defined). For boolean queries, we can directly resort to the mechanism produced using Reduction 7.1.

Theorem D.2. *Let M be a system configuration, ifEnf be an enforcement mechanism for* WhileSql *programs, u be a user, and dbEnf be the database access control mechanism for the Truman model semantics constructed using Reduction 7.1 and Reduction 7.2. If for all user $u \in UID$, (1) ifEnf is sound with respect to the progress-sensitive variant of Definition 7.2 given E^{user} and S^{user}_u, and (2) ifEnf is u-stable, then dbEnf is secure and sound for the Truman model, i.e., it satisfies Definitions 3.8 and 3.18.*

Proof. We show security for a single non-boolean query. The security of the multi-query case follows directly (cf. [165]). Let $\{\overline{x} \mid \varphi\}$ be a non-boolean query, s be a database state, and T be $\{\overline{x} \mid \varphi\}$'s result on s. For each of the tuples in $\overline{t} \in T$, we check whether the boolean query $\varphi[\overline{x} \mapsto \overline{t}]$ is authorized by the mechanism *NTdbEnf* constructed using Reduction 7.1. From Theorem 7.1, it follows that *NTdbEnf* provides data confidentiality. From this, it follows that the result of $\varphi[\overline{x} \mapsto \overline{t}]$ is the same for all indistinguishable database states. From this and $\overline{t} \in T$, it follows that $\overline{t} \in [\{\overline{x} \mid \varphi\}]^{s'}$ for all s' u-indistinguishable from s. From this, it follows that *dbEnf* is secure in the sense of Definition 3.8. The soundness of *dbEnf* (see Definition 3.18) directly follows from the fact that the result returned by *NTdbEnf* is always a subset of the result of the query $\{\overline{x} \mid \varphi\}$ for all s' that are indistinguishable from s for the user u. □

D.1.8 From internal attackers to external attackers

Here, we prove the results from Section 7.5.5.

Lemma D.1 states that the computation associated with simple query-only programs does not depend on the memory's content.

Lemma D.1. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs, and s be a system state. For any two sequences of memories $M_1, M_2 \in Mem^*_{UID}$, if $|M_1| = |M_2|$, $|M_1| = |C|$, and for all $1 \leq i \leq |C|$, $u_i = u^i_1$ and $u_i = u^i_2$ (where $C(i) = \langle u_i, c_i \rangle$, $M_1(i) = \langle u^i_1, m^i_1 \rangle$, and $M_2(i) = \langle u^i_2, m^i_2 \rangle$), the following conditions hold:*

- *If there are $n, \tau, C', M'_1, s', ctx', \mathcal{S}'$ such that $\langle C, M_1, \langle s, ctx \rangle, \mathcal{S} \rangle \xrightarrow{\tau}{}^n \langle C', M'_1, \langle s', ctx' \rangle, \mathcal{S}' \rangle$, then there exists an M'_2 such that $\langle C, M_2, \langle s, ctx \rangle, \mathcal{S} \rangle \xrightarrow{\tau}{}^n \langle C', M'_2, \langle s', ctx' \rangle, \mathcal{S}' \rangle$.*
- *If there are $n, \tau, C', M'_2, s', ctx', \mathcal{S}'$ such that $\langle C, M_2, \langle s, ctx \rangle, \mathcal{S} \rangle \xrightarrow{\tau}{}^n \langle C', M'_2, \langle s', ctx' \rangle, \mathcal{S}' \rangle$, then there exists an M'_1 such that $\langle C, M_1, \langle s, ctx \rangle, \mathcal{S} \rangle \xrightarrow{\tau}{}^n \langle C', M'_1, \langle s', ctx' \rangle, \mathcal{S}' \rangle$.*

Proof. The lemma directly follows from (1) the programs in C are simple query-only programs, (2) the result of a query statement depends only on the content of the database, and (3) simple query-only programs do not use program variables inside queries. Hence, the computation does not depend on the memories in the initial configuration. □

For simple query-only programs, we can simplify the security condition by ignoring memories in the global configurations, as stated in Lemma D.2. In the following, given a set of global configurations S, we denote by $DB(S)$ the set of databases in S, i.e., $DB(S) = \{s \mid \exists M.\ \langle M, s\rangle \in S\}$.

Lemma D.2. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs, s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, $u \in UID$ be a user. The sequence of programs C is secure with respect to u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user} for the progress-sensitive variant of Definition 7.2 iff for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^* \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$.*

Proof. Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs, s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, $u \in UID$ be a user.

($\Rightarrow$). Assume that C is secure with respect to u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user} for the progress-sensitive variant of Definition 7.2. Assume, furthermore, for contradiction's sake, that there is an epoch $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^* \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$ such that $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) \neq DB([\langle M_i, s_i\rangle]_{\approx_u})$. This happens iff $PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau) \neq [\langle M_i, s_i\rangle]_{\approx_u}$, contradicting the security of C.

($\Leftarrow$). Assume that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^* \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. Assume, furthermore, for contradiction's sake, that C is not secure with respect to u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user} for the progress-sensitive variant of Definition 7.2. Hence, there is an epoch $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^* \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$ such that $PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau) \neq [\langle M_i, s_i\rangle]_{\approx_u}$. Since $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$, the difference must have been caused by differences in the initial memory. Since the programs in C are simple and query-only, this contradicts Lemma D.1. □

Lemma D.3 states that, for the sequential scheduler, simple query-only programs, and user-based epochs, if the program corresponding to each epoch is secure, then so is the original program.

Lemma D.3. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs, s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, and $u \in UID$ be a user. Furthermore, let r be the longest run obtained starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$. If for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^n \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, (1) the sequence of programs C_i^m is secure with respect to u for $\mathcal{S}$, s_i, M_i^m, E^{user}, S_u^{user} for the progress-sensitive variant of Definition 7.2, and (2) $\mathcal{R}(\langle C_i, M_i, \langle s_i, \epsilon\rangle, \mathcal{S}_i\rangle, \lceil n/2\rceil)$ is a (E^{user}, S_u^{user})-single-epoch configuration , then C is secure with respect to u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user} for the progress-sensitive variant of Definition 7.2, where $\mathcal{R}(\langle C, M, s, \mathcal{S}\rangle, m)$ denotes the global configuration $\langle C^m, M^m, s, \mathcal{S}\rangle$.*

Proof. Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs, s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, and $u \in UID$ be a user. Furthermore, let r be the longest run obtained by starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$. Observe that since r is the longest run obtained starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$, all epochs in $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^n \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E, S]\!](r)$ have an even length (hence $n/2 = \lceil n/2\rceil$). Assume that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^n \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E, S]\!](r)$, (1) the sequence of programs C_i^m is secure with respect to u for $\mathcal{S}$, s_i, M_i^m, E^{user}, S_u^{user}, and (2) $\mathcal{R}(\langle C_i, M_i, \langle s_i, \epsilon\rangle, \mathcal{S}_i\rangle, m)$ is a (E^{user}, S_u^{user})-*single-epoch* configuration, where $m = \lceil n/2\rceil$. Observe that n is always even since executing each program in C_k takes two steps of the global semantics (one application of the M-Eval-Step rule and one application of the M-Eval-End rule) and r is the longest run obtained by starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$. Furthermore, assume, for contradiction's sake, that C is not secure with respect to u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user}. From this and Lemma D.2, it follows that there is an epoch $\langle C_k, M_k, \langle s_k, ctx_k\rangle, \mathcal{S}_k\rangle \xrightarrow{\tau}{}^n \langle C_l, M_l, \langle s_l, ctx_l\rangle, \mathcal{S}_l\rangle$ such that $DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau)) \neq DB([\langle M_k, s_k\rangle]_{\approx_u})$. Let $m = \lceil n/2\rceil$. From (1), we know that C_k^m is secure with respect to u for $\mathcal{S}$, s_k, M_k^m, E^{user}, and S_u^{user}. From (2), we know that $\mathcal{R}(\langle C_k, M_k, \langle s_k, \epsilon\rangle, \mathcal{S}_k\rangle, m)$ is a (E^{user}, S_u^{user})-*single-epoch* configuration. From the security of C_k^m and Lemma D.2, it follows that $DB(PK_{db(u)}(\langle M_k^m, s_k\rangle, C_k^m, \mathcal{S}_k, \tau)) = DB([\langle M_k^m, s_k\rangle]_{\approx_u})$. We claim that $DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau)) = DB(PK_{db(u)}(\langle M_k^m, s_k\rangle, C_k^m, \mathcal{S}_k, \tau))$. From this and $DB([\langle M_k, s_k\rangle]_{\approx_u}) = DB([\langle M_k^m, s_k\rangle]_{\approx_u})$, it follows that $DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau)) = DB([\langle M_k, s_k\rangle]_{\approx_u})$, leading to a contradiction.

We now prove our claim that $DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau)) = DB(PK_{db(u)}(\langle M_k^m, s_k\rangle, C_k^m, \mathcal{S}_k, \tau))$. We first show that $DB(PK_{db(u)}(\langle M_k^m, s_k\rangle, C_k^m, \mathcal{S}_k, \tau)) \subseteq DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau))$. We observe that C_k and C_k^m are simple query-only programs. Furthermore, we know that (1) the first m programs in C_k and C_k^m are identical by construction, (2) they all consist of commands whose observations are visible by the user u (since we consider user-based epochs), and (3) $\tau\restriction_{db(u)} = \tau$ since τ has been produced inside one of the epochs associated with u (which contains only commands associated with u). Let s be a database in $DB(PK_{db(u)}(\langle M_k^m, s_k\rangle, C_k^m, \mathcal{S}_k, \tau))$. From this,

there is a sequence of memories M such that $s \approx_u s_k$, $M \approx_u M_k^m$, and there are $ctx', \tau', C', M', s', \mathcal{S}'$ such that $\langle C_k^m, M, \langle s, \epsilon\rangle, \mathcal{S}_k\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ and $\tau\restriction_{db(u)} = \tau'\restriction_{db(u)}$. We now need to show that $s \in DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau))$. This happens iff there is a memory M_1 such that $M_1 \approx_u M_k$, and there are $ctx', \tau', C', M', s', \mathcal{S}'$ such that $\langle C_k, M_1, \langle s, \epsilon\rangle, \mathcal{S}_k\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ and $\tau\restriction_{db(u)} = \tau'\restriction_{db(u)}$. Without loss of generality, we can assume that $M_k^m = M_1^m$ and $M_1 \approx_u M_k$ (this immediately follows from Lemma D.1 and the fact that the programs in C_k are simple and query-only). Hence, we just need to show that there are $ctx', \tau', C', M', s', \mathcal{S}', i$ such that $\langle C_k, M_1, \langle s, \epsilon\rangle, \mathcal{S}_k\rangle \xrightarrow{\tau'}{}^{i} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ and $\tau\restriction_{db(u)} = \tau'\restriction_{db(u)}$. For $i = 2m \vee i = 2m-1$, it follows from (1–3) that τ' is exactly $\tau\restriction_{db(u)}$. Hence, $DB(PK_{db(u)}(\langle M_k^m, s_k\rangle, C_k^m, \mathcal{S}_k, \tau)) \subseteq DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau))$. For the other direction, $DB(PK_{db(u)}(\langle M_k, s_k\rangle, C_k, \mathcal{S}_k, \tau)) \subseteq DB(PK_{db(u)}(\langle M_k^m, s_k\rangle, C_k^m, \mathcal{S}_k, \tau))$ immediately follows from Lemma D.1 and the fact that any trace produced by a configuration with code C_k^m and database s_k is produced also by a configuration starting from C_k and database s_k. □

Lemma D.4 and Lemma D.5 prove some useful intermediate results for the transformation in Definition 7.4.

Lemma D.4. *Let C be a sequence of simple single-query programs (without `ADD USER` commands), $\mathcal{S}$ be the sequential scheduler 0^∞, $\langle M_1, s_1\rangle$ be a reachable global state, $0 \leq k \leq |C|$, C' be the code in $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle)$, and $m = |C'(0)|$. The following conditions hold (where $\mathcal{T}^{-1}(\tau_1')$ is the trace obtained by (1) dropping the first $|C'(0)|$ observations from τ_1', (2) dropping observations associated with `CREATE VIEW` commands that involve predicate symbols of the form TMP_R, and (3) replacing each occurrence of TMP_R with R):*

1. *If $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_1}{}^{2k} \langle C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}\rangle$, then $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau_1'}{}^{2(k+m+l)} \langle C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}\rangle$, $s_1' = \mathcal{T}^{-1}(s_1'')$, and $\tau_1 = \mathcal{T}^{-1}(\tau_1')$, where l is the number of `CREATE VIEW` queries occurring in C^k.*
2. *If $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau_1'}{}^{2(k+m+l)} \langle C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$, then $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_1}{}^{2k} \langle C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}\rangle$, $s_1' = \mathcal{T}^{-1}(s_1'')$, and $\tau_1 = \mathcal{T}^{-1}(\tau_1')$, where l is the number of `CREATE VIEW` queries occurring in C^k.*

Proof. Let C be a sequence of simple single-query programs (without `ADD USER` commands), $\mathcal{S}$ be the sequential scheduler 0^∞, $\langle M_1, s_1\rangle$ be a reachable global state, $0 \leq k \leq |C|$, C' be the code in $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle)$, and $m = |C'(0)|$.

We now prove the first claim by induction on k. The proof of the second claim is similar.

Base case. We assume $k = 0$. From this, it also follows that $l = 0$. Then, in the computation $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau_1'}{}^{2m} \langle C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}\rangle$ we execute only the program $C'(0)$, which consists only of `REVOKE` commands. The trace τ_1' consists of exactly m observations (one per `REVOKE` command), therefore $\mathcal{T}^{-1}(\tau_1') = \epsilon$ which is equivalent to τ_1. We now show that $\mathcal{T}^{-1}(s_1'') = s_1$. For the database content, this trivially follows from the fact that $C'(0)$ consists of just `REVOKE` commands. For the users, it follows from (1) $\mathcal{T}$ does not change the set of users, and (2) C does not contain `ADD USER` commands. For the triggers and the views, this follows from (1) $\mathcal{T}^{-1}(s_1'')$ just inverts $\mathcal{T}$, and (2) $C'(0)$ consists of just `REVOKE` commands. For the security policy, the only difference may arise from `GRANT`s associated with `SELECT` privileges. Observe, however, that in $C'(0)$ we revoke all privileges to relation schemas of the form TMP_R that do not correspond to privileges on R in *sec*. From this and $\langle M_1, s_1\rangle$ is a reachable global state, all `REVOKE` commands are authorized. Hence, after executing $C'(0)$, there is a privilege $\langle op, u, \langle \texttt{SELECT}, TMP_R\rangle, u'\rangle$ in the final configuration iff $\langle op, u, \langle \texttt{SELECT}, R\rangle, u'\rangle$ is in the initial configuration. Thus, by first removing all `GRANT`s associated with privileges associated with relation schemas of the form R and later replacing all occurrences of TMP_R with R, we obtain again the initial security policy. This completes the proof of the base case.

Induction step. We assume that the claim holds for all $k' < k$ and we show that the claim holds also for k. There are two cases:

- The last command executed in $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_1}{}^{2k} \langle C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}\rangle$ is not a `CREATE VIEW` command. Hence, it is easy to see that we can always execute 2 steps of the computation and therefore execute the program $C'(k+1)$ in the run that starts form $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle)$. There are several cases depending on the actual command.
  If the command in $C'(k+1)$ is a `SELECT` command, then $s_1' = \mathcal{T}^{-1}(s_1'')$ follows from (1) the induction hypothesis and (2) the semantics of `SELECT` commands. Instead, $\tau_1 = \mathcal{T}^{-1}(\tau_1')$ directly follows from the induction hypothesis (since the database where the `SELECT` query is executed is the same as in the original run).
  If the command in $C'(k+1)$ is a `CREATE TRIGGER` command, then $s_1' = \mathcal{T}^{-1}(s_1'')$ and $\tau_1 = \mathcal{T}^{-1}(\tau_1')$ follow from (1) the induction hypothesis, (2) the result of `CREATE TRIGGER` commands

depend only on the database configuration (which is the same in both runs before executing the command), and (3) if the action in the created trigger is a `GRANT` or `REVOKE` and the privilege is `SELECT`, then the action refers to TMP_R iff the original action refers to R.
If the command in $C'(k+1)$ is a `GRANT` or `REVOKE` command, there are two cases depending on whether the privilege is a `SELECT` or not. If the command does not involve a `SELECT` privilege, then $s'_1 = \mathcal{T}^{-1}(s''_1)$ and $\tau_1 = \mathcal{T}^{-1}(\tau'_1)$ follow from (1) the induction hypothesis, and (2) the result of `GRANT` and `REVOKE` commands depend only on the non-`SELECT` privileges in the policy (which are the same in both runs before executing the command). If the command involves a `SELECT` privilege, then in the run starting from $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle$ we attempt to modify the privilege associated with R while in the run starting from $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle)$ we attempt to modify the privilege associated with TMP_R. From the induction hypothesis, the database states in the k-th and $(k+m)$-th configurations in the two runs are equivalent after applying $\mathcal{T}^{-1}$. This means that the privileges over predicate symbols of the form TMP_R mirror exactly those on symbols of the form R. Hence, the result of a `GRANT` or `REVOKE` operation would be the same in both configurations since the only two things that may influence the result are (1) the policy (which is mirrored correctly), and (2) the views of the form TMP_V (which in $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle)$ refer only to the mirrored version of the symbols). Thus, $s'_1 = \mathcal{T}^{-1}(s''_1)$ and $\tau_1 = \mathcal{T}^{-1}(\tau'_1)$.
If the command in $C'(k+1)$ is an `INSERT` or `DELETE` command, there are two cases. If the corresponding triggers do not involve `GRANT` or `REVOKE` commands, then $s'_1 = \mathcal{T}^{-1}(s''_1)$ and $\tau_1 = \mathcal{T}^{-1}(\tau'_1)$ directly follow from the induction hypothesis. If the triggers involve `GRANT` and `REVOKE` commands, then $s'_1 = \mathcal{T}^{-1}(s''_1)$ and $\tau_1 = \mathcal{T}^{-1}(\tau'_1)$ follow from (1) the induction hypothesis, (2) the fact that all triggers that modify `SELECT` privileges have been transformed to refer to the mirrored predicate symbols (of the form TMP_R), and (3) a similar argument to that for `GRANT` and `REVOKE` commands.

- The last command executed in $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_1}{}^{2k} \langle C'_1, M'_1, \langle s'_1, ctx'_1\rangle, \mathcal{S}\rangle$ is a `CREATE VIEW` command. From this, it follows that we can always execute 4 steps of the computation and therefore execute the program $C'(k+1)$ in the run that starts form $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle)$. Observe that now the number of the executed `CREATE VIEW` commands is increased by 1. Hence, the computation is as follows: $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau'_1}{}^{2((k-1)+m+(l-1))} \langle C'''_1, M'''_1, \langle s'''_1, ctx'''_1\rangle, \mathcal{S}\rangle \xrightarrow{\tau''_1}{}^{4} \langle C''_1, M''_1, \langle s''_1, ctx''_1\rangle, \mathcal{S}\rangle$. Thus, we can rewrite the computation as $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau'_1\cdot\tau''_1}{}^{2(k+m+l)} \langle C''_1, M''_1, \langle s''_1, ctx''_1\rangle, \mathcal{S}\rangle$. Furthermore, $s'_1 = \mathcal{T}^{-1}(s''_1)$ and $\tau_1 = \mathcal{T}^{-1}(\tau'_1{\cdot}\tau'')$ directly follow from (1) the induction hypothesis, and (2) the result of `CREATE VIEW` commands depend only on the database configuration (which is the same in both runs before executing the command). Observe that the observations executed by the two `CREATE VIEW` commands are identical modulo the use of temporary predicate symbols.

This completes the proof of the induction step. □

Lemma D.5. *Let C be a sequence of simple single-query programs (without `ADD USER` commands) and $\mathcal{S}$ be the sequential scheduler 0^∞. For any two reachable global states $\langle M_1, s_1\rangle$ and $\langle M_2, s_2\rangle$ (with the same database configuration) and $0 \le k \le |C|$, there are $C'_1, C'_2 M'_1, M'_2, s'_1, s'_2, ctx'_1, ctx'_2, \tau_1, \tau_2$ such that $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_1}{}^{2k} \langle C'_1, M'_1, \langle s'_1, ctx'_1\rangle, \mathcal{S}\rangle$, $\langle C, M_2, \langle s_2, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_2}{}^{2k} \langle C'_2, M'_2, \langle s'_2, ctx'_2\rangle, \mathcal{S}\rangle$, and $\tau_1 = \tau_2$ iff there are $C''_1, C''_2, M''_1, M''_2, s''_1, s''_2, ctx''_1, ctx''_2, \tau'_1, \tau'_2$ such that $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau'_1}{}^{2(k+m+l)} \langle C''_1, M''_1, \langle s''_1, ctx''_1\rangle, \mathcal{S}''_1\rangle$, $\mathcal{T}(\langle C, M_2, \langle s_2, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau'_2}{}^{2(k+m+l)} \langle C''_2, M''_2, \langle s''_2, ctx''_2\rangle, \mathcal{S}''_2\rangle$, and $\tau'_1 = \tau'_2$, where m is the number of statements in $C'(0)$ and l is the number of `CREATE VIEW` queries in C^k.*

Proof. Let C be a sequence of simple single-query programs (without `ADD USER` commands), $\mathcal{S}$ be the sequential scheduler 0^∞, $\langle M_1, s_1\rangle$ and $\langle M_2, s_2\rangle$ be two reachable global states (with the same database configuration), $0 \le k \le |C|$, m is the number of statements in $C'(0)$ and l is the number of `CREATE VIEW` queries in C^k.

(⇒). Assume that there are $C'_1, C'_2 M'_1, M'_2, s'_1, s'_2, ctx'_1, ctx'_2, \tau_1, \tau_2$ such that $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_1}{}^{2k} \langle C'_1, M'_1, \langle s'_1, ctx'_1\rangle, \mathcal{S}\rangle$, $\langle C, M_2, \langle s_2, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_2}{}^{2k} \langle C'_2, M'_2, \langle s'_2, ctx'_2\rangle, \mathcal{S}\rangle$, and $\tau_1 = \tau_2$. Then, by applying Lemma D.4.(1) to both runs, it follows that there are $C''_1, C''_2, M''_1, M''_2, s''_1, s''_2, ctx''_1, ctx''_2, \tau'_1, \tau'_2$ such that $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau'_1}{}^{2(k+m+l)} \langle C''_1, M''_1, \langle s''_1, ctx''_1\rangle, \mathcal{S}''_1\rangle$, $\mathcal{T}(\langle C, M_2, \langle s_2, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau'_2}{}^{2(k+m+l)} \langle C''_2, M''_2, \langle s''_2, ctx''_2\rangle, \mathcal{S}''_2\rangle$, $\tau_1 = \mathcal{T}^{-1}(\tau'_1)$, and $\tau_2 = \mathcal{T}^{-1}(\tau'_2)$. From $\tau_1 = \mathcal{T}^{-1}(\tau'_1)$, $\tau_2 = \mathcal{T}^{-1}(\tau'_2)$, and $\tau_1 = \tau_2$, it follows that the suffixes containing all but the first m observations of τ'_1 and τ'_2 are equivalent. Furthermore, the first m observations in τ_1 and τ_2 are the same since $\langle M_1, s_1\rangle$ and

$\langle M_2, s_2\rangle$ have the same database configuration. Hence, τ_1' and τ_2' are equivalent. This completes the proof of the if direction.

($\Leftarrow$). Assume that there are $C_1'', C_2'', M_1'', M_2'', s_1'', s_2'', ctx_1'', ctx_2'', \tau_1', \tau_2'$ such that $\mathcal{T}(\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau_1'}{}^{2(k+m+l)} \langle C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$, $\mathcal{T}(\langle C, M_2, \langle s_2, \epsilon\rangle, \mathcal{S}\rangle) \xrightarrow{\tau_2'}{}^{2(k+m+l)} \langle C_2'', M_2'', \langle s_2'', ctx_2''\rangle, \mathcal{S}_2''\rangle$, and $\tau_1' = \tau_2'$. By applying Lemma D.4.(2), it follows that there are $C_1', C_2' M_1', M_2', s_1', s_2', ctx_1', ctx_2'$, τ_1, τ_2 such that $\langle C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_1}{}^{2k} \langle C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}\rangle$, $\langle C, M_2, \langle s_2, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_2}{}^{2k} \langle C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}\rangle$, $\tau_1 = \mathcal{T}^{-1}(\tau_1')$, and $\tau_2 = \mathcal{T}^{-1}(\tau_2')$. From $\tau_1 = \mathcal{T}^{-1}(\tau_1')$, $\tau_2 = \mathcal{T}^{-1}(\tau_2')$, and $\tau_1' = \tau_2'$, it follows that $\tau_1 = \tau_2$. This completes the proof of the only if direction. □

In Lemma D.6, we finally connect the security conditions for external and internal attackers.

Lemma D.6. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs (without `ADD USER` commands), s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, $u \in UID$ be a user, and $\langle C', M', \langle s', \epsilon\rangle, \mathcal{S}'\rangle$ be the configuration $\mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$. If $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$ is (E^{user}, S_u^{user})-*single-epoch *and the sequence of programs C' is secure with respect to the user $db(u)$ for $\mathcal{S}'$, s', and M' according to the progress-sensitive variant of Definition 7.3 (extended to handle users in $UID \cup \{db(u) \mid u \in UID\}$), then the sequence of programs C is secure with respect to the attacker u for $\mathcal{S}$, s, M, E^{user}, and S_u^{user} according to the progress-sensitive variant of Definition 7.2.*

Proof. Let $\mathcal{S}$ be the sequential scheduler 0^∞, $C \in Com^*_{UID}$ be a sequence of simple query-only programs (without `ADD USER` commands), s be a system state, $M \in Mem^*_{UID}$ be a sequence of memories, $u \in UID$ be a user, and $\langle C', M', \langle s', \epsilon\rangle, \mathcal{S}'\rangle$ be the configuration $\mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$. We assume that (a) $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$ is (E, S)-*single-epoch*, and (b) the sequence of programs C' is secure with respect to the user $db(u)$ for $\mathcal{S}'$, s', and M' according to the progress-sensitive variant of Definition 7.3. Let r be the longest run obtained from $\langle C', M', \langle s', \epsilon\rangle, \mathcal{S}'\rangle$ (whose length we denote by max) and r_1 be the longest run obtained from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$ (whose length we denote by max_1). From the security of C' with respect to the progress-sensitive variant of Definition 7.3, it follows that whenever $r = \langle C', M', \langle s', \epsilon\rangle, \mathcal{S}'\rangle \xrightarrow{\tau}{}^n \langle C'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle$, then for all $1 \leq i \leq n$, $PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{i-1})) \cap A_{u,sec}(\langle M', s'\rangle) \subseteq PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^i))$, where sec is the security policy in the i-th configuration in r. Observe that $A_{u,sec}(\langle M', s'\rangle)$ never contributes to the security of C'. This follows from the transformation in Definition 7.4 since (a) the programs in C' modify only the `SELECT` privileges associated with predicate symbols of the form TMP_R, and (b) the user u is initially allowed to read all the values in tables TMP_R (for any table R and user u, the security policy sec_{init} in $\mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$ is such that $TMP_R \in auth(sec_{init}, u)$). Therefore, for all $1 \leq i \leq n$, $PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', \epsilon) \subseteq A_{u,sec}(\langle M', s'\rangle)$, where sec is the security policy in the i-th configuration in r. From this and the security of C', it follows that whenever $r = \langle C', M', \langle s', \epsilon\rangle, \mathcal{S}'\rangle \xrightarrow{\tau}{}^n \langle C'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle$, then for all $1 \leq i \leq n$, $PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{i-1})) \subseteq PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^i))$. From this, it follows that $PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}, \epsilon) \subseteq PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}, trace(r^i))$ for all $1 \leq i \leq max$, where max is the length of the longest run obtained by starting from $\langle C', M', \langle s', \epsilon\rangle, \mathcal{S}'\rangle$. Observe also that (1) $PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', \epsilon) = [\langle M', s'\rangle]_{\approx_u}$, and (2) $PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', \tau) \subseteq [\langle M', s'\rangle]_{\approx_u}$ for all τ. Hence, we have that $[\langle M', s'\rangle]_{\approx_u} = PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^i))$ for all $1 \leq i \leq max$. Let $unwrap(K)$ be the set $\{\langle M, \mathcal{T}^{-1}(s)\rangle \mid \langle m{\cdot}M, s\rangle \in K\}$. We claim that for all $0 \leq j \leq max_1$, there is a $j \leq i \leq max$ such that $unwrap(PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^i))) = PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^j))$. By applying our claim to max_1, we obtain that there is a $max_1 \leq i \leq max$ such that $unwrap(PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}, trace(r^i))) = PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1))$. From this and $[\langle M', s'\rangle]_{\approx_u} = PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}, trace(r^i))$ for all $1 \leq i \leq max$, it follows that $unwrap([\langle M', s'\rangle]_{\approx_u}) = PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1))$. From this and $unwrap([\langle M', s'\rangle]_{\approx_u}) = [\langle M, s\rangle]_{\approx_u}$, it follows that $[\langle M, s\rangle]_{\approx_u} = PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1))$. From $[\langle M, s\rangle]_{\approx_u} = PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1))$, $[\langle M, s\rangle]_{\approx_u} \subseteq PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^j))$ for all $1 \leq j \leq max_1$, and $PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^j)) \subseteq PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1))$ for all $1 \leq j \leq max_1$, it follows that $[\langle M, s\rangle]_{\approx_u} = PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^j))$ for all $1 \leq j \leq max_1$. From this, r_1 is the longest computation, and $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$ is (E^{user}, S_u^{user})-*single-epoch*, it follows that whenever $r = \langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^* \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$, for all epochs $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^* \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}_i, \tau) = [\langle M, s\rangle]_{\approx_u}$. Hence, C is secure with respect to the progress-sensitive variant of Definition 7.2.

Auxiliary results. Let C be a sequence of simple query-only programs, M be a sequence of memories, and $\langle C', M', \langle s', \epsilon\rangle, \mathcal{S}\rangle = \mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$. We now show that $unwrap(\{\langle M', s'\rangle\}) = \{\langle M, s\rangle\}$. By construction $M' = \langle admin, m_0\rangle{\cdot}M$, where m_0 is the memory that initializes every variable to 0. Furthermore, $\mathcal{T}^{-1}(s') = s$ (as we have already shown in Lemma D.4). Hence, $unwrap(\{\langle M', s'\rangle\}) = \{\langle M, s\rangle\}$.

We now prove that $unwrap([\langle M', s'\rangle]_{\approx_u}) = [\langle M, s\rangle]_{\approx_u}$. Let $\langle M'', s''\rangle \in [\langle M, s\rangle]_{\approx_u}$. The configuration $\langle M', s'\rangle$ is obtained by extending M with one additional pair $\langle u, m\rangle$ and by extending s. Given that $\langle M'', s''\rangle \approx_u \langle M, s\rangle$, extending $\langle M'', s''\rangle$ in the same way (i.e., by applying the transformation $\mathcal{T}$ given the code C) produces a configuration $\langle M''', s'''\rangle$ that is indistinguishable from $\langle M', s'\rangle$. Indeed, $M''' \approx_u M'$ directly follows from $M''' = \langle admin, m_0\rangle{\cdot}M''$, $M' = \langle admin, m_0\rangle{\cdot}M$, and $M \approx_u M''$. Similarly, $s''' \approx_u s'$ follows from (1) the database configuration is the same in s and s'', (2) the configuration is modified in the same way by $\mathcal{T}$, (3) s and s'' are data indistinguishable, and (4) the transformation $\mathcal{T}$ extends the database only on the relation schemas of the form TMP_R (by setting them to $\emptyset$) and all these relation schemas can be read by any user according to the new database configurations. Hence, $\langle M''', s'''\rangle \in [\langle M', s'\rangle]_{\approx_u}$ and $\langle M'', s''\rangle \in unwrap([\langle M', s'\rangle]_{\approx_u})$. For the other direction, let $\langle M'', s''\rangle \in unwrap([\langle M', s'\rangle]_{\approx_u})$. From this, there is $\langle M''', s'''\rangle \in [\langle M', s'\rangle]_{\approx_u}$. Since $M' = \langle admin, m_0\rangle{\cdot}M$, it follows that $M'' \approx_u M$. Furthermore, it also follows that (1) s''' and s' are data indistinguishable, and (2) the configurations in s''' and s' are the same. From (1), the policy in s''' and s' is a superset of that in s, and *unwrap* drops all relation schemas of the form TMP_R, it follows that s'' and s are data indistinguishable. From (2) and the fact that *unwrap* modifies the policy in the same way in s' and s''', it follows that s'' and s have the same configuration. Hence, $s'' \approx_u s$. From this and $M'' \approx_u M$, it follows that $\langle M'', s''\rangle \in [\langle M, s\rangle]_{\approx_u}$.

Proof of our claim. We now prove our claim that for all $0 \leq j \leq max_1$, there is a $j \leq i \leq max$ such that $unwrap(PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^i))) = PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^j))$. Let j be a value such that $0 \leq j \leq max_1$. Since odd steps in the trace do not produce observations (they are associated with the M-Eval-End rule), we can assume that j is an even number $2k$. We now show our claim for $i = 2(m + k + l)$, where m is $|C'(0)|$ and l is the number of `CREATE VIEW` queries in $C(0){\cdot}\ldots{\cdot}C(k)$.

We now prove that $PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^{2k}))$ is a subset of $unwrap(PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{2(m+k+l)})))$. Let $\langle M'', s''\rangle \in PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^{2k}))$. From this and C be a sequence of simple single-query programs, it follows that executing the first $2k$ steps starting from $\langle C, M'', \langle s'', \epsilon\rangle, \mathcal{S}\rangle$ produces the same trace as executing the first $2k$ steps starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$. From this and Lemma D.5, it follows that executing the first $2(k + m + l)$ steps starting from $\mathcal{T}(\langle C, M'', \langle s'', \epsilon\rangle, \mathcal{S}\rangle)$ produces the same trace as executing the first $2(k + m + l)$ steps starting from $\mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$. We denote by $\langle M_\mathcal{T}, s_\mathcal{T}\rangle$ the global configuration associated with $\mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$ and by $\langle M''_\mathcal{T}, s''_\mathcal{T}\rangle$ the global configuration associated with $\mathcal{T}(\langle C, M'', \langle s'', \epsilon\rangle, \mathcal{S}\rangle)$. Hence, $\langle M''_\mathcal{T}, s''_\mathcal{T}\rangle \in PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{2(m+k)}))$. From this and $unwrap(\{\langle M''_\mathcal{T}, s''_\mathcal{T}\rangle\}) = \{\langle M'', s''\rangle\}$, it follows that $\langle M'', s''\rangle \in unwrap(PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{2(m+k+l)})))$, completing the proof of this direction.

We now prove that $unwrap(PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{2(m+k+l)}))) \subseteq PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^{2k}))$. Let $\langle M'', s''\rangle \in unwrap(PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{2(m+k+l)})))$. From this, it follows that there is an $\langle M''', s'''\rangle \in PK_{db(u)}(\langle M', s'\rangle, C', \mathcal{S}', trace(r^{2(m+k+l)}))$ such that $unwrap(\{\langle M''', s'''\rangle\}) = \langle M'', s''\rangle$. From this and C be a sequence of simple single-query programs, it follows that executing the first $2(m + k + l)$ steps starting from $\mathcal{T}(\langle C, M'', \langle s'', \epsilon\rangle, \mathcal{S}\rangle)$ produces the same trace as executing the first $2(m + k + l)$ steps starting from $\mathcal{T}(\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle)$. From this and Lemma D.5, it follows that executing the first $2k$ steps starting from $\langle C, M'', \langle s'', \epsilon\rangle, \mathcal{S}\rangle$ produces the same trace as executing the first $2k$ steps starting from $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$. Hence, $\langle M'', s''\rangle \in PK_{db(u)}(\langle M, s\rangle, C, \mathcal{S}, trace(r_1^{2k}))$, completing the proof in this direction. □

D.1.9 From progress-sensitive to progress-insensitive security

Here we prove the results from Section 7.5.6. Namely, we connect the progress-sensitive and progress-insensitive versions of Definition 7.2.

In Lemma D.7, we connect the attacker's knowledge in the progress-sensitive and progress-insensitive cases.

Lemma D.7. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com_u^*$ be a sequence of simple query-only programs, $\langle M_0, s_0\rangle$ be a reachable global state, and τ be a trace of observations such that $|\tau| \leq \#_{qry}(C)$ and $\tau\restriction_u = \tau$. If C is safe for $\mathcal{S}$ and $[\langle M_0, s_0\rangle]_u$, then $PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau) = K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$.*

Proof. Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com_u^*$ be a sequence of simple query-only programs, $\langle M_0, s_0\rangle$ be a reachable global state, and τ be a trace such that $|\tau| \leq \#_{qry}(C)$ and $\tau\restriction_u = \tau$. Furthermore, we assume that C is safe for $\mathcal{S}$ and $[\langle M_0, s_0\rangle]_u$.

We first prove that $PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau) \supseteq K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$. Let $\langle M, s\rangle \in K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$. From this, it follows that $M \approx_u M_0$, $s \approx_u s_0$, and for all ctx', τ', C', M', s' such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^* \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, then $\tau\restriction_u \preceq \tau'\restriction_u$ or $\tau\restriction_u \succeq \tau'\restriction_u$. From this, $C \in Com_u^*$,

and $\tau\restriction_u = \tau$, it follows that for all ctx', τ', C', M', s' such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, then $\tau \preceq \tau'$ or $\tau \succeq \tau'$. From this, C is a sequence of simple query-only programs, C's safety, and $|\tau| \leq \#_{qry}(C)$, it follows that there are $n, ctx', \tau', C', M', s'$ such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, $\tau \preceq \tau' \vee \tau \succeq \tau'$, and $|\tau| = |\tau'|$. From $\tau \preceq \tau' \vee \tau \succeq \tau'$ and $|\tau| = |\tau'|$, it follows that $\tau = \tau'$. Hence, there are $n, ctx', \tau', C', M', s'$ such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$ and $\tau = \tau'$. From this, $\tau\restriction_u = \tau$, and $C \in Com_u^*$, it follows that there are $n, ctx', \tau', C', M', s'$ such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$ and $\tau\restriction_u = \tau'\restriction_u$. Hence, $\langle M, s\rangle \in PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$.

We now show that $PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau) \subseteq K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$. Let $\langle M, s\rangle \in PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$. From this, it follows that $M \approx_u M_0$, $s \approx_u s_0$, and there are ctx', τ', C', M', s' such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, and $\tau\restriction_u = \tau'\restriction_u$. From this, $C \in Com_u^*$, and $\tau\restriction_u = \tau$, it follows that there are $n, ctx', \tau', C', M', s'$ such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$ and $\tau = \tau'$. Let r be an arbitrary run such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau''}{}^{m} \langle C'', M'', \langle s'', ctx''\rangle, \mathcal{S}\rangle$. If $n < m$, then $\tau'' \preceq \tau'$. From this and $\tau = \tau'$, $\tau'' \preceq \tau$. If $n = m$, then $\tau'' = \tau'$. From this and $\tau = \tau'$, $\tau'' \preceq \tau$. If $n > m$, then $\tau'' \succeq \tau'$. From this and $\tau = \tau'$, $\tau'' \succeq \tau$. Hence, for all $n, ctx', \tau', C', M', s'$ such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, then $\tau' \preceq \tau \vee \tau' \succeq \tau$. From this, $C \in Com_u^*$, and $\tau\restriction_u = \tau$, it follows that for all $n, ctx', \tau', C', M', s'$ such that $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, then $\tau'\restriction_u \preceq \tau\restriction_u \vee \tau'\restriction_u \succeq \tau\restriction_u$. Hence, $\langle M, s\rangle \in K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$. □

Theorem D.3 proves that the progress-sensitive and progress-insensitive security conditions for external attackers are equivalent for simple query-only programs and sequential schedulers.

Theorem D.3. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com_u^*$ be a sequence of simple query-only programs, and $\langle M_0, s_0\rangle$ be a reachable global state such that C is safe for $\mathcal{S}$ and $[\langle M_0, s_0\rangle]_u$. Then, C is secure with respect to Definition 7.3 for $\mathcal{S}$, M, s, and u iff C is secure with respect to the progress-sensitive variant of Definition 7.3 for $\mathcal{S}$, M, s, and u.*

Proof. Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com_u^*$ be a sequence of simple query-only programs, and $\langle M_0, s_0\rangle$ be a reachable global state such that C is safe for $\mathcal{S}$ and $[\langle M_0, s_0\rangle]_u$.

(⇒). Assume that C is secure with respect to Definition 7.3 for $\mathcal{S}$, M_0, s_0, and u. From this, it follows that whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{u,sec}(M_0, s_0) \subseteq K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^i))$, where the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$. From this, C's safety, and Lemma D.7, it follows that whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{u,sec}(M_0, s_0) \subseteq PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^i))$, where the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$. Hence, C is secure with respect to the progress-sensitive variant of Definition 7.3 for $\mathcal{S}$, M_0, s_0, and u.

(⇐). Assume that C is secure with respect to the progress-sensitive variant of Definition 7.3 for $\mathcal{S}$, M_0, s_0, and u. From this, it follows that whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{u,sec}(M_0, s_0) \subseteq PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^i))$, where the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$. From this, C's safety, and Lemma D.7, it follows that whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{u,sec}(M_0, s_0) \subseteq K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^i))$, where the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$. Hence, C is secure with respect to Definition 7.3 for $\mathcal{S}$, M_0, s_0, and u. □

Theorem D.4 proves that the progress-sensitive and progress-insensitive security conditions for internal attackers are equivalent for simple query-only programs and sequential schedulers.

Theorem D.4. *Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com_u^*$ be a sequence of simple query-only programs, and $\langle M_0, s_0\rangle$ be a reachable global state such that $|C| = |M|$ and for all $1 \leq i \leq |M|$, $M(i) = \langle u_i, m_i\rangle$ and $C(i) = \langle u_i, c_i\rangle$. Furthermore, let r be the longest run obtained starting from $\langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle$. If $C_i^{\lceil n/2\rceil}$ is safe for $\mathcal{S}$ and $[\langle M_i^{\lceil n/2\rceil}, s_i\rangle]_u$ for all $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then C is secure with respect to Definition 7.2 for $\mathcal{S}$, M, s, u, E^{user}, and S_u^{user} iff C is secure with respect to the progress-sensitive variant of Definition 7.2 for $\mathcal{S}$, M, s, u, E^{user}, and S_u^{user}.*

Proof. Let $\mathcal{S}$ be the sequential scheduler 0^∞, u be a user, $C \in Com_u^*$ be a sequence of simple query-only programs, and $\langle M_0, s_0\rangle$ be a reachable global state such that $|C| = |M|$ and for all $1 \leq i \leq |M|$, $M(i) = \langle u_i, m_i\rangle$ and $C(i) = \langle u_i, c_i\rangle$. Furthermore, let r_{max} be the longest run obtained starting from

$\langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle$. Finally, we assume that $C_i^{\lceil n/2\rceil}$ is safe for $\mathcal{S}$ and $[\langle M_i^{\lceil n/2\rceil}, s_i\rangle]_u$ for all $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r_{max})$ (if this is not the case, then the claim trivially holds). Observe that from the assumptions on the memories M_0 and Lemma D.1, we can focus only on the database states and ignore the memories. We claim that for all runs $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$, for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then (1) $DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB(K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau))$, and (2) $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB(PK_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau))$, where C_i' contains only the programs executed inside the epoch, i.e., $C_i' = C_i^{\lceil n/2\rceil}$ and $M_i' = M_i^{\lceil n/2\rceil}$, and $DB(K) = \{s \mid \exists M.\ \langle M, s\rangle \in K\}$. We use this claim in our proofs below.

(⇒). Assume that C is secure with respect to Definition 7.2 for $\mathcal{S}$, M_0, s_0, u, E^{user}, and S_u^{user}. This means that whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$, for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{*} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau) = [\langle M_i, s_i\rangle]_{\approx_u}$. Let r be an arbitrary run $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$. Hence, for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{*} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. From claim (1), it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. From the safety of C_i' and Lemma D.1, by applying Lemma D.7 to each epoch, it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(PK_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. From this and claim (2), it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u}))$. From this and Lemma D.1, it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau) = [\langle M_i, s_i\rangle]_{\approx_u}$. Hence, C is secure with respect to the progress-sensitive variant of Definition 7.2 for $\mathcal{S}$, M_0, s_0, u, E^{user}, and S_u^{user}.

(⇐). Assume that C is secure with respect to the progress-sensitive variant of Definition 7.2 for $\mathcal{S}$, M_0, s_0, u, E^{user}, and S_u^{user}. This means that whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$, for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{*} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau) = [\langle M_i, s_i\rangle]_{\approx_u}$. Let r be an arbitrary run $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$. Hence, for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{*} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. From claim (1), it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(PK_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. From the safety of C_i' and Lemma D.1, by applying Lemma D.7 to each epoch, it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{*} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. From this and claim (2), it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB([\langle M_i, s_i\rangle]_{\approx_u})$. From this and Lemma D.1, it follows that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau) = [\langle M_i, s_i\rangle]_{\approx_u}$. Hence, C is secure with respect to Definition 7.2 for $\mathcal{S}$, M_0, s_0, u, E^{user}, and S_u^{user}.

Proof of claim (1). Let r be an arbitrary run $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$. We now prove our claim that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB(K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau))$, where $M_i' = M_i^{\lceil n/2\rceil}$ and $C_i' = C_i^{\lceil n/2\rceil}$. Let $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ be an arbitrary epoch in $[\![E^{user}, S_u^{user}]\!](r)$, where $M_i' = M_i^{\lceil n/2\rceil}$ and $C_i' = C_i^{\lceil n/2\rceil}$.

We first show that $DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) \subseteq DB(K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau))$. Let $s \in DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau))$. Then, there is a memory M such that $M \approx_u M_i$, $s \approx_u s_i$, and for all ctx', C', M', s', τ' such that $\langle C_i, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, $\tau'\restriction_u \preceq \tau\restriction_u \vee \tau'\restriction_u \succeq \tau\restriction_u$. From this, $M = M''\cdot M'''$, and $C_i = C_i'\cdot C''$, it follows that $M'' \approx_u M_i'$ and for all ctx', C', M', s', τ' such that $\langle C_i', M'', \langle s_i, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, $\tau'\restriction_u \preceq \tau\restriction_u \vee \tau'\restriction_u \succeq \tau\restriction_u$. Hence, $\langle M'', s\rangle \in K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau)$. Therefore, $s \in DB(K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau))$

We now show that $DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) \supseteq DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i', \mathcal{S}_i, \tau))$. Let $s \in DB(K_{db(u)}(\langle M_i', s_i\rangle, C_i', \mathcal{S}_i, \tau))$. From this, there is a memory M such that $M \approx_u M_i'$, $s \approx_u s_i$, and for all ctx', C', M', s', τ' such that $\langle C_i', M, \langle s_i, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, $\tau'\restriction_u \preceq \tau\restriction_u \vee \tau'\restriction_u \succeq \tau\restriction_u$. From this, τ has been produced by executing only the programs in the epoch, C_i' contains exactly the programs in the epoch, and C_i is an extension of C_i', it directly follows that for the memory $M'' = M\cdot M_i''$ (where $M_i = M_i'\cdot M_i''$ such that $|M_i'| = |M|$) then for all ctx', C', M', s', τ' such that

$\langle C_i, M'', \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$, $\tau'\restriction_u \preceq \tau\restriction_u \vee \tau'\restriction_u \succeq \tau\restriction_u$. Hence, $\langle M'', s\rangle \in K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)$ and $s \in DB(K_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau))$.

Proof of claim (2). Let r be an arbitrary run $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau_0}{}^{*} \langle C_n, M_n, \langle s_n, ctx_n\rangle, \mathcal{S}_n\rangle$. We now prove our claim that for all epochs $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ in $[\![E^{user}, S_u^{user}]\!](r)$, then $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) = DB(PK_{db(u)}(\langle M'_i, s_i\rangle, C'_i, \mathcal{S}_i, \tau))$, where $M'_i = M_i^{\lceil n/2\rceil}$ and $C'_i = C_i^{\lceil n/2\rceil}$. Let $\langle C_i, M_i, \langle s_i, ctx_i\rangle, \mathcal{S}_i\rangle \xrightarrow{\tau}{}^{n} \langle C_j, M_j, \langle s_j, ctx_j\rangle, \mathcal{S}_j\rangle$ be an arbitrary epoch in $[\![E^{user}, S_u^{user}]\!](r)$, where $M'_i = M_i^{\lceil n/2\rceil}$ and $C'_i = C_i^{\lceil n/2\rceil}$..

We show that $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) \subseteq DB(PK_{db(u)}(\langle M'_i, s_i\rangle, C'_i, \mathcal{S}_i, \tau))$. Let s be a database state in $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau))$. Then, there is a memory M such that $M \approx_u M_i$, $s \approx_u s_i$, and there are ctx', C', M', s', τ' such that $\langle C_i, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$ and $\tau'\restriction_u = \tau\restriction_u$. Since τ has been produced in the epoch associated with the user u, it follows that $\tau\restriction_u = \tau$. Furthermore, the first $|\tau|$ observations in τ' are produced by the programs in the epoch associated with the user u and therefore $\tau'\restriction_u = \tau'$. Then, there is a run that produces exactly the trace τ and executes only the programs in the epoch (i.e., C'_i). Namely, there are ctx', C', M', s', τ' such that $\langle C'_i, M^{\lceil n/2\rceil}, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$ and $\tau'\restriction_u = \tau\restriction_u$. Hence, $\langle M^{\lceil n/2\rceil}, s\rangle \in PK_{db(u)}(\langle M_i, s_i\rangle, C'_i, \mathcal{S}_i, \tau)$ and $s \in DB(PK_{db(u)}(\langle M_i, s_i\rangle, C'_i, \mathcal{S}_i, \tau))$.

We show that $DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)) \supseteq DB(PK_{db(u)}(\langle M'_i, s_i\rangle, C'_i, \mathcal{S}_i, \tau))$. Let s be a database state in $DB(PK_{db(u)}(\langle M'_i, s_i\rangle, C'_i, \mathcal{S}_i, \tau))$. Then, there is a memory M such that $M \approx_u M'_i$, $s \approx_u s_i$, and there are ctx', C', M', s', τ' such that $\langle C'_i, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$ and $\tau'\restriction_u = \tau\restriction_u$. From this, it follows that there are ctx', C', M', s', τ' such that $\langle C_i, M'', \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}\rangle$ and $\tau'\restriction_u = \tau\restriction_u$ where $M'' = M{\cdot}M''_i$ (where $M_i = M'_i{\cdot}M''_i$ such that $|M'_i| = |M|$). Hence, $\langle M'', s\rangle \in PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau)$ and $s \in DB(PK_{db(u)}(\langle M_i, s_i\rangle, C_i, \mathcal{S}_i, \tau))$. □

D.2 Monitor's transparency

Here, we show that the monitor of Section 7.6 is transparent. In more detail, we prove that the monitor's local semantics is transparent. Furthermore, we also show that for sequential schedulers the monitor's global semantics is transparent as well.

D.2.1 Local semantics

Before proving the correctness of the local semantics, we introduce some terminology and notation. Given an extended WhileSql program c, we denote by $strip(c)$ the program obtained by (1) removing statements of the form **set pc to** l, **dbout**(u, v, o, τ), $\|x \leftarrow q\|$, and **asuser**(u, c') and (2) replacing $[c]$ with c. Furthermore, we write $safe(c)$ iff $strip(c)$ is not of the form $x \leftarrow q$, $x \leftarrow q\,;\,c'$, **asuser**(u, c'), **asuser**$(u, c')\,;\,c''$, **dbout**(u, v, o, τ), **dbout**$(u, v, o, \tau)\,;\,c''$, $\|x \leftarrow q\|$, or $\|x \leftarrow q\|\,;\,c''$.

In Lemma D.8 we prove the correctness of the weakest precondition operator for `INSERT` and `DELETE` commands.

Lemma D.8. *Let ϕ be a sentence that does not refer to views, m be a memory, and db a database state. For all well-formed assignments ν for ϕ, the following facts hold:*

1. $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db}$ *holds iff* $[\phi\nu]^{db'}$ *holds, where* $db'(R) = db(R)$ *for all* $R \neq T$ *and* $db'(T) = db(T) \cup \{\overline{v}(m)\}$.
2. $[wp(\phi, T \ominus \overline{v}(m))\nu]^{db}$ *holds iff* $[\phi\nu]^{db'}$ *holds, where* $db'(R) = db(R)$ *for all* $R \neq T$ *and* $db'(T) = db(T) \setminus \{\overline{v}(m)\}$.
3. $[\neg wp(\phi, c)\nu]^{db} = [wp(\neg\phi, c)\nu]^{db}$.

Proof. Let ϕ be a sentence that does not refer to views, m be a memory, and db a database state.

Proof of (1). Let db' be the database state such that $db'(R) = db(R)$ for all $R \neq T$ and $db'(T) = db(T) \cup \{\overline{v}(m)\}$.

For the *if* direction, we assume that for all well-formed assignments ν for ϕ, $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$. We now prove, by structural induction on ϕ, that $[\phi\nu]^{db'} = \top$. There are two base cases:

- ϕ is $R(\overline{x})$. If $R \neq T$, then $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds since $db(R) = db'(R)$. If $R = T$, then $wp(\phi, T \oplus \overline{v}(m)) = R(\overline{x}) \vee \overline{x} = \overline{v}(m)$. From this and $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$, it follows that $\nu(\overline{x}) \in db(T)$ or $\nu(\overline{x}) = \overline{v}(m)$. If $\nu(\overline{x}) \in db(T)$, then $\nu(\overline{x}) \in db'(T)$ as well. If $\nu(\overline{x}) = \overline{v}(m)$, then $\nu(\overline{x}) \in db'(T)$ by construction. Hence, $[\phi\nu]^{db'} = \top$.
- ϕ is $x_1 = x_2$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.
- ϕ is $x_1 = c$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.

For the induction step, we assume that the claim holds for all sub-formulae of ϕ and we show that it holds for ϕ as well. There are several cases:

- ϕ is $\psi \wedge \gamma$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = wp(\psi, T \oplus \overline{v}(m)) \wedge wp(\gamma, T \oplus \overline{v}(m))$, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $[wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$. From this, ν is a well-formed assignment for ψ and γ, and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \top$ and $[\gamma\nu]^{db'} = \top$. From this, $[(\psi \wedge \gamma)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\psi \vee \gamma$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = wp(\psi, T \oplus \overline{v}(m)) \vee wp(\gamma, T \oplus \overline{v}(m))$, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \top$ or $[wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$. From this, ν is a well-formed assignment for ψ and γ, and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \top$ or $[\gamma\nu]^{db'} = \top$. From this, $[(\psi \vee \gamma)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\neg\psi$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = \neg wp(\psi, T \oplus \overline{v}(m))$, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \bot$. From this and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \bot$. From this, $[(\neg\psi)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\exists x.\ \psi$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = \exists x.\ wp(\psi, T \oplus \overline{v}(m))$, it follows that there is a value $v \in \mathbf{dom}$ such that $[wp(\psi, T \oplus \overline{v}(m))\nu[x \mapsto v]]^{db} = \top$. From this, $\nu[x \mapsto v]$ is a well-formed assignment for ψ, and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \top$. From this, $[(\exists x.\ \psi)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\forall x.\ \psi$. The proof of this case is similar to the $\exists x.\ \psi$ case.

This concludes the proof of the *if* direction.

For the *only if* direction, we assume that for all well-formed assignments ν for ϕ, $[\phi\nu]^{db'} = \top$. We now prove, by structural induction on ϕ, that $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$. There are two base cases:

- ϕ is $R(\overline{x})$. If $R \neq T$, then $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds since $db(R) = db'(R)$. If $R = T$, then $[\phi\nu]^{db'} = \top$. From this and $db'(T) = db(T) \cup \{\overline{v}(m)\}$, it follows that $\nu(\overline{x}) \in db(T)$ or $\nu(\overline{x}) = \overline{v}(m)$. From this, $[(T(\overline{x}) \vee \overline{x} = \overline{v}(m))\nu]^{db} = \top$. Hence, $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.
- ϕ is $x_1 = x_2$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.
- ϕ is $x_1 = c$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.

For the induction step, we assume that the claim holds for all sub-formulae of ϕ and we show that it holds for ϕ as well. There are several cases:

- ϕ is $\psi \wedge \gamma$. From $[\phi\nu]^{db'} = \top$, it follows $[\psi\nu]^{db'} = \top$ and $[\gamma\nu]^{db'} = \top$. From this and the induction hypothesis, $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $[wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$. Hence, $[wp(\psi, T \oplus \overline{v}(m))\nu \wedge wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$ and therefore $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.
- ϕ is $\psi \vee \gamma$. The proof of this case is similar to that of $\psi \wedge \gamma$.
- ϕ is $\neg\psi$. From $[\phi]^{db'} = \top$, it follows that $[\psi]^{db'} = \bot$. From this and the induction hypothesis, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \bot$. From this and $wp(\phi, T \oplus \overline{v}(m)) = \neg wp(\psi, T \oplus \overline{v}(m))$, $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.
- ϕ is $\exists x.\ \psi$. From $[\phi\nu]^{db'} = \top$, it follows that there is a value $v \in \mathbf{dom}$ such that $[\psi\nu[x \mapsto v]]^{db'} = \top$. From this, $\nu[x \mapsto v]$ is a well-formed assignment for ψ, and the induction hypothesis, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu[x \mapsto v]]^{db} = \top$. From this and $wp(\phi, T \oplus \overline{v}(m)) = \exists x.\ wp(\psi, T \oplus \overline{v}(m))$, it follows that $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.
- ϕ is $\forall x.\ \psi$. The proof of this case is similar to the $\exists x.\ \psi$ case.

This completes the proof of the *only if* direction.

Proof of (2). The proof for (2) is similar to that of (1). The only difference is the base case $R(\overline{x})$ in case $R = T$. We show how the proof works for this case. For the *if* direction, assume that $[wp(\phi, T \ominus \overline{v}(m))\nu]^{db} = \top$. From this and $wp(\phi, T \ominus \overline{v}(m)) = T(\overline{x}) \wedge \overline{x} \neq \overline{v}(m)$, it follows that $[(T(\overline{x}) \wedge \overline{x} \neq \overline{v}(m))\nu]^{db} = \top$. From this, $\nu(\overline{x}) \in db(T)$ and $\nu(\overline{x}) \neq \overline{v}(m)$. From this, $\nu(\overline{x}) \in db(T) \setminus \{\overline{v}(m)\}$. Hence, $[T(\overline{x})\nu]^{db'} = \top$.

For the *only if* direction, assume that $[\phi\nu]^{db'} = \top$. From this and $db'(T) = db(T) \setminus \{\overline{v}(m)\}$, it follows that $\nu(\overline{x}) \in db(T)$ and $\nu(\overline{x}) \neq \overline{v}(m)$. From this, $[(T(\overline{x}) \wedge \overline{x} \neq \overline{v}(m))\nu]^{db} = \top$. Hence, $[wp(\phi, T \ominus \overline{v}(m))\nu]^{db} = \top$.

Proof of (3). The third claim immediately follows from (1) and (2). □

Lemma D.9 states that the rules handling the expansion procedure are correct, i.e., that they mimic the operational semantics of the WhileSql statements of the form $x \leftarrow q$.

Lemma D.9. *Let $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, $u \in UID$ be a user, and Δ be a monitor state. Whenever $\langle \Delta, x \leftarrow q, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u} \langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$, then $\langle x \leftarrow q, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_{u} \langle \varepsilon, m'', \langle s'', ctx''\rangle\rangle$ and the following conditions hold: (1) $m'(x) = m''(x)$, (2) $s' = s''$, (3) $triggers(ctx') = triggers(ctx'') = \epsilon$, and (4) $\tau' = \tau''$.*

Proof. Let $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, $u \in UID$ be a user, and Δ be a monitor state. Furthermore, we assume that $\langle \Delta, x \leftarrow q, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u} \langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$. In the computation, we applied the rule F-Expand once, the rule F-ExpandedCode

multiple times, and the rule F-RemoveExpandedCode once. Hence, $\langle \Delta, x \leftarrow q, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u}$ $\langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$ iff $\langle \Delta, [c], m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u}$ $\langle \Delta', [\varepsilon], m', \langle s', ctx'\rangle\rangle$, where $c = expand(\langle s, ctx\rangle, x, q, u)$. This, in turn, happens iff $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u}$ $\langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$, where $c = expand(\langle s, ctx\rangle, x, q, u)$. From *expand*'s definition, it follows that $c = decls(\langle s, ctx\rangle, m, u, x \leftarrow q)$; $body(s, m, u, x \leftarrow q)$, where $decls(\langle s, ctx\rangle, m, u, x \leftarrow q)$ is a sequence of `SELECT` queries and $body(s, m, u, x \leftarrow q)$ is a sequence of **if** statements. We claim that (1) at least one of the conditions of the **if** statements in $body(s, m, u, x \leftarrow q)$ is satisfied, and (2) the conditions in the **if** statements in $body(s, m, u, x \leftarrow q)$ are mutually exclusive, i.e., in each execution we execute only one of the **if** statements. Let c' be the **if** statement associated with the satisfied condition. From *expand*'s definition, $c' =$ **if** $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ **then** $body_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ **else skip**, where $\bar{t}$ is a configuration-consistent execution path. We additionally claim that (3) if $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied, then the configuration-consistent execution path $\bar{t}$ represents an actual execution of the command and the corresponding triggers, and (4) $body_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ correctly implements the semantics of the execution path $\bar{t}$. From (3) and (4), it follows that $\langle x \leftarrow q, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_{u} \langle \varepsilon, m'', \langle s'', ctx''\rangle\rangle$, $m'(x) = m''(x)$, $s' = s''$, $\tau' = \tau''$. Finally, $triggers(ctx') = triggers(ctx'') = \epsilon$, directly follows from the WhileSql and monitor's semantics.

At least one condition is satisfied. Here we prove our claim that at least one condition in the **if** statements in $body(s, m, u, x \leftarrow q)$ is satisfied. First, observe that there is always at least one configuration-consistent execution path. If $\langle q, \texttt{secEx}\rangle$ is configuration-consistent, then the claim trivially holds as there is an **if** statement with condition $\top$, which is trivially satisfied. Assume now that $\langle q, \texttt{secEx}\rangle$ is not configuration-consistent. We observe that the encoding is such that all possible combinations of variables are covered. Namely, for a query q, there are two **if** statements: one checking whether the integrity constraints are satisfied and one checking whether the constraints are not satisfied. Similarly, for a trigger t, there are four possible **if** statements: one checking if the trigger is enabled and the constraints are satisfied, one checking if the trigger is enabled and the constraints are not satisfied, one checking if the trigger is enabled, and one checking if the trigger is not enabled. From these observations, it follows that there is always at least one satisfied condition.

Mutually exclusive conditions. Here we prove our claim that the conditions in the **if** statements in $body(s, m, u, x \leftarrow q)$ are mutually exclusive. Assume, for contradiction's sake, that this is not the case. This requires that there are two distinct configuration-consistent execution paths $\bar{t}$ and $\bar{t}'$ such that both $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t}')$ are satisfied. Since $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t}')$ are $\wedge(map(c_{s,m,u,x,q,\bar{t}}, 1\cdot\ldots\cdot|\bar{t}|))$ and $\wedge(map(c_{s,m,u,x,q,\bar{t}'}, 1\cdot\ldots\cdot|\bar{t}'|))$, this requires that all $c_{s,m,u,x,q,\bar{t}}(i)$ and $c_{s,m,u,x,q,\bar{t}'}(j)$ are satisfied for $1 \leq i \leq |\bar{t}|$ and $1 \leq j \leq |\bar{t}'|$. Let k be the first position where $\bar{t}$ and $\bar{t}'$ differ. There are two cases:

1. $k = 1$. Then, the two paths differ on the initial query q. There are 6 cases depending on the values for $\bar{t}(1)$ and $\bar{t}'(1)$:
   (a) $\bar{t}(1) = \langle q, \texttt{ok}\rangle$ and $\bar{t}(1) = \langle q, \texttt{secEx}\rangle$. Since $\bar{t}$ and $\bar{t}'$ are configuration-consistent paths, it follows that $allowed(s, u, q)$ and $\neg allowed(s, u, q)$, leading to a contradiction.
   (b) $\bar{t}(1) = \langle q, \texttt{ok}\rangle$ and $\bar{t}(1) = \langle q, \texttt{ex}\rangle$. Therefore, $c_{s,m,u,x,q,\bar{t}}(1) = x^{\bar{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{1,\gamma_n}$ and $c_{s,m,u,x,q,\bar{t}'}(1) = \neg(x^{\bar{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}'}_{1,\gamma_n})$. From *decls*'s definition, it follows that $x^{\bar{t}}_{1,\gamma_i}$ and $x^{\bar{t}'}_{1,\gamma_i}$ are respectively initialized by the statements $x^{\bar{t}}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \bar{t}^1)$ and $x^{\bar{t}'}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \bar{t}'^1)$. From this, $\bar{t}(1) = \langle q, \texttt{ok}\rangle$, $\bar{t}(1) = \langle q, \texttt{ex}\rangle$, and *wp*'s definition, it follows that $wp(\gamma_i, \bar{t}^1) = wp(\gamma_i, \bar{t}'^1)$ for all $\gamma_i \in \Gamma$. This combined with $c_{s,m,u,x,q,\bar{t}}(1) = x^{\bar{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{1,\gamma_n}$ and $c_{s,m,u,x,q,\bar{t}'}(1) = \neg(x^{\bar{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}'}_{1,\gamma_n})$, leads to a contradiction (since the result of the queries are the same given that in *decls* we just executed `SELECT` queries).
   (c) $\bar{t}(1) = \langle q, \texttt{secEx}\rangle$ and $\bar{t}(1) = \langle q, \texttt{ok}\rangle$. Since $\bar{t}$ and $\bar{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s, u, q)$ and $allowed(s, u, q)$, leading to a contradiction.
   (d) $\bar{t}(1) = \langle q, \texttt{secEx}\rangle$ and $\bar{t}(1) = \langle q, \texttt{ex}\rangle$. Since $\bar{t}$ and $\bar{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s, u, q)$ and $allowed(s, u, q)$, leading to a contradiction.
   (e) $\bar{t}(1) = \langle q, \texttt{ex}\rangle$ and $\bar{t}(1) = \langle q, \texttt{ok}\rangle$. Then, $c_{s,m,u,x,q,\bar{t}}(1) = \neg(x^{\bar{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{1,\gamma_n})$ and $c_{s,m,u,x,q,\bar{t}'}(1) = x^{\bar{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}'}_{1,\gamma_n}$. From *decls*'s definition, it follows that $x^{\bar{t}}_{1,\gamma_i}$ and $x^{\bar{t}'}_{1,\gamma_i}$ are respectively initialized by the statements $x^{\bar{t}}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \bar{t}^1)$ and $x^{\bar{t}'}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \bar{t}'^1)$. From this, $\bar{t}(1) = \langle q, \texttt{ok}\rangle$, $\bar{t}(1) = \langle q, \texttt{ex}\rangle$, and *wp*'s definition, it follows that $wp(\gamma_i, \bar{t}^1) = wp(\gamma_i, \bar{t}'^1)$ for all $\gamma_i \in \Gamma$. This combined with $c_{s,m,u,x,q,\bar{t}}(1) = \neg(x^{\bar{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{1,\gamma_n})$ and $c_{s,m,u,x,q,\bar{t}'}(1) = x^{\bar{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}'}_{1,\gamma_n}$, leads to a contradiction (since the result of the queries are the same given that in *decls* we just executed `SELECT` queries).

(f) $\overline{t}(1) = \langle q, \texttt{ex}\rangle$ and $\overline{t}(1) = \langle q, \texttt{secEx}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s,u,q)$ and $allowed(s,u,q)$, leading to a contradiction.

2. $k > 1$. Then the two paths differ on the $(k-1)$-th scheduled trigger t. There are 9 cases depending on the values $\overline{t}(k)$ and $\overline{t}'(k)$:
   (a) $\overline{t}(k) = \langle t, \texttt{ok}\rangle$ and $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $allowed(s,u,t(m))$ and $\neg allowed(s,u,t(m))$, leading to a contradiction.
   (b) $\overline{t}(k) = \langle t, \texttt{ok}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ex}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = x^{\overline{t}'}_{k,cond} \wedge \neg(x^{\overline{t}'}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{k,\gamma_n})$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$, $x^{\overline{t}}_{k,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}^{k})$, and $x^{\overline{t}'}_{k,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}'^{k})$ for all $\gamma_i \in \Gamma$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. From $\overline{t}(k) = \langle t, \texttt{ok}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ex}\rangle$, it follows that $wp(\gamma_i, \overline{t}^{k}) = wp(\gamma_i, \overline{t}'^{k})$ for all $\gamma_i \in \Gamma$. Hence, all the variables are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = x^{\overline{t}'}_{k,cond} \wedge \neg(x^{\overline{t}'}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{k,\gamma_n})$ leads to a contradiction (since just one of the conditions could have been satisfied).
   (c) $\overline{t}(k) = \langle t, \texttt{ok}\rangle$ and $\overline{t}(k) = \langle t, \texttt{dis}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, and $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. Hence, all the variables $x^{\overline{t}}_{k,cond}$ and $x^{\overline{t}'}_{k,cond}$ are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$ leads to a contradiction (since just one of the conditions could have been satisfied).
   (d) $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ok}\rangle$. The proof of this case is similar to that of $\overline{t}(k) = \langle t, \texttt{ok}\rangle$ and $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$.
   (e) $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ex}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s,u,t(m))$ and $allowed(s,u,t(m))$, leading to a contradiction.
   (f) $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$ and $\overline{t}(k) = \langle t, \texttt{dis}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, and $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. Hence, all the variables $x^{\overline{t}}_{k,cond}$ and $x^{\overline{t}'}_{k,cond}$ are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$ leads to a contradiction (since just one of the conditions could have been satisfied).
   (g) $\overline{t}(k) = \langle t, \texttt{ex}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ok}\rangle$. The proof of this case is similar to that of $\overline{t}(k) = \langle t, \texttt{ex}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ok}\rangle$.
   (h) $\overline{t}(k) = \langle t, \texttt{ex}\rangle$ and $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $allowed(s,u,t(m))$ and $\neg allowed(s,u,t(m))$, leading to a contradiction.
   (i) $\overline{t}(k) = \langle t, \texttt{ex}\rangle$ and $\overline{t}(k) = \langle t, \texttt{dis}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge \neg(x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n})$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, and $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. Hence, all the variables $x^{\overline{t}}_{k,cond}$ and $x^{\overline{t}'}_{k,cond}$ are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge \neg(x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n})$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$ leads to a contradiction (since just one of the conditions could have been satisfied).
   (j) $\overline{t}(k) = \langle t, \texttt{dis}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ok}\rangle$. The proof of this case is similar to that of $\overline{t}(k) = \langle t, \texttt{ok}\rangle$ and $\overline{t}(k) = \langle t, \texttt{dis}\rangle$.
   (k) $\overline{t}(k) = \langle t, \texttt{dis}\rangle$ and $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$. The proof of this case is similar to that of $\overline{t}(k) = \langle t, \texttt{secEx}\rangle$ and $\overline{t}(k) = \langle t, \texttt{dis}\rangle$.
   (l) $\overline{t}(k) = \langle t, \texttt{dis}\rangle$ and $\overline{t}(k) = \langle t, \texttt{ex}\rangle$. The proof of this case is similar to that of $\overline{t}(k) = \langle t, \texttt{ex}\rangle$ and $\overline{t}(k) = \langle t, \texttt{dis}\rangle$.

Since all cases lead to a contradiction, we proved our claim.

Conditions and execution paths. Here we prove our claim that if $cond_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied,

then the configuration-consistent execution path $\bar{t}$ represents an actual execution of the command and the corresponding triggers. Let $\bar{t}$ be an execution path such that $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied in the local state $\langle m, s\rangle$. We now show that each prefix of $\bar{t}$ corresponds to a computation on the database (i.e., correspond to a run in the database). We show this by induction on the prefix's length.

For the base case, let $\bar{t}'$ be the prefix of length 1. Then, $\bar{t}'$ is $\langle q, r\rangle$, where $r \in \{\texttt{secEx}, \texttt{ok}, \texttt{ex}\}$.

If $r = \texttt{secEx}$, it follows that $allowed(s, u, q) = \bot$. From this and *allowed*'s definition, it follows that executing the command on the database throws a security exception.

If $r = \texttt{ex}$, it follows that $allowed(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the command on the database does not throw a security exception. Furthermore, from $r = \texttt{ex}$, it follows that (1) $\bar{t} = \bar{t}'$, and (2) $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t}) = \neg(x^{\bar{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{1,\gamma_n})$. From (1), (2), and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that one of $\{x^{\bar{t}}_{1,\gamma_1}, \ldots, x^{\bar{t}}_{1,\gamma_n}\}$ evaluates to $\bot$. From this and *decls*'s definition, there is a γ_i such that the result of $\texttt{SELECT}\ wp(\gamma_i, \bar{t}^k)$ on the database s is $\bot$. From this and Lemma D.8, executing the query q throws an integrity exception.

Finally, if $r = \texttt{ok}$, it follows that $allowed(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the command on the database does not throw a security exception. Furthermore, it follows that one of the conjuncts in $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is $x^{\bar{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{1,\gamma_n}$. From (1), (2), and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that all variables in $\{x^{\bar{t}}_{1,\gamma_1}, \ldots, x^{\bar{t}}_{1,\gamma_n}\}$ evaluates to $\bot$. From this and *decls*'s definition, it follows that for all $\gamma_i \in \Gamma$, the result of $\texttt{SELECT}\ wp(\gamma_i, \bar{t}^k)$ on the database s is $\top$. From this and Lemma D.8, executing the query q does not throw an integrity exception.

For the induction step, we assume that all prefixes of length less than k correspond to actual computations. We now show that the same holds for prefixes of length k. Then, $\bar{t}' = \bar{t}'' \cdot \langle t, r\rangle$, where $r \in \{\texttt{secEx}, \texttt{ok}, \texttt{ex}, \texttt{dis}\}$.

If $r = \texttt{secEx}$, it follows that $allowed(s, u, q) = \bot$. From this and *allowed*'s definition, it follows that executing the trigger on the database throws a security exception. Furthermore, $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ contains the conjunct $x^{\bar{t}}_{k,cond}$. From this and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that $x^{\bar{t}}_{k,cond}$'s value is $\top$. From this and *decls*'s definition, the result of $\texttt{SELECT}\ wp(\phi, \bar{t}^{k-1})$ on the database s is $\top$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\bar{t}^{k-1}$ represents a computation. From this, the result of $\texttt{SELECT}\ wp(\psi, \bar{t}^{k-1})$ on the database s is $\top$, and Lemma D.8, it follows that the trigger is enabled in the computation.

If $r = \texttt{ex}$, it follows that $allowed(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the trigger on the database does not throw a security exception. Furthermore, from $r = \texttt{ex}$, it follows that $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ contains the conjunct $x^{\bar{t}}_{k,cond} \wedge \neg(x^{\bar{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{k,\gamma_n})$. From this and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that $x^{\bar{t}}_{k,cond}$'s value is $\top$. From this and *decls*'s definition, the result of $\texttt{SELECT}\ wp(\phi, \bar{t}^{k-1})$ on the database s is $\top$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\bar{t}^{k-1}$ represents a computation. From this, the result of $\texttt{SELECT}\ wp(\psi, \bar{t}^{k-1})$ on the database s is $\top$, and Lemma D.8, it follows that the trigger is enabled in the computation. Furthermore, there is one of $\{x^{\bar{t}}_{1,\gamma_1}, \ldots, x^{\bar{t}}_{1,\gamma_n}\}$ that evaluates to $\bot$. From this and *decls*'s definition, there is a γ_i such that the result of $\texttt{SELECT}\ wp(\gamma_i, \bar{t}^k)$ on the database s is $\bot$. From this and Lemma D.8, executing the trigger t throws an integrity exception.

If $r = \texttt{dis}$, $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ contains the conjunct $\neg x^{\bar{t}}_{k,cond}$. From this and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that $x^{\bar{t}}_{k,cond}$'s value is $\bot$. From this and *decls*'s definition, the result of $\texttt{SELECT}\ wp(\phi, \bar{t}^{k-1})$ on the database s is $\bot$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\bar{t}^{k-1}$ represents a computation. From this, the result of $\texttt{SELECT}\ wp(\psi, \bar{t}^{k-1})$ on the database s is $\bot$, and Lemma D.8, it follows that the trigger is disabled in the computation.

Finally, if $r = \texttt{ok}$, it follows that $allowed(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the trigger on the database does not throw a security exception. Furthermore, from $r = \texttt{ex}$, it follows that $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ contains the conjunct $x^{\bar{t}}_{k,cond} \wedge x^{\bar{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{k,\gamma_n}$. From this and $cond_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that the values of $x^{\bar{t}}_{k,cond}$, $x^{\bar{t}}_{k,\gamma_1}, \ldots, x^{\bar{t}}_{k,\gamma_n}$ are $\top$. From this and *decls*'s definition, the results of $\texttt{SELECT}\ wp(\phi, \bar{t}^{k-1})$, $\texttt{SELECT}\ wp(\gamma_1, \bar{t}^k), \ldots, \texttt{SELECT}\ wp(\gamma_n, \bar{t}^k)$ on the database s are $\top$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\bar{t}^{k-1}$ represents a computation. From this, the results of $\texttt{SELECT}\ wp(\phi, \bar{t}^{k-1})$, $\texttt{SELECT}\ wp(\gamma_1, \bar{t}^k), \ldots, \texttt{SELECT}\ wp(\gamma_n, \bar{t}^k)$ on the database s are $\top$, and Lemma D.8, it follows that (1) the trigger t is enabled and (2) executing the trigger t does not throw integrity exceptions.

Encoding and execution paths. Here, we show that $body_{\langle s,ctx\rangle,m,u,x,q}(\bar{t})$ correctly implements the semantics of the execution path $\bar{t}$. Let $\bar{t}$ be a configuration-consistent execution path. If $\bar{t} = \langle q, \texttt{secEx}\rangle$, then the produced code only assigns $\langle \textbf{SecEx}, \emptyset\rangle$ to x and does not modify the database. If $\bar{t} = \langle q, \texttt{ex}\rangle$, then the produced code only assigns $\langle \textbf{IntEx}, \theta\rangle$ to x, where θ is the set of violated constraints, and does not modify the database. Furthermore, in both cases the produced code creates the corresponding database-level event for the user $db(u)$. Since we use the weakest precondition, θ contains exactly the violated constraints. If $\bar{t}$ does not ends in `ex` or `secEx` and $|\bar{t} = 1|$, the generated code produces the database-level event (for the user $db(u)$ or *public* depending on the command), modifies the database state as described in $\bar{t}$, and sets the correct value for x. If $\bar{t}$ does not ends in `ex` or `secEx` and $|\bar{t} = 1|$, the generated code produces the database-level event (for the user $db(u)$ or *public* depending on the command) that contains also the observations produced by triggers, modifies the database state as described in $\bar{t}$, and sets the correct value for x. If $|\bar{t}| > 1$ and $\bar{t}$ ends in `ex`, then the generated code stores in x the error message $\langle t, \textbf{IntEx}, \theta\rangle$ (which contains the trigger t that has thrown the exception and the set θ of all violated integrity constraints) and produces the associated database-level event. Finally, if $|\bar{t}| > 1$ and $\bar{t}$ ends in `secEx`, then the generated code stores in x the error message $\langle t, \textbf{SecEx}, \emptyset\rangle$ (which contains the trigger t that has thrown the exception) and produces the associated database-level event. □

Lemma D.10 states that the local semantics of the security monitor correctly mimics the operational semantics of WhileSql for all statements that are not queries $x \leftarrow q$.

Lemma D.10. *Let $c \in Com$ be an extended* WhileSql *program such that $safe(c)$, $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, and $u \in UID$ be a user. Furthermore, let Δ be a monitor state. If $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, then $\langle strip(c), m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}{}^{R}_{u} \langle strip(c'), m'', \langle s'', ctx''\rangle\rangle$ and the following conditions hold: (1) for all variables x occurring in $strip(c)$, then $m'(x) = m''(x)$, (2) $s' = s''$, (3) $triggers(ctx') = triggers(ctx'') = \epsilon$, and (4) $\tau' = \tau''$, where $\rightarrow^R_u$ is the reflexive closure of $\rightarrow_u$.*

Proof. Let $c \in Com$ be an extended WhileSql program such that $safe(c)$, $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, $u \in UID$ be a user, and Δ be a monitor state. Assume that $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$. We prove our claim by structural induction on the rules defining $\overset{\tau'}{\rightsquigarrow}_u$.

Base case. There are several cases depending on the rule used in the computation:

- Rule F-Skip. From the rule, it follows that $c = \textbf{skip}$, $\Delta = \Delta'$, $c' = \varepsilon$, $m = m'$, $\langle s, ctx\rangle = \langle s', ctx'\rangle$, and $\tau' = \epsilon$. By applying the E-Skip rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \textbf{skip}$, we obtain that $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $c'' = \varepsilon$, $m'' = m$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-Assign. From the rule, it follows that $c = x := e$, $c' = \varepsilon$, $m' = m[x \mapsto [\![e]\!](m)]$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = \epsilon$. By applying the E-Assign rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = x := e$, we obtain $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $c'' = \varepsilon$, $m'' = m[x \mapsto [\![e]\!](m)]$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-Out. From the rule, it follows that $c = \textbf{out}(u', e)$, $c' = \varepsilon$, $m' = m$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = (u', [\![e]\!](m))$. By applying the E-Out rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \textbf{out}(u', e)$, we obtain $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \langle u', [\![e]\!](m)\rangle$, $c'' = \varepsilon$, $m'' = m$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-IfTrue. From the rule, it follows that $c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$, $[\![e]\!](m) = \top$, $c' = [c_1 \ ; \ \textbf{set pc to } \Delta(\texttt{pc}_u)]$, $m' = m$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = \epsilon$. By applying the E-IfTrue rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$, we obtain $\langle strip(c), m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle strip(c'), m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $m'' = m$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$ (because $[\![e]\!](m) = \top$, $strip(c) = \textbf{if } e \textbf{ then } strip(c_1) \textbf{ else } strip(c_2)$, and $strip(c') = strip(c_1)$). Therefore, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-IfFalse. The proof of this case is similar to that of the F-IfTrue case.
- Rule F-WhileTrue. The proof of this case is similar to that of the F-IfTrue case.
- Rule F-WhileFalse. The proof of this case is similar to that of the F-IfTrue case.
- Rule F-SeqEmpty. From the rule, it follows that $c = \varepsilon \ ; c_1$, $c' = c_1$, $m = m'$, $\langle s, ctx\rangle = \langle s', ctx'\rangle$, and $\tau' = \varepsilon$. By applying the E-SeqEmpty rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \varepsilon \ ; c_1$, we obtain $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $c'' = c_1$, $m'' = m$, and

$\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.

- Rule F-RemoveExpandedCode. The proof of this case is similar to that of the F-SeqEmpty case.
- Rule F-UpdateLabels. From the rule, it follows that $c =$ **set pc to** l, $c' = \varepsilon$, $m' = m$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = \epsilon$. Since the rule modifies only the monitor configuration and $strip(c) = strip(c') = \varepsilon$, it follows that $\langle strip(c), m, \langle s, ctx\rangle\rangle \rightarrow_u^R \langle strip(c'), m'', \langle s'', ctx''\rangle\rangle$, where $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, and $triggers(ctx') = triggers(ctx'') = \epsilon$.

This completes the proof of the base step.

Induction Step. For the induction step, we consider only the F-Seq and F-ExpandedCode rules.

- Rule F-Seq. From the rule, it follows that $c = c_1 \ ; \ c_2$ and $\langle \Delta, c_1, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', ctx''\rangle\rangle$. Furthermore, from $safe(c)$, it follows that we can apply the induction hypothesis. From $\langle \Delta, c_1, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', ctx''\rangle\rangle$ and the induction's hypothesis, it follows that $\langle strip(c_1), m, \langle s, ctx\rangle\rangle \overset{R}{\underset{\tau''}{\longrightarrow}}_u \langle strip(c_1'), m''', \langle s''', ctx'''\rangle\rangle$ such that m''' and m'' agree on all variables occurring in $strip(c)$, $\tau'' = \tau'$, $s'' = s'''$, and $triggers(ctx'') = triggers(ctx''') = \epsilon$. Observe also that $strip(c_1; c_2) = strip(c_1); strip(c_2)$. There are two cases:
  - the first statement executed in c_1 is of the form **set pc to** l. Therefore, $m''' = m'' = m$, $\langle s, ctx\rangle = \langle s'', ctx''\rangle = \langle s''', ctx'''\rangle$, and $\tau' = \tau'' = \epsilon$. From this, it directly follows that $\langle (strip(c_1) \ ; \ strip(c_2)), m, \langle s, ctx\rangle\rangle \overset{R}{\underset{\tau'}{\longrightarrow}}_u \langle strip(c_1') \ ; \ strip(c_2), m''', \langle s''', ctx'''\rangle\rangle$.
  - the first statement executed in c_1 is not of the form **set pc to** l. By applying the E-Seq rule to $\langle \Delta, strip(c_1 \ ; \ c_2), m, \langle s, ctx\rangle\rangle$ (given that (1) $\langle strip(c_1), m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle strip(c_1'), m''', \langle s''', ctx'''\rangle\rangle$, and (2) $strip(c_1 \ ; \ c_2) = strip(c_1) \ ; \ strip(c_2)$) we obtain $\langle (strip(c_1) \ ; \ strip(c_2)), m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle strip(c_1') \ ; \ strip(c_2), m''', \langle s''', ctx'''\rangle\rangle$. Our claim directly follows from (1) m''' and m'' agree on all variables modified in the computation, (2) $\tau'' = \tau'$, (3) $s'' = s'''$, and (4) $triggers(ctx'') = triggers(ctx''') = \epsilon$.
- Rule F-ExpandedCode. From the rule, it follows that $c = [c_1]$ and $\langle \Delta, c_1, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', ctx''\rangle\rangle$. Furthermore, from $safe(c)$, it follows that we can apply the induction hypothesis. From $\langle \Delta, c_1, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', ctx''\rangle\rangle$ and the induction's hypothesis, it follows that $\langle strip(c_1), m, \langle s, ctx\rangle\rangle \overset{R}{\underset{\tau''}{\longrightarrow}}_u \langle strip(c_1'), m''', \langle s''', ctx'''\rangle\rangle$ such that m''' and m'' agree on all variables occurring in $strip(c)$, $\tau'' = \tau'$, $s'' = s'''$, and $triggers(ctx'') = triggers(ctx''') = \epsilon$ (since $strip([c]) = strip(c)$).

This completes the proof of the induction step. □

Finally, Theorem D.5 states that the local semantics of the security monitor correctly implements the local semantics of WhileSql.

Theorem D.5. *Let $c \in Com$ be a* WhileSql *program (without extended commands from Section 7.6), $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, and $u \in UID$ be a user. Furthermore, let Δ be a monitor state. If $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u^* \langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$, then $\langle c, m, \langle s, ctx\rangle\rangle \overset{\tau''}{\longrightarrow}^*_u \langle \varepsilon, m'', \langle s'', ctx''\rangle\rangle$ and the following conditions hold: (1) for all variables x occurring in c, then $m'(x) = m''(x)$, (2) $s' = s''$, (3) $triggers(ctx') = triggers(ctx'') = \epsilon$, and (4) $\tau' = \tau''$.*

Proof. This claim directly follows from Lemma D.9, Lemma D.10. In particular, Lemma D.9 is used to handle statements of the form $x \leftarrow q$, whereas Lemma D.10 is used to handle all other statements. □

D.2.2 Global semantics

Theorem D.6 shows that the monitor's global semantics is transparent for sequential schedulers. We remark, however, that the monitor's global semantics is, in general, not transparent as the monitor modifies the scheduling of commands to avoid timing leaks that may be introduced by the parallel execution of multiple WhileSql programs.

Theorem D.6. *Let $C \in Com^*_{UID}$ be a sequence of* WhileSql *programs (without the extended commands from Section 7.6), $M \in Mem^*_{UID}$ be a sequence of memories, s be a system state, and $\mathcal{S}$ be the sequential scheduler 0^∞. Whenever $\langle \Delta, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}^* \langle \Delta', \epsilon, M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ then $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightarrow}^* \langle \epsilon, M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle$ and the following conditions hold: (1) for all $1 \leq i \leq |M|$, for all variables that occur in $C(i)$, then $M'(i)(x) = M''(i)(x)$, (2) $s' = s''$, (3) $triggers(ctx) = triggers(ctx') = \epsilon$, and (4) $\mathcal{S}' = \mathcal{S}''$.*

$$deps(\langle u', [\![e]\!](m)\rangle, \langle \Delta, \mathbf{out}(u', e), m, s\rangle \overset{\langle u', [\![e]\!](m)\rangle}{\rightsquigarrow_u} \langle \Delta, \varepsilon, m, s\rangle) = vars(e)$$

$$deps(\langle u, v', o', \tau'\rangle, conf \overset{\langle u, v', o', \tau'\rangle}{\rightsquigarrow_u} conf') = vars(v) \cup vars(o) \cup \bigcup_{1 \le i \le |\tau|} vars(\tau(i))$$
$$\text{where } conf = \langle \Delta, \mathbf{dbout}(u, v, o, \tau), m, s\rangle \text{ and } conf' = \langle \Delta, \varepsilon, m, s\rangle$$

$$deps(obs, \langle \Delta, c_1 \,;\, c_2, m, s\rangle \overset{obs}{\rightsquigarrow_u} \langle \Delta', c_1' \,;\, c_2, m', s'\rangle) = deps(obs, \langle \Delta, c_1, m, s\rangle \overset{obs}{\rightsquigarrow_u} \langle \Delta', c_1', m', s'\rangle)$$

$$deps(obs, \langle \Delta, \mathbf{asuser}(u', c), m, s\rangle \overset{obs}{\rightsquigarrow_u} \langle \Delta', c', m', s'\rangle) = deps(obs, \langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow_{u'}} \langle \Delta', c', m', s'\rangle)$$
$$\text{where } query(c) = \top$$

$$deps(obs, \langle \Delta, [c], m, s\rangle \overset{obs}{\rightsquigarrow_u} \langle \Delta', [c'], m', s'\rangle) = deps(obs, \langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow_u} \langle \Delta', c', m', s'\rangle)$$

$$deps(obs, \langle \Delta, C, M, s, n' \cdot \mathcal{S}\rangle \overset{obs}{\rightsquigarrow} \langle \Delta', C', M', s', \mathcal{S}'\rangle) = deps(obs, \langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow_u} \langle \Delta', c', m', s'\rangle)$$
$$\text{where } n = 1 + (n' \bmod |C|), C(n) = \langle u, c\rangle, \text{ and } M(n) = \langle u, m\rangle$$

$$deps(obs, conf \overset{\tau}{\rightarrow} conf') = \emptyset \text{ for any } obs \text{ and } conf \overset{\tau}{\rightarrow} conf' \text{ not matching the above cases}$$

FIGURE D.1: Direct dependencies.

Proof. The claim directly follows from (1) the use of the sequential scheduler, (2) the application of Theorem D.5 to the execution of each program in C, and (3) the fact that the monitor's global semantics does not add new observations to the trace. □

D.3 Monitor's soundness

Here, we prove the main result of Section 7.6, namely that our enforcement mechanism is sound with respect to our security condition for external attackers. In the following, let $\langle D, \Gamma\rangle$ be a system configuration such that the constraints in Γ are well-formed.

D.3.1 Auxiliary notation

In the following, we lift the trace projection operator $\tau\restriction_u$ to sets of users. The projection of a trace τ for a set of users U, written $\tau\restriction_U$, is as follows: $\epsilon\restriction_U = \epsilon$, $(\langle u', e\rangle\cdot\tau')\restriction_U = \langle u', e\rangle\cdot\tau'\restriction_U$ if there exists $u \in U$ such that $u' \preceq_{\mathcal{U}} u$, $(\langle u', q, o, \tau''\rangle\cdot\tau')\restriction_U = \langle u', q, o, \tau''\restriction_U\rangle\cdot\tau'\restriction_U$ if there exists a user $u \in U$ such that (1) $u' \preceq_{\mathcal{U}} u$ or (2) there is a $u'' \in users(\tau'')$ such that $u'' \preceq_{\mathcal{U}} u$, and $(\langle u', e\rangle\cdot\tau')\restriction_U = (\langle u', q, o, \tau''\rangle\cdot\tau')\restriction_U = \tau'\restriction_U$ otherwise, where $users(\tau)$ is the set of all users appearing in the observations in τ.

Let obs be either a program-level or a database-level observation. We denote by $user(obs)$ the user associated with the observation. Namely, $user(\langle u, o\rangle) = u$, $user(\langle public, q, o, \tau\rangle) = ATK$, and $user(\langle u', q, o, \tau\rangle) = u'$ if $u' \neq ATK \wedge \tau\restriction_{ATK} = \epsilon$ and $user(\langle u', q, o, \tau\rangle) = ATK$ otherwise.

Let obs be either a program-level or a database-level observation and $conf \overset{obs}{\rightsquigarrow} conf'$ be a step of the local or global semantics. The direct dependencies of obs given $conf \overset{obs}{\rightsquigarrow} conf'$ are defined in Figure D.1.

Let c be a WHILESQL extended program. The function $first(c)$ returns the first statement to be executed in c. Formally:

$$first(c) = \begin{cases} first(c_1) & \text{if } \exists c_1.\ c = [c_1] \\ first(c_1) & \text{if } \exists c_1, c_2.\ c = c_1 \,;\, c_2 \\ first(c_1) & \text{if } \exists u, c_1.\ c = \mathbf{asuser}(u, c_1) \\ c & \text{otherwise} \end{cases}$$

D.3.2 Equivalence definitions

We now introduce a number of equivalence relations that we use throughout the proofs. We first introduce equivalence relations between monitor state, database states, and memories.

Definition D.2. Let Δ, Δ' be two monitor states, $\langle m, s\rangle, \langle m', s'\rangle$ be two local states, and u be a user.

We say that Δ and Δ' are L-equivalent, where L is a subset of $Vars \cup RC^{pred} \cup \{\mathsf{pc}_u \mid u \in UID\}$, written $\Delta \approx_L \Delta'$, iff for all $x \in L$, $\Delta(x) = \Delta'(x)$.

We say that $s = \langle db, U, S, T, V\rangle$ and $s' = \langle db', U', S', T', V'\rangle$ are configuration equivalent, written $s \equiv^{cfg} s'$, iff $U = U'$, $S = S'$, $T = T'$, and $V = V'$.

We say that $\langle m, s\rangle$ and $\langle m', s'\rangle$ are (V, Q)-equivalent, where $V \subseteq \mathit{Vars}$ and $Q \subseteq RC$, written $\langle m, s\rangle \approx_{V,Q} \langle m', s'\rangle$, iff (1) for all $x \in V$, $m(x) = m'(x)$, and (2) for all $q \in Q$, $[q]^{db} = [q]^{db'}$, where $s = \langle db, U, sec, T, V\rangle$ and $s' = \langle db', U', S', T', V'\rangle$. □

We now formalize equivalence of local configurations.

Definition D.3. Let $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$, $\langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ be two local configurations and $\ell \in \mathcal{L}$ be a label. We say that $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$ and $\langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ are *ℓ-equivalent*, written $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \approx_\ell \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ iff

- for all $x \in \mathit{Vars} \cup \{pc_u \mid u \in \mathit{UID}\}$, $\Delta(x) \sqsubseteq \ell$ iff $\Delta'(x) \sqsubseteq \ell$,
- for all $q \in RC^{pred}$, $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$,
- $\langle m, s\rangle \approx_{V,Q} \langle m', s'\rangle$, where $V = \{x \in \mathit{Vars} \mid \Delta(x) \sqsubseteq \ell\}$ and $Q = \{q \in RC \mid L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \ell\}$, and
- $\Delta \approx_L \Delta'$, where $L = \{x \in \mathit{Vars} \mid \Delta(x) \sqsubseteq \ell\} \cup \{q \in RC^{pred} \mid \Delta(q) \sqsubseteq \ell\} \cup \{\mathtt{pc}_u \mid u \in \mathit{UID}\}$.

Similarly, we say that two global configurations $\langle \Delta, C, M, \langle s, ctx\rangle, \mathcal{S}\rangle$ and $\langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ are ℓ-equivalent, written $\langle \Delta, C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \approx_\ell \langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, iff $|C| = |C'|$, $|M| = |M'|$, and for all $1 \le i \le |C|$, $\langle \Delta, C(i), M(i), \langle s, ctx\rangle\rangle \approx_\ell \langle \Delta', C'(i), M'(i), \langle s', ctx'\rangle\rangle$ □

D.3.3 Results about $\sqcup$

Here we show a simple property of joins in our disclosure lattice, namely that $l_1 \sqcup l_2 \sqsubseteq l_3$ holds iff both $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$ hold. While one of the directions (namely $l_1 \sqcup l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3 \wedge l_2 \sqsubseteq l_3$) holds for disclosure lattices in general, the other one (i.e., $l_1 \sqsubseteq l_3 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqcup l_2 \sqsubseteq l_3$) holds specifically for the determinacy-based lattice. We do not explicitly refer to Proposition D.8 in the rest of the proofs. Observe that from Proposition D.8 it follows that $\bigwedge_i (l_i \sqsubseteq l)$ iff $(\bigsqcup_i l_i) \sqsubseteq l$.

Proposition D.8. *Let D be a database schema, Γ be a set of integrity constraints, $\preceq^{\rightarrow}_{D,\Gamma}$ be the relation such that $Q \preceq^{\rightarrow}_{D,\Gamma} Q'$ iff $D, \Gamma \vdash Q' \twoheadrightarrow Q$. Furthermore, the $\preceq^{\rightarrow}_{D,\Gamma}$-disclosure lattice is $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top\rangle$, where $\sqsubseteq$ is $\preceq^{\rightarrow}_{D,\Gamma}$. For any $l_1, l_2, l_3 \in \mathcal{L}$, the following properties hold:*

- *If $l_1 \sqcup l_2 \sqsubseteq l_3$, then $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$.*
- *If $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$, then $l_1 \sqcup l_2 \sqsubseteq l_3$.*

Proof. Let D be a database schema, Γ be a set of integrity constraints, $\preceq^{\rightarrow}_{D,\Gamma}$ be the relation such that $Q \preceq^{\rightarrow}_{D,\Gamma} Q'$ iff $D, \Gamma \vdash Q' \twoheadrightarrow Q$. Furthermore, the $\preceq^{\rightarrow}_{D,\Gamma}$-disclosure lattice is $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top\rangle$, where $\sqsubseteq$ is $\preceq^{\rightarrow}_{D,\Gamma}$.

First statement. Let l_1, l_2, l_3 be three elements in $\mathcal{L}$ such that $l_1 \sqcup l_2 \sqsubseteq l_3$. From this and $\mathcal{L}$'s definition, it follows that there are three sets of queries $Q_1, Q_2, Q_3 \in 2^{RC}$ such that $l_i = cl(Q_i)$ for $1 \le i \le 3$. From this and $l_1 \sqcup l_2 \sqsubseteq l_3$, it follows that $cl(Q_1) \sqcup cl(Q_2) \sqsubseteq cl(Q_3)$. From this and $\sqcup$'s definition, it follows that $cl(Q_1 \cup Q_2) \sqsubseteq cl(Q_3)$. From the definition of closure, $cl(Q_1) \subseteq cl(Q_1 \cup Q_2)$ and $cl(Q_2) \subseteq cl(Q_1 \cup Q_2)$. From this and the notion of disclosure order, $cl(Q_1) \sqsubseteq cl(Q_1 \cup Q_2)$ and $cl(Q_2) \sqsubseteq cl(Q_1 \cup Q_2)$. From this and $cl(Q_1 \cup Q_2) \sqsubseteq cl(Q_3)$, $cl(Q_1) \sqsubseteq cl(Q_3)$ and $cl(Q_2) \sqsubseteq cl(Q_3)$. Hence, $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$.

Second statement. Let l_1, l_2, l_3 be three elements in $\mathcal{L}$ such that $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$. From this and $\mathcal{L}$'s definition, it follows that there are three sets of queries $Q_1, Q_2, Q_3 \in 2^{RC}$ such that $l_i = cl(Q_i)$ for $1 \le i \le 3$. From $cl(Q_1) \sqsubseteq cl(Q_3)$, $cl(Q_2) \sqsubseteq cl(Q_3)$, and property (2) of disclosure lattices, it follows that $Q_1 \preceq^{\rightarrow}_{D,\Gamma} Q_3$ and $Q_2 \preceq^{\rightarrow}_{D,\Gamma} Q_3$. From this and $\preceq^{\rightarrow}_{D,\Gamma}$'s definition, $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ and $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$. We claim that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1 \cup Q_2$. From this, $Q_1 \cup Q_2 \preceq^{\rightarrow}_{D,\Gamma} Q_3$. From this and property (2) of disclosure lattices, $cl(Q_1 \cup Q_2) \sqsubseteq cl(Q_3)$. From this and $cl(Q_1) \sqcup cl(Q_2) = cl(Q_1 \cup Q_2)$, $cl(Q_1) \sqcup cl(Q_2) \sqsubseteq cl(Q_3)$.

We now prove our claim that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ and $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$ imply $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1 \cup Q_2$. Let Q_1, Q_2 and Q_3 be three sets of queries in 2^{RC} such that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ and $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$. Assume, for contradiction's sake, that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1 \cup Q_2$ does not hold. From this, it follows that there are two database states db and db' and a query $q' \in Q_1 \cup Q_2$ such that $[q]^{db} = [q]^{db'}$ for all $q \in Q_3$ and $[q']^{db} \neq [q']^{db'}$. If $q' \in Q_1$, then there are two database states db and db' and a query $q' \in Q_1$ such that $[q]^{db} = [q]^{db'}$ for all $q \in Q_3$ and $[q']^{db} \neq [q']^{db'}$. Therefore, $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ does not hold, leading to a contradiction. Similarly, if $q' \in Q_2$, then there are two database states db and db' and a query $q' \in Q_2$ such that $[q]^{db} = [q]^{db'}$ for all $q \in Q_3$ and $[q']^{db} \neq [q']^{db'}$. Therefore, $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$ does not hold, leading to a contradiction. Since both cases lead to a contradiction, this completes the proof of our claim. □

D.3.4 Results about $L_{\mathcal{Q}}$

Here we state some simple facts about $L_{\mathcal{Q}}$.

Proposition D.9. *Let Δ and Δ' be two monitor states. If $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$, then $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \ell$ iff $L_{\mathcal{Q}}(\Delta', q) \sqsubseteq \ell$ for all $q \in RC$.*

Proof. Let Δ and Δ' be two monitor states such that $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$.
($\Rightarrow$). Assume that $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \ell$ holds. From this, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this and $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this, $L_{\mathcal{Q}}(\Delta', q) \sqsubseteq \ell$.
($\Leftarrow$). Assume that $L_{\mathcal{Q}}(\Delta', q) \sqsubseteq \ell$ holds. From this, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this and $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this, $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \ell$. □

Proposition D.10. *Given a monitor state Δ and a predicate query $q \in RC^{pred}$, $\Delta(q) = L_{\mathcal{Q}}(\Delta, q)$.*

Proof. Let Δ be a monitor state and $q \in RC^{pred}$ be a predicate query. From the definition of $L_{\mathcal{Q}}(\Delta, q)$, it follows that $L_{\mathcal{Q}}(\Delta, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta(q')$. Since Γ is a set of well-formed integrity constraints and $q \in RC^{pred}$, $supp_{D,\Gamma}(q) = \{\{q\}\}$. From this, $L_{\mathcal{Q}}(\Delta, q) = \Delta(q)$. □

D.3.5 Results about relaxed NSU checks

We now prove some simple results about relaxed NSU checks.

Proposition D.11. *Let sec_0 be the initial policy, ℓ be a label such that $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Δ be a monitor state. If $\mathbf{nsu}(x, \mathtt{pc}_u)$ is satisfied for Δ and $\Delta(x) \sqsubseteq \ell$, then $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$.*

Proof. Let sec_0 be the initial policy, ℓ be a label such that $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Δ be a monitor state. Furthermore, assume that $\mathbf{nsu}(x, \mathtt{pc}_u)$ is satisfied for Δ and $\Delta(x) \sqsubseteq \ell$. From this, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, ATK)) \vee \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$. There are two cases:

1. $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, ATK))$ holds. From this and $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$.
2. $\Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$ holds. From this and $\Delta(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$.

This completes the proof. □

D.3.6 Lemmas about the local semantics

Here we present some auxiliary results about the local semantics of our enforcement mechanism.

Lemma D.11 states that whenever the security monitor produces an output, the labels associated with pc and with the event's dependencies are less than (or equal to) the label associated with the user that can observe the event. We remark that this lemma directly implies two facts: (1) observable events for ATK occur only in low contexts, i.e., in contexts such that $\Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, ATK)$, and (2) all direct flows are authorized, namely ATK observes only events that directly depend on information at a lower (or equal) level in the security lattice.

Lemma D.11. *Whenever $r = \langle \Delta, c, m, s \rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s' \rangle$ and $obs \neq \epsilon$, we have $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$.*

Proof. Let $\langle \Delta, c, m, s \rangle$ and $\langle \Delta', c', m', s' \rangle$ be local configurations such that $\langle \Delta, c, m, s \rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s' \rangle$ and $obs \neq \epsilon$. In the following, we denote $\langle \Delta, c, m, s \rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s' \rangle$ as r. We now prove, by structural induction on $\overset{\mathcal{I}}{\rightarrow}_u$, that $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$. In the following we consider only on those rules that produce observations. Proving the claim for rules not producing outputs is trivial.

Base Case. There are two cases depending on the rule used to produce the event.

- Rule F-Out. From the rule, it directly follows that (1) $obs = \langle u', [\![e]\!](m) \rangle$, and (2) $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')$. From *user*'s definition, it follows that $user(\langle u', [\![e]\!](m) \rangle) = u'$. From this and $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')$, it follows that $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', [\![e]\!](m) \rangle))$. From this, $deps(\langle u', [\![e]\!](m) \rangle, r) = free(e)$, and $\Delta(free(e)) = \Delta(e)$, it follows that $\Delta(deps(\langle u', [\![e]\!](m) \rangle, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', [\![e]\!](m) \rangle))$. Hence, $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$.

- Rule F-DbOut. From the rule, it follows that (1) $obs = \langle u', v', o', \tau' \rangle$, where $v' = v[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_x]\!](m)]$, $o' = o[o_1 \mapsto [\![o_1]\!](m), \ldots, o_n \mapsto [\![o_y]\!](m)]$, and $\tau'(i) = \tau(i)[k_1 \mapsto [\![k_1]\!](m), \ldots, k_n \mapsto [\![k_{n_i}]\!](m)]$, and (2) $\Delta(\mathtt{pc}_u) \sqcup \ell_{obs} \sqsubseteq L_{\mathcal{U}}(s, u'')$, where $\ell_{obs} = \bigsqcup_{x \in vars(v)} \Delta(x) \sqcup \bigsqcup_{x \in vars(o)} \Delta(x) \sqcup \bigsqcup_{1 \leq i \leq |\tau|} \bigsqcup_{x \in vars(\tau(i))} \Delta(x)$ and $u'' = u'$ if $u' \neq ATK$ and $\tau\restriction_{ATK} = \epsilon$ and $u'' = ATK$ otherwise. From *user*'s definition, there are two cases:
  1. $user(\langle u', v', o', \tau' \rangle) = u'$ and $u' \neq ATK$. This happens iff $u' \neq ATK$ and $\tau'\restriction_{ATK} = \epsilon$. From this, $L_{\mathcal{U}}(s, user(\langle u', v', o', \tau' \rangle)) = \top$ and the claim trivially holds.
  2. $user(\langle u', v', o', \tau' \rangle) = ATK$. From this, it follows that $u' = ATK$, $u' = public$, or $\tau'\restriction_{ATK} \neq \epsilon$. From this, $u'' = user(\langle u', v', o', \tau' \rangle) = ATK$. From this and $\Delta(\mathtt{pc}_u) \sqcup \ell_{obs} \sqsubseteq L_{\mathcal{U}}(s, u'')$, it follows that $\Delta(\mathtt{pc}_u) \sqcup \ell_{obs} \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', v', o', \tau' \rangle))$. From this and $deps(\langle u', v', o', \tau' \rangle, r) = vars(v) \cup vars(o) \cup \bigcup_{1 \leq i \leq |\tau|} vars(\tau(i))$, it follows that $\Delta(deps(\langle u', v', o', \tau' \rangle, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', v', o', \tau' \rangle))$. Hence, $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$.

This completes the proof of the base case.

Induction Step. The proof for the induction step directly follows from the induction hypothesis (since the rules do not further introduce events). □

Lemma D.12 states that, given a label ℓ, whenever $\mathtt{pc}_u$ becomes high with respect to ℓ, this is caused by a branching statement, a loop statement, or a set-label statement.

Lemma D.12. *Let $\ell \in \mathcal{L}$ be a label. Whenever $\langle \Delta, c, m, \langle s, ctx \rangle \rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx' \rangle \rangle$, if $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$, then one of the following conditions hold:*

- *$first(c) =$ **if** e **then** c_1 **else** c_2 and $\Delta(e) \not\sqsubseteq \ell$,*
- *$first(c) =$ **while** e **do** c_1 and $\Delta(e) \not\sqsubseteq \ell$, or*
- *$first(c) =$ **set pc to** ℓ_1 and $\ell_1 \not\sqsubseteq \ell$.*

Proof. Let c be a WhileSql program, $\ell \in \mathcal{L}$ be a label, and $\langle \Delta, c, m, \langle s, ctx \rangle \rangle$ and $\langle \Delta', c', m', \langle s', ctx' \rangle \rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx \rangle \rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx' \rangle \rangle$, and (2) $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$. We now prove, by structural induction on $\rightsquigarrow_u$, that our claim holds. In the following, we focus only on the rules that directly modify $\Delta(\mathtt{pc}_u)$. The proof for the other cases is trivial.

Base Case. There are several cases depending on the rule used to derive $\langle \Delta, c, m, \langle s, ctx \rangle \rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx' \rangle \rangle$.

- Rule F-UpdateLabels. Then, $first(c) =$ **set pc to** ℓ_1. From the rule, it follows that $\Delta'(\mathtt{pc}_u) = \ell_1$. From this and $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$, it follows that $\ell_1 \not\sqsubseteq \ell$.
- Rule F-IfTrue. Then, $first(c) =$ **if** e **then** c_1 **else** c_2. Furthermore, from $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$, $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$, and $\Delta'(\mathtt{pc}_u) = \Delta(e) \sqcup \Delta(\mathtt{pc}_u)$, it follows that $\Delta(e) \not\sqsubseteq \ell$.
- Rule F-IfFalse. The proof of this case is similar to that of F-IfTrue.
- Rule F-WhileTrue. The proof of this case is similar to that of F-IfTrue.
- Rule F-WhileFalse. The proof of this case is similar to that of F-IfTrue.

Induction Step. The proof of the induction step directly follows from the induction hypothesis. □

Lemmas D.13 states that, given a label ℓ, whenever we are in a high context (i.e., $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$), there are no changes to the values associated with all variables and queries whose labels are lower than (or equal to) ℓ.

Lemma D.13. *Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, ATK)) \sqsubseteq \ell$. Whenever $\langle \Delta, c, m, \langle s, ctx \rangle \rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx' \rangle \rangle$, if $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$, then $\langle m, s \rangle \approx_{V,Q} \langle m', s' \rangle$, where $V = \{x \in Vars \mid \Delta(x) \sqsubseteq \ell\}$ and $Q = \{q \in RC \mid L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \ell\}$.*

Proof. Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, ATK)) \sqsubseteq \ell$. Furthermore, let $u \in UID$ be a user and $\langle \Delta, c, m, \langle s, ctx \rangle \rangle, \langle \Delta', c', m', \langle s', ctx' \rangle \rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx \rangle \rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx' \rangle \rangle$, and (2) $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$. Finally, let $V = \{x \in Vars \mid \Delta(x) \sqsubseteq \ell\}$ and $Q = \{q \in RC \mid L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \ell\}$. We now show, by structural induction on the rules defining $\rightsquigarrow_u$, that $\langle m, s \rangle \approx_{V,Q} \langle m', s' \rangle$. In the following, we consider only those rules that modify the memory m or the database state s. For the other rules, the claim holds trivially (since $\langle m, s \rangle = \langle m', s' \rangle$).

Base Case. There are several cases depending on the applied rule:

1. Rule F-Assign. Assume, for contradiction's sake, that $\langle m, s \rangle \not\approx_{V,Q} \langle m', s' \rangle$. From the rule, it follows that $s = s'$ and $m' = m[x \mapsto [\![e]\!](m)]$. From this, $\langle m, s \rangle \not\approx_{V,Q} \langle m', s' \rangle$, and $\approx_{V,Q}$'s definition, it follows that (1) $x \in V$ and, therefore, $\Delta(x) \sqsubseteq \ell$, and (2) $m(x) \neq m'(x)$. From the rule, it follows that $\mathbf{nsu}(x, \mathtt{pc}_u)$ holds. From this, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, $\Delta(x) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.

2. Rule F-SELECT. Assume, for contradiction's sake, that $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$. From the rule, it follows that $s = s'$ and $m' = m[x \mapsto r]$, where $(\langle s, ctx'\rangle, r, \epsilon, \epsilon) = [\![q]\!](s, u)$. From this, $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$, and $\approx_{V,Q}$'s definition, it follows that (1) $x \in V$ and, therefore, $\Delta(x) \sqsubseteq \ell$, and (2) $m(x) \neq m'(x)$. From the rule, it follows that $\mathbf{nsu}(x, \mathtt{pc}_u)$. From this, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, $\Delta(x) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
3. Rule F-UPDATEDATABASEOK. Without loss of generality, we assume that $q = T' \oplus \overline{e}$ and that $\overline{v}$ is the tuple inserted in the database (after evaluating all the expressions). Assume, for contradiction's sake, that $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$. From the rule, it follows that $[\![q]\!](s, u) = (\langle s', ctx'\rangle, r, \epsilon, \epsilon)$ and $m' = m[x \mapsto r]$. From this, $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$, and $\approx_{V,Q}$'s definition, it follows that there are two cases:
   (a) $x \in V$ and $m(x) \neq m'(x)$. From $x \in V$ and V's definition, it follows that $\Delta(x) \sqsubseteq \ell$. From the rule, it follows that $\mathbf{nsu}(x, \mathtt{pc}_u)$. From this, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, $\Delta(x) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   (b) There is a query $q' \in Q$ such that $[q']^s \neq [q']^{s'}$. From $q' \in Q$, it follows that $L_Q(\Delta, q') \sqsubseteq \ell$. From this, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q')} \bigsqcup_{q'' \in Q} \Delta(q'') \sqsubseteq \ell$. We claim that that for all $Q \in supp_{D,\Gamma}(q')$, $T'(\overline{v}) \in Q$. From this and $\bigsqcup_{Q \in supp_{D,\Gamma}(q')} \bigsqcup_{q'' \in Q} \Delta(q'') \sqsubseteq \ell$, it follows that $\Delta(T'(\overline{v})) \sqsubseteq \ell$. From the rule, it follows that $\mathbf{nsu}(T'(\overline{v}), \mathtt{pc}_u)$. From this, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, $\Delta(T'(\overline{v})) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   We now prove our claim that for all $Q \in supp_{D,\Gamma}(q')$, $T'(\overline{v}) \in Q$. Assume that there exists a $Q \in supp_{D,\Gamma}(q')$ such that $T'(\overline{v}) \notin Q$. From $supp_{D,\Gamma}$'s definition, it follows that the predicate queries in Q determine q'. From the database semantics, the result of all queries in Q is the same in s and s' (since we modify only $T'(\overline{v})$). From this and Q determines q', it follows that the result of q' is the same in s and s'. This, however, contradicts $[q']^s \neq [q']^{s'}$.
4. Rule F-UPDATECONFIGURATIONOK. The proof of this case is similar to the F-SELECT case.

Induction Step. The proof for the induction step directly follows from the induction hypothesis (since the rules do not further modify the memory and the database). □

Lemma D.14 states that, given a label ℓ, whenever we are in a high context (i.e., $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$), then (1) there are no changes to the labels associated with variables and queries whose labels are initially below ℓ, and (2) the label of a variable (or query) is initially below ℓ iff it is below ℓ also at the end of the computation.

Lemma D.14. *Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, ATK)) \sqsubseteq \ell$. Whenever $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, if $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$, then the following conditions hold:*

- *for all $x \in Vars$, $\Delta(x) \sqsubseteq \ell$ iff $\Delta'(x) \sqsubseteq \ell$,*
- *for all $q \in RC^{pred}$, $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$,*
- *$\Delta \approx_L \Delta'$, where $L = \{x \in Vars \mid \Delta(x) \sqsubseteq \ell\} \cup \{q \in RC^{pred} \mid \Delta(q) \sqsubseteq \ell\}$.*

Proof. Let sec_0 be the policy used to initialize the monitor, $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, $u \in UID$ be a user, and $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$, $\langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, and (2) $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$. We prove our claim by structural induction on the rules defining $\rightsquigarrow_u$. In the following, we consider only those rules that modify the monitor state Δ for the identifiers in $Vars \cup RC^{pred}$. For the other rules, the claim holds trivially.

Base Case. There are several cases depending on the applied rule:

1. Rule F-ASSIGN. From the rule, it follows that (1) $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \Delta(e)]$, and (2) $\mathbf{nsu}(x, \mathtt{pc}_u)$. Assume, for contradiction's sake, that our claim does not hold. From $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \Delta(e)]$, there are three cases:
   - $\Delta(x) \sqsubseteq \ell$ and $\Delta'(x) \not\sqsubseteq \ell$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \not\sqsubseteq \ell$ and $\Delta'(x) \sqsubseteq \ell$. From $\Delta'(x) = \Delta(\mathtt{pc}_u) \sqcup \Delta(e)$ and $\Delta'(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqcup \Delta(e) \sqsubseteq \ell$. From this, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \sqsubseteq \ell$, $\Delta'(x) \sqsubseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.

2. Rule F-SELECT. From the rule, it follows that $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi]$, where $\ell_\varphi = L_{\mathcal{Q}}(\Delta, \texttt{SELECT}\ \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta(v)$, and $\mathbf{nsu}(x, \mathtt{pc}_u)$. Assume, for contradiction's sake, that our claim does not hold. From $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi]$, there are three cases:
   - $\Delta(x) \sqsubseteq \ell$ and $\Delta'(x) \not\sqsubseteq \ell$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \not\sqsubseteq \ell$ and $\Delta'(x) \sqsubseteq \ell$. From $\Delta'(x) = \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi$ and $\Delta'(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqcup \ell_\varphi \sqsubseteq \ell$. From this, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \sqsubseteq \ell$, $\Delta'(x) \sqsubseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
3. Rule F-UPDATEDATABASEOK. From the rule, it follows that (1) $\Delta' = \Delta[T(\overline{v}) \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e, x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e]$, where $\ell_e = \bigsqcup_{1 \leq i \leq |T|} \Delta(e_i)$, (2) $\mathbf{nsu}(T(\overline{v}), \mathtt{pc}_u)$, and (3) $\mathbf{nsu}(x, \mathtt{pc}_u)$. Assume, for contradiction's sake, that our claim does not hold. From $\Delta' = \Delta[T(\overline{v}) \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e, x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e]$, there are six cases:
   - $\Delta(x) \sqsubseteq \ell$ and $\Delta'(x) \not\sqsubseteq \ell$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \not\sqsubseteq \ell$ and $\Delta'(x) \sqsubseteq \ell$. From $\Delta'(x) = \Delta(\mathtt{pc}_u) \sqcup \ell_e$ and $\Delta'(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \sqsubseteq \ell$, $\Delta'(x) \sqsubseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(T(\overline{v})) \sqsubseteq \ell$ and $\Delta'(T(\overline{v})) \not\sqsubseteq \ell$. From the rule, $\mathbf{nsu}(T(\overline{v}), \mathtt{pc}_u)$. From this, $\Delta(T(\overline{v})) \sqsubseteq \ell$, $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, and Proposition D.11, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(T(\overline{v})) \not\sqsubseteq \ell$ and $\Delta'(T(\overline{v})) \sqsubseteq \ell$. From $\Delta'(T(\overline{v})) = \Delta(\mathtt{pc}_u) \sqcup \ell_e$ and $\Delta'(T(\overline{v})) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(T(\overline{v})) \sqsubseteq \ell$, $\Delta'(T(\overline{v})) \sqsubseteq \ell$, and $\Delta(T(\overline{v})) \neq \Delta'(T(\overline{v}))$. From $\Delta'(T(\overline{v})) \sqsubseteq \ell$ and $\Delta'(T(\overline{v})) = \Delta(\mathtt{pc}_u) \sqcup \ell_e$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
4. Rule F-UPDATECONFIGURATION-OK. The proof of this case is similar to that of F-SELECT.

Induction Step. The proof for the induction step directly follows from the induction hypothesis (since the rules do not further modify the memory and the database). □

Lemma D.15 states that whenever $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$ and $cl(auth(sec_0, ATK)) \sqsubseteq \ell$ then there are no changes to the database configuration.

Lemma D.15. *Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label. Whenever $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, if $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$ and $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, then $s \equiv^{cfg} s'$.*

Proof. Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label. Furthermore, let $u \in UID$ be a user and $\langle \Delta, c, m, \langle s, ctx\rangle\rangle, \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, (2) $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$, and (3) $cl(auth(sec_0, ATK)) \sqsubseteq \ell$. We now show, by structural induction on the rules defining $\rightsquigarrow_u$, that $s \equiv^{cfg} s'$. In the following, we consider only those rules that modify the database configuration. For the other rules, the claim holds trivially (since the configuration is the same in s and s').

Base Case. There only interesting case is the rule F-UPDATECONFIGURATION-OK. From the rule, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, ATK))$. From this and $cl(auth(sec_0, ATK)) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.

Induction Step. The proof of the induction step follows from the induction hypothesis. □

Lemma D.16 states that, under appropriate conditions, executing the same command on two ℓ-equivalent states produces outputs that are indistinguishable for the attacker ATK.

Lemma D.16. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $L_{\mathcal{U}}(s_1, ATK) \sqsubseteq \ell$ *and* $L_{\mathcal{U}}(s_2, ATK) \sqsubseteq \ell$,
4. $c_1 = c_2$,

5. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
6. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,

then $\tau_1{\restriction}_{ATK} = \tau_2{\restriction}_{ATK}$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold: (1) $s_1 \equiv^{cfg} s_2$, (2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, (3) $L_{\mathcal{U}}(s_1, ATK) \sqsubseteq \ell$ and $L_{\mathcal{U}}(s_2, ATK) \sqsubseteq \ell$, (4) $c_1 = c_2$, (5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, (6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, We prove our claim by induction on the rules defining $\rightsquigarrow_u$. Without loss of generality, we focus only on the rules producing observations. The claim trivially holds for all rules that do not produce observations.

Base Case. There are a number of cases depending on the rule applied to derive $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$.

- Rule F-Out. From the rule, it follows that $c_1 = \mathbf{out}(u', e)$. From this and (4), it follows that $c_2 = \mathbf{out}(u', e)$. In the following, we assume that $u' = ATK$. If this is not the case the proof is trivial. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \mathbf{out}(ATK, e)$, it follows that $\Delta_1 = \Delta_1'$, $m_1 = m_1'$, $\tau_1 = \langle u', [\![e]\!](m_1)\rangle$, $c_1' = \varepsilon$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, $c_2 = \mathbf{out}(u', e)$, and the F-Out rule, it follows that $\Delta_2 = \Delta_2'$, $m_2 = m_2'$, $\tau_2 = \langle u', [\![e]\!](m_2)\rangle$, $c_2' = \varepsilon$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. From the rule, it also follows that $\Delta_1(e) \sqcup \Delta_1(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_1, ATK)$ and $\Delta_2(e) \sqcup \Delta_2(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_2, ATK)$. From this, it follows that $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq L_{\mathcal{U}}(s_1, ATK)$ and $\bigwedge_{y \in free(e)} \Delta_2(y) \sqsubseteq L_{\mathcal{U}}(s_2, ATK)$. From this, (2), and (3), it follows that $\bigwedge_{y \in free(e)} m_1(y) = m_2(y)$. From this, it follows that $[\![e]\!](m_1) = [\![e]\!](m_2)$. From this, $\tau_1 = \langle u', [\![e]\!](m_1)\rangle$, and $\tau_2 = \langle u', [\![e]\!](m_2)\rangle$, it follows that $\tau_1 = \tau_2$. Therefore, $\tau_1{\restriction}_{ATK} = \tau_2{\restriction}_{ATK}$.
- Rule F-DbOut. From the rule, it follows that $c_1 = \mathbf{dbout}(u', v, o, \tau)$. From this and (4), it follows that $c_2 = \mathbf{dbout}(u', v, o, \tau)$. In the following, we assume that $u = ATK$, $u = public$, or $\tau{\restriction}_{ATK} \neq \epsilon$. If this is not the case the proof is trivial (since the event cannot be observed by the attacker). From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \mathbf{dbout}(u', v, o, \tau)$, it follows that $\Delta_1 = \Delta_1'$, $m_1 = m_1'$, $c_1' = \varepsilon$, $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$, and $\tau_1 = \langle u', v_1', o_1', \tau_1'\rangle$, where $vars(v) = \{v_1, \ldots, v_x\}$, $vars(o) = \{o_1, \ldots, o_y\}$, $v_1' = v[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_x]\!](m_1)]$, $o_1' = o[o_1 \mapsto [\![o_1]\!](m_1), \ldots, o_n \mapsto [\![o_y]\!](m_1)]$, and for all $1 \leq i \leq |\tau|$, $vars(\tau(i)) = \{k_1, \ldots, k_{n_i}\}$ and $\tau_1'(i) = \tau(i)[k_1 \mapsto [\![k_1]\!](m_1), \ldots, k_n \mapsto [\![k_{n_i}]\!](m_1)]$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, $c_2 = \mathbf{dbout}(u', v, o, \tau)$, and the F-DbOut rule, it follows that $\Delta_2 = \Delta_2'$, $m_2 = m_2'$, $c_2' = \varepsilon$, $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$, and $\tau_2 = \langle u', v_2', o_2', \tau_2'\rangle$, where $vars(v) = \{v_1, \ldots, v_x\}$, $vars(o) = \{o_1, \ldots, o_y\}$, $v_2' = v[v_1 \mapsto [\![v_1]\!](m_2), \ldots, v_n \mapsto [\![v_x]\!](m_2)]$, $o_2' = o[o_1 \mapsto [\![o_1]\!](m_2), \ldots, o_n \mapsto [\![o_y]\!](m_2)]$, and for all $1 \leq i \leq |\tau|$, $vars(\tau(i)) = \{k_1, \ldots, k_{n_i}\}$ and $\tau_2'(i) = \tau(i)[k_1 \mapsto [\![k_1]\!](m_2), \ldots, k_n \mapsto [\![k_{n_i}]\!](m_2)]$. From the rule, it also follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{x \in vars(v)} \Delta_1(x) \sqcup \bigsqcup_{x \in vars(o)} \Delta_1(x) \sqcup \bigsqcup_{1 \leq i \leq |\tau|} \bigsqcup_{x \in vars(\tau(i))} \Delta_1(x) \sqsubseteq L_{\mathcal{U}}(s_1, u'')$ and $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{x \in vars(v)} \Delta_2(x) \sqcup \bigsqcup_{x \in vars(o)} \Delta_2(x) \sqcup \bigsqcup_{1 \leq i \leq |\tau|} \bigsqcup_{x \in vars(\tau(i))} \Delta_2(x) \sqsubseteq L_{\mathcal{U}}(s_2, u'')$, where $u'' = u$ if $u' \neq ATK \wedge u' \neq public \wedge \tau{\restriction}_{ATK} = \epsilon$ and $u' = ATK$ otherwise. From this and $u = ATK$, $u = public$, or $\tau{\restriction}_{ATK} \neq \epsilon$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{x \in vars(v)} \Delta_1(x) \sqcup \bigsqcup_{x \in vars(o)} \Delta_1(x) \sqcup \bigsqcup_{1 \leq i \leq |\tau|} \bigsqcup_{x \in vars(\tau(i))} \Delta_1(x) \sqsubseteq L_{\mathcal{U}}(s_1, ATK)$ and $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{x \in vars(v)} \Delta_2(x) \sqcup \bigsqcup_{x \in vars(o)} \Delta_2(x) \sqcup \bigsqcup_{1 \leq i \leq |\tau|} \bigsqcup_{x \in vars(\tau(i))} \Delta_2(x) \sqsubseteq L_{\mathcal{U}}(s_2, ATK)$. From this and (3), it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{x \in vars(v)} \Delta_1(x) \sqcup \bigsqcup_{x \in vars(o)} \Delta_1(x) \sqcup \bigsqcup_{1 \leq i \leq |\tau|} \bigsqcup_{x \in vars(\tau(i))} \Delta_1(x) \sqsubseteq \ell$ and that $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{x \in vars(v)} \Delta_2(x) \sqcup \bigsqcup_{x \in vars(o)} \Delta_2(x) \sqcup \bigsqcup_{1 \leq i \leq |\tau|} \bigsqcup_{x \in vars(\tau(i))} \Delta_2(x) \sqsubseteq \ell$. From this, it follows $\bigwedge_{x \in vars(v)} \Delta_1(x) \sqsubseteq \ell$, $\bigwedge_{x \in vars(o)} \Delta_1(x) \sqsubseteq \ell$, and $\bigwedge_{1 \leq i \leq |\tau|} \bigwedge_{x \in vars(\tau(i))} \Delta_1(x) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{x \in vars(v)} m_1(x) = m_2(x)$, $\bigwedge_{x \in vars(o)} m_1(x) = m_2(x)$, and $\bigwedge_{1 \leq i \leq |\tau|} \bigwedge_{x \in vars(\tau(i))} m_1(x) = m_2(x)$. From this, it follows that $v_1' = v_2'$, $o_1' = o_2'$, and $\tau_1' = \tau_2'$. From this, $\tau_1 = \tau_2$. Therefore, $\tau_1{\restriction}_{ATK} = \tau_2{\restriction}_{ATK}$.

Induction Step. The proof of the induction step directly follows from the induction hypothesis. □

Lemma D.17 states that, under appropriate conditions, executing the same command on two ℓ-equivalent states modifies the database configuration in the same way.

Lemma D.17. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $c_1 = c_2$,

4. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
5. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,
6. $cl(auth(sec_0, ATK)) \sqsubseteq \ell$,
then $s_1' \equiv^{cfg} s_2'$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold: 1. $s_1 \equiv^{cfg} s_2$, 2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, 3. $c_1 = c_2$, 4. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, 5. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, 6. $cl(auth(sec_0, ATK)) \sqsubseteq \ell$. We prove our claim by induction on the rules defining $\rightsquigarrow_u$. In the following, we focus only on rules that modify the database configuration. For rules that do not modify the database configuration, the claim directly follows from $s_1 \equiv^{cfg} s_2$.

Base Case. The only interesting case is the rule F-UpdateConfigurationOk. From the rule, it follows that $c_1 = \|x \leftarrow q\|$, where q is a configuration command. From this and (3), it follows that $c_2 = \|x \leftarrow q\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v)]$, $m_1' = m_1[x \mapsto r_1]$, $vars(q) = \{v_1, \ldots, v_n\}$, $q_1' = q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m)]$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, and $[\![q_1']\!](\langle s_1, ctx_1\rangle) = (\langle s_1', ctx_1'\rangle, r_1, \epsilon, \epsilon)$. Similarly, from $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = \|x \leftarrow q\|$, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v)]$, $m_2' = m_2[x \mapsto r_2]$, $vars(q) = \{v_1, \ldots, v_n\}$, $q_2' = q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m)]$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, and $[\![q_2']\!](\langle s_2, ctx_2\rangle) = (\langle s_2', ctx_2'\rangle, r_2, \epsilon, \epsilon)$. From the rule, it follows that $\bigsqcup_{v \in vars(q)} \Delta_1(v) \sqsubseteq cl(auth(sec_0, ATK))$. From this, it follows that $\bigwedge_{v \in vars(q)} \Delta_1(v) \sqsubseteq cl(auth(sec_0, ATK))$. From this and (6), it follows that $\bigwedge_{v \in vars(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in vars(q)} m_1(v) = m_2(v)$. From this, $q_1' = q_2'$. From this, (1), $[\![q_1']\!](\langle s_1, ctx_1\rangle) = (\langle s_1', ctx_1'\rangle, r_1, \epsilon, \epsilon)$, and $[\![q_2']\!](\langle s_2, ctx_2\rangle) = (\langle s_2', ctx_2'\rangle, r_2, \epsilon, \epsilon)$, it directly follows that $s_1' \equiv^{cfg} s_2'$ (since the initial configuration is the same and the database semantics is deterministic).

Induction Step. The proof of the induction step directly follows from the induction hypothesis. □

Lemma D.18 states that, given a label ℓ, whenever we are in a low context (i.e., $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$), then executing the same command on two ℓ-equivalent states produces to ℓ-equivalent states.

Lemma D.18. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $c_1 = c_2$,
5. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
6. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,
then $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold:

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $c_1 = c_2$,
5. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
6. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$.

We prove our claim by induction on the rules defining $\rightsquigarrow_u$. In the following, we focus only on those rules that modify the monitor state, the database, or the memory. For the other rules, the claim directly follows from (2).

Base Case. There are a number of cases depending on the rule applied to derive $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$.

- Rule F-UpdateLabels. From the rule, it follows that $c_1 = \textbf{set pc to } l$. From this and (4), it follows that $c_2 = \textbf{set pc to } l$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \textbf{set pc to } l$, it follows that $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto l]$, $m_1 = m_1'$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. Similarly, from $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and

$c_2 = \textbf{set pc to } l$, it follows that $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto l]$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$. From this, $m_1 = m_1'$, $m_2 = m_2'$, $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$, $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$, and (2), there are two cases:

- $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$ (or vice versa). From $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto l]$ and $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto l]$, it follows that $\Delta_1'(\mathtt{pc}_u) = \Delta_2'(\mathtt{pc}_u)$. From this and $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$, it follows that $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
- $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$, $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$, and $\Delta_1'(\mathtt{pc}_u) \neq \Delta_2'(\mathtt{pc}_u)$. From $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto l]$ and $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto l]$, it follows that $\Delta_1'(\mathtt{pc}_u) = \Delta_2'(\mathtt{pc}_u)$, leading to a contradiction.

- Rule F-Assign. From the rule, it follows that $c_1 = x := e$. From this and (4), it follows that $c_2 = x := e$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = x := e$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, $m_1' = m_1[x \mapsto [\![e]\!](m_1)]$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. Similarly, from $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = x := e$, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2' = m_2[x \mapsto [\![e]\!](m_2)]$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$. From this, $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$, $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$, and (2), there are three cases:
  - $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_2'(x) \not\sqsubseteq \ell$ (or vice versa). From the rule, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$ and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$. From this and $\Delta_1'(x) \sqsubseteq \ell$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) \sqsubseteq \ell$. From this, $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$. From $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{y \in free(e)} \Delta_1(y) = \Delta_2(y)$. From $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$ and $\bigwedge_{y \in free(e)} \Delta_1(y) = \Delta_2(y)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) = \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)$. From this, $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, it follows that $\Delta_2'(x) = \Delta_1'(x)$. From this and $\Delta_1'(x) \sqsubseteq \ell$, it follows that $\Delta_2'(x) \sqsubseteq \ell$, leading to a contradiction.
  - $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_2'(x) \sqsubseteq \ell$, and $\Delta_1'(x) \neq \Delta_2'(x)$. We have already shown above that from $\Delta_1'(x) \sqsubseteq \ell$ and (2), it follows $\Delta_2'(x) = \Delta_1'(x)$, which contradicts $\Delta_1'(x) \neq \Delta_2'(x)$.
  - $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_2'(x) \sqsubseteq \ell$, and $m_1'(x) \neq m_2'(x)$. From the rule, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$ and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$. From this and $\Delta_1'(x) \sqsubseteq \ell$, it follows that $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{y \in free(e)} \Delta_2(y) \sqsubseteq \ell$. From $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$, $\bigwedge_{y \in free(e)} \Delta_2(y) \sqsubseteq \ell$, and (2), it follows that $\bigwedge_{y \in free(e)} m_1(y) = m_2(y)$. From this, it follows that $[\![e]\!](m_1) = [\![e]\!](m_2)$. From this, $m_1' = m_1[x \mapsto [\![e]\!](m_1)]$, and $m_2' = m_2[x \mapsto [\![e]\!](m_2)]$, it follows that $m_1'(x) = m_2'(x)$, leading to a contradiction.

- Rule F-IfTrue. From the rule, it follows that $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From this and (4), it follows that $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, $m_1 = m_1'$, $\tau_1 = \epsilon$, $c_1' = c'$; $\textbf{set pc to } \Delta_1(\mathtt{pc}_u)$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From the rule, it also follows that $[\![e]\!](m_1) = \textbf{tt}$. There are two cases:
  - $[\![e]\!](m_2) = \textbf{tt}$. From this, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = c'$; $\textbf{set pc to } \Delta_2(\mathtt{pc}_u)$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$. From this, $m_1 = m_1'$, $m_2 = m_2'$, $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$, $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$, and (2), there are two cases:
    - $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$ (or vice versa). From the rule, it follows that $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$ and $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$. From this and $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) \sqsubseteq \ell$. From this, $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$. From $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{y \in free(e)} \Delta_1(y) = \Delta_2(y)$. From $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$ and $\bigwedge_{y \in free(e)} \Delta_1(y) = \Delta_2(y)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) = \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)$. From this, $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, and $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, it follows that $\Delta_2'(x) = \Delta_1'(x)$. From this and $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$, it follows that $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
    - $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$, $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$, and $\Delta_1'(\mathtt{pc}_u) \neq \Delta_2'(\mathtt{pc}_u)$. We have already shown above that from $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows $\Delta_2'(\mathtt{pc}_u) = \Delta_1'(\mathtt{pc}_u)$, which contradicts $\Delta_1'(\mathtt{pc}_u) \neq \Delta_2'(\mathtt{pc}_u)$.
  - $[\![e]\!](m_2) = \textbf{ff}$. From this, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = c''$; $\textbf{set pc to } \Delta_2(\mathtt{pc}_u)$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. The rest of the proof is similar to the case above.

- Rule F-IfFalse. The proof of this case is similar to that of F-IfTrue.
- Rule F-WhileTrue. The proof of this case is similar to that of F-IfTrue.
- Rule F-WhileFalse. The proof of this case is similar to that of F-IfTrue.
- Rule F-Select. From the rule, it follows that $c_1 = \|x \leftarrow \texttt{SELECT}\ \varphi\|$. From this and (4), it follows that $c_2 = \|x \leftarrow \texttt{SELECT}\ \varphi\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v)]$, $m_1' = m_1[x \mapsto r_1]$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, where $free(\varphi) = \{v_1, \ldots, v_n\}$, $q_1 = \texttt{SELECT}\ \varphi[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m_1),]$, and $[\![q_1]\!](\langle s_1, ctx_1\rangle) = (\langle s_1', ctx_1'\rangle, r_1, \epsilon, \epsilon)$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = \|x \leftarrow q\|$, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_2(v)]$, $m_2' = m_2[x \mapsto r_2]$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, where $free(\varphi) = \{v_1, \ldots, v_n\}$, $q_2 = \texttt{SELECT}\ \varphi[v_1 \mapsto [\![v_1]\!](m_2), \ldots, v_n \mapsto [\![v_n]\!](m_2),]$, and $[\![q_2]\!](\langle s_2, ctx_2\rangle) = (\langle s_2', ctx_2'\rangle, r_2, \epsilon, \epsilon)$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$. From this and (2), there are three cases:
  - $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_2'(x) \not\sqsubseteq \ell$ (or vice versa). From $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v)]$, it follows that $\Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\texttt{pc}_u) \sqsubseteq \ell$, $L_{\mathcal{Q}}(\Delta_1, q_1) \sqsubseteq \ell$, and $\bigwedge_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From $\Delta_1(\texttt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\texttt{pc}_u) \sqsubseteq \ell$. From $\bigwedge_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{v \in free(\varphi)} \Delta_2(v) \sqsubseteq \ell$ and $\bigwedge_{v \in free(\varphi)} m_1(v) = m_2(v)$. From $\bigwedge_{v \in free(\varphi)} m_1(v) = m_2(v)$, it follows that $q_1 = q_2$. From $q_1 = q_2$, $L_{\mathcal{Q}}(\Delta_1, q_1) \sqsubseteq \ell$, (2), and Proposition D.9, it follows that $L_{\mathcal{Q}}(\Delta_2, q_2) \sqsubseteq \ell$. From $\Delta_2(\texttt{pc}_u) \sqsubseteq \ell$, $L_{\mathcal{Q}}(\Delta_2, q_2) \sqsubseteq \ell$, $\bigwedge_{v \in free(\varphi)} \Delta_2(v) \sqsubseteq \ell$, and point (3) in the notion of disclosure order, it follows that $\Delta_2(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_2(v) \sqsubseteq \ell$. From this and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_2(v)]$, it follows that $\Delta_2'(x) \sqsubseteq \ell$, leading to a contradiction.
  - $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_1'(x) \sqsubseteq \ell$, and $\Delta_1'(x) \neq \Delta_2'(x)$. From $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v)]$, it follows that $\Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\texttt{pc}_u) \sqsubseteq \ell$, $L_{\mathcal{Q}}(\Delta_1, q_1) \sqsubseteq \ell$, and $\bigwedge_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From $\Delta_1(\texttt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\texttt{pc}_u) = \Delta_1(\texttt{pc}_u)$. From $\bigwedge_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{v \in free(\varphi)} \Delta_2(v) = \Delta_1(v)$ and $\bigwedge_{v \in free(\varphi)} m_1(v) = m_2(v)$. From $\bigwedge_{v \in free(\varphi)} m_1(v) = m_2(v)$, it follows that $q_1 = q_2$. From $L_{\mathcal{Q}}(\Delta_1, q_1) \sqsubseteq \ell$, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q_1)} \bigsqcup_{q' \in Q} \Delta_1(q') \sqsubseteq \ell$. From this, $\bigwedge_{Q \in supp_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') = \Delta_2(q')$. From this, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') = \bigwedge_{Q \in supp_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_2(q')$. From this and $q_1 = q_2$, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') = \bigwedge_{Q \in supp_{D,\Gamma}(q_2)} \bigwedge_{q' \in Q} \Delta_2(q')$. From this, it follows that $L_{\mathcal{Q}}(\Delta_1, q_1) = L_{\mathcal{Q}}(\Delta_2, q_2)$. From $\Delta_2(\texttt{pc}_u) = \Delta_1(\texttt{pc}_u)$, $\bigwedge_{v \in free(\varphi)} \Delta_2(v) = \Delta_1(v)$, $L_{\mathcal{Q}}(\Delta_1, q_1) = L_{\mathcal{Q}}(\Delta_2, q_2)$, $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v)]$, and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_2(v)]$, it follows that $\Delta_1'(x) = \Delta_2'(x)$, leading to a contradiction.
  - $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_1'(x) \sqsubseteq \ell$, and $m_1'(x) \neq m_2'(x)$. From $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v)]$, it follows that $\Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in free(\varphi)} m_1(v) = m_2(v)$. From this, it follows that $q_1 = q_2$. From $\Delta_1(\texttt{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in free(\varphi)} \Delta_1(v) \sqsubseteq \ell$, it also follows that $L_{\mathcal{Q}}(\Delta_1, q_1) \sqsubseteq \ell$. From this, $q_1 = q_2$, (2), and Proposition D.9, it follows that $L_{\mathcal{Q}}(\Delta_2, q_2) \sqsubseteq \ell$. From this, $L_{\mathcal{Q}}(\Delta_2, q_2) \sqsubseteq \ell$, and (2), it follows that $[q_1]^{db_1} = [q_2]^{db_2}$, where db_1 and db_2 are the database states in s_1 and s_2 respectively. From the database semantics, it follows that $m_1'(x) = r_1 = [q_1]^{db_1}$ and $m_2'(x) = r_2 = [q_2]^{db_2}$. From this and $[db_1]^{s_1} = [db_2]^{s_2}$, it follows that $m_1'(x) = m_2'(x)$, leading to a contradiction.
- Rule F-UpdateDatabaseOk. Without loss of generality, we assume that the query is an `INSERT` query. From the rule, it follows that $c_1 = \|x \leftarrow T \oplus \overline{e}\|$. From this and (4), it follows that $c_2 = \|x \leftarrow T \oplus \overline{e}\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \|x \leftarrow T \oplus \overline{e}\|$, it follows that $\Delta_1' = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\texttt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i), x \mapsto \Delta_1(\texttt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i)]$, $m_1' = m_1[x \mapsto r_1]$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, where $\overline{e} = (e_1, \ldots, e_n)$, $\overline{v}_1 = ([\![e_1]\!](m_1), \ldots, [\![e_n]\!](m_1))$, $q_1 = T \oplus \overline{v}_1$, and $[\![q_1]\!](\langle s_1, ctx_1\rangle) = (\langle s_1', ctx_1'\rangle, r_1, \epsilon, \epsilon)$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = \|x \leftarrow T \oplus \overline{e}\|$, it follows that $\Delta_2' = \Delta_2[T(\overline{v}_2) \mapsto \Delta_2(\texttt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_2(e_i), x \mapsto \Delta_2(\texttt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_2(e_i)]$, $m_2' = m_2[x \mapsto r_2]$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, where $\overline{e} = (e_1, \ldots, e_n)$, $\overline{v}_2 = ([\![e_1]\!](m_2), \ldots, [\![e_n]\!](m_2))$, $q_2 = T \oplus \overline{v}_2$, and $[\![q_2]\!](\langle s_2, ctx_2\rangle) = (\langle s_2', ctx_2'\rangle, r_2, \epsilon, \epsilon)$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$. From this and

(2), there are several cases:

1. $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_2'(x) \not\sqsubseteq \ell$ (or vice versa). From $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_1'(x) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$. From $\bigwedge_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{1\leq i\leq n} \Delta_2(e_i) \sqsubseteq \ell$. From this, $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i) \sqsubseteq \ell$. From this and $\Delta_2'(x) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i)$, it follows that $\Delta_2'(x) \sqsubseteq \ell$, leading to a contradiction.
2. $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_2'(x) \sqsubseteq \ell$, and $\Delta_1(x) \neq \Delta_2(x)$. From $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_1'(x) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\mathtt{pc}_u) = \Delta_1(\mathtt{pc}_u)$. From $\bigwedge_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{1\leq i\leq n} \Delta_2(e_i) = \Delta_1(e_i)$. From this, $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)$. From this and $\Delta_2'(x) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i)$, it follows that $\Delta_2'(x) = \Delta_1'(x)$, leading to a contradiction.
3. $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_2'(x) \sqsubseteq \ell$, and $m_1'(x) \neq m_2'(x)$. From the database semantics, both r_1 and r_2 are $\top$. From this, $m_1'(x) = r_1$, and $m_2'(x) = r_2$, it follows that $m_1'(x) = m_2'(x)$, leading to a contradiction.
4. $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_2'(T(\overline{v}_1)) \not\sqsubseteq \ell$. From $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_1)) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this, $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$, $\bigwedge_{1\leq i\leq n} \Delta_2(e_i) \sqsubseteq \ell$ and $\bigwedge_{1\leq i\leq n} [\![e_i]\!](m_1) = [\![e_i]\!](m_2)$. From $\bigwedge_{1\leq i\leq n} [\![e_i]\!](m_1) = [\![e_i]\!](m_2)$, it follows that $\overline{v}_1 = \overline{v}_2$. From $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1\leq i\leq n} \Delta_2(e_i) \sqsubseteq \ell$, it follows that $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i) \sqsubseteq \ell$. From this, $\Delta_2'(T(\overline{v}_2)) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i)$, and $\overline{v}_1 = \overline{v}_2$, it follows that $\Delta_2'(T(\overline{v}_2)) \sqsubseteq \ell$, leading to a contradiction.
5. $\Delta_1'(T(\overline{v}_2)) \sqsubseteq \ell$ and $\Delta_2'(T(\overline{v}_2)) \not\sqsubseteq \ell$. There are two cases:
   - $\overline{v}_1 = \overline{v}_2$. We already proved above (case 4) that this leads to a contradiction.
   - $\overline{v}_1 \neq \overline{v}_2$. From $\Delta_2'(T(\overline{v}_2)) \not\sqsubseteq \ell$ and $\Delta_2'(T(\overline{v}_2)) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i)$, it follows that $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i) \not\sqsubseteq \ell$. From (3), it follows that $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$. From this and $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i) \not\sqsubseteq \ell$, it follows that $\bigsqcup_{1\leq i\leq n} \Delta_2(e_i) \not\sqsubseteq \ell$. From the rule, it follows that $\bigsqcup_{1\leq i\leq n} \Delta_2(e_i) \sqsubseteq \Delta_2(T(\overline{v}_2))$. From this and $\bigsqcup_{1\leq i\leq n} \Delta_2(e_i) \not\sqsubseteq \ell$, it follows that $\Delta_2(T(\overline{v}_2)) \not\sqsubseteq \ell$. From this and (2), it follows that $\Delta_1(T(\overline{v}_2)) \not\sqsubseteq \ell$. From $\Delta_1' = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i), x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)]$ and $\overline{v}_1 \neq \overline{v}_2$, it follows that $\Delta_1'(T(\overline{v}_2)) = \Delta_1(T(\overline{v}_2))$. From this and $\Delta_1'(T(\overline{v}_2)) \sqsubseteq \ell$, it follows that $\Delta_1(T(\overline{v}_2)) \sqsubseteq \ell$. This contradicts $\Delta_1(T(\overline{v}_2)) \not\sqsubseteq \ell$.
6. $\Delta_2'(T(\overline{v}_2)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_2)) \not\sqsubseteq \ell$. The proof of this case is similar to that of case 4.
7. $\Delta_2'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_1)) \not\sqsubseteq \ell$. The proof of this case is similar to that of case 5.
8. $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$, $\Delta_2'(T(\overline{v}_1)) \sqsubseteq \ell$, and $\Delta_1'(T(\overline{v}_1)) \neq \Delta_2'(T(\overline{v}_1))$. From $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_1)) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this, $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1\leq i\leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(\mathtt{pc}_u) = \Delta_1(\mathtt{pc}_u)$, $\bigwedge_{1\leq i\leq n} \Delta_2(e_i) = \Delta_1(e_i)$ and $\bigwedge_{1\leq i\leq n} [\![e_i]\!](m_1) = [\![e_i]\!](m_2)$. From $\bigwedge_{1\leq i\leq n} [\![e_i]\!](m_1) = [\![e_i]\!](m_2)$, it follows that $\overline{v}_1 = \overline{v}_2$. From $\Delta_2(\mathtt{pc}_u) = \Delta_1(\mathtt{pc}_u)$ and $\bigwedge_{1\leq i\leq n} \Delta_2(e_i) = \Delta_1(e_i)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i)$. From this, $\overline{v}_1 = \overline{v}_2$, $\Delta_1'(T(\overline{v}_1)) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)$, $\Delta_2'(T(\overline{v}_2)) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_2(e_i)$, it follows that $\Delta_2'(T(\overline{v}_1)) = \Delta_1(T(\overline{v}_1))$.
9. $\Delta_1'(T(\overline{v}_2)) \sqsubseteq \ell$, $\Delta_2'(T(\overline{v}_2)) \sqsubseteq \ell$, and $\Delta_1'(T(\overline{v}_2)) \neq \Delta_2'(T(\overline{v}_2))$. The proof is similar to that of case 8.
10. There is a query q such that $L_{\mathcal{Q}}(\Delta_1', q) \sqsubseteq \ell$, $L_{\mathcal{Q}}(\Delta_2', q) \sqsubseteq \ell$, and $[q]^{db_1'} \neq [q]^{db_2'}$, where db_1' and db_2' are the databases in s_1' and s_2' respectively. From $L_{\mathcal{Q}}$'s definition, it follows $\bigsqcup_{Q\in supp_{D,\Gamma}(q)} \bigsqcup_{q'\in Q} \Delta_1'(q') \sqsubseteq \ell$ and $\bigsqcup_{Q\in supp_{D,\Gamma}(q)} \bigsqcup_{q'\in Q} \Delta_2'(q') \sqsubseteq \ell$. There are two cases:
    - There is a $Q \in supp_{D,\Gamma}(q)$ such that $T(\overline{v}_1) \notin Q$ and $T(\overline{v}_2) \notin Q$. From $L_{\mathcal{Q}}(\Delta_1', q) \sqsubseteq \ell$, it follows that $\bigwedge_{q'\in Q} \Delta_1'(q') \sqsubseteq \ell$. From $T(\overline{v}_1) \notin Q$ and $\Delta_1' = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i), x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1\leq i\leq n} \Delta_1(e_i)]$, it follows that $\bigwedge_{q'\in Q} \Delta_1'(q') = \Delta_1(q')$. From this and $\bigwedge_{q'\in Q} \Delta_1'(q') \sqsubseteq \ell$, it follows that $\bigwedge_{q'\in Q} \Delta_1(q') \sqsubseteq \ell$. From this, $Q \subset RC^{pred}$, and $L_{\mathcal{Q}}(\Delta, q'') = \Delta(q'')$ for any $q'' \in RC^{pred}$ (Proposition D.10), it follows that $\bigwedge_{q'\in Q} L_{\mathcal{Q}}(\Delta_1, q') \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{q'\in Q} [q']^{db_1'} = [q']^{db_2'}$, where db_1' and db_2' are the database in s_1' and s_2' respectively. From this, $T(\overline{v}_1) \notin Q$, $T(\overline{v}_2) \notin Q$, and the fact that we modify only the values of $T(\overline{v}_1)$ and $T(\overline{v}_2)$, it follows that $\bigwedge_{q'\in Q} [q']^{s_1'} = [q']^{s_2'}$. From this and Q determines q, it follows that $[q]^{s_1} = [q]^{s_2}$,

leading to a contradiction.

– For all $Q \in supp_{D,\Gamma}(q)$, $T(\overline{v}_1) \in Q$ or $T(\overline{v}_2) \in Q$. Assume that $T(\overline{v}_1) \in Q$ (the proof in case $T(\overline{v}_2) \in Q$ is analogous). From $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$, it follows that $\bigwedge_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$. From this and $T(\overline{v}_1) \in Q$, it follows that $\Delta'_1(T(\overline{v}_1)) \sqsubseteq \ell$. From this and $\Delta'_1(T(\overline{v}_1)) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this, it follows that $\bigwedge_{1 \leq i \leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{1 \leq i \leq n} [\![e_i]\!](m_1) = [\![e_i]\!](m_2)$. From this, it follows that $\overline{v}_1 = \overline{v}_2$. From $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$ and $Q \in supp_{D,\Gamma}(q)$, it follows that $\bigwedge_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$. Let q'' be a query in $Q \setminus \{T(\overline{v}_1)\}$. From this and $\bigwedge_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$, it follows that $\Delta'_1(q'') \sqsubseteq \ell$. From this, $\overline{v}_1 = \overline{v}_2$, $q'' \in Q \setminus \{T(\overline{v}_1)\}$, and $\Delta'_1 = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i), x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i)]$, it follows that $\Delta'_1(q'') = \Delta_1(q'')$. From this and $\Delta'_1(q'') \sqsubseteq \ell$, it follows that $\Delta_1(q'') \sqsubseteq \ell$. From this, $q'' \in RC^{pred}$, and $L_{\mathcal{Q}}(\Delta, q) = \Delta(q)$ for all $q \in RC^{pred}$ (Proposition D.10), it follows that $L_{\mathcal{Q}}(\Delta_1, q'') \sqsubseteq \ell$. From this and (2), it follows that $[q'']^{db_1} = [q'']^{db_2}$, where db_1 and db_2 are the databases in s_1 and s_2 respectively. From this, $\overline{v}_1 = \overline{v}_2$, and the fact that the update operation only modifies the value of $T(\overline{v}_1)$ and $T(\overline{v}_2)$, it follows that $[q'']^{db'_1} = [q'']^{db'_2}$, where db'_1 and db'_2 are the databases in s'_1 and s'_2 respectively. Since q'' is an arbitrary query in $Q \setminus \{T(\overline{v}_1)\}$, it follows that $[q'']^{db'_1} = [q'']^{db'_2}$ for all $q'' \in Q \setminus \{T(\overline{v}_1)\}$. From $\overline{v}_1 = \overline{v}_2$ and the database semantics, it also follows that $[T(\overline{v}_1)]^{db'_1} = [T(\overline{v}_1)]^{db'_2} = \top$. From this, $[q'']^{db'_1} = [q'']^{db'_2}$ for all $q'' \in Q \setminus \{T(\overline{v}_1)\}$, and $Q = \{T(\overline{v}_1)\} \cup (Q \setminus \{T(\overline{v}_1)\})$, it follows that $[q'']^{db'_1} = [q'']^{db'_2}$ for all $q'' \in Q$. From this and $Q \in supp_{D,\Gamma}(q)$ (and therefore Q determines q), it follows that $[q]^{db'_1} = [q]^{db'_2}$, leading to a contradiction.

- Rule F-UpdateConfigurationOk. From the rule, it follows that $c_1 = \|x \leftarrow q\|$. From this and (4), it follows that $c_2 = \|x \leftarrow q\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v)]$, $m'_1 = m_1[x \mapsto r_1]$, $q'_1 = q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m_1)]$ $\tau_1 = \epsilon$, $c'_1 = \varepsilon$, where $[\![q'_1]\!](\langle s_1, ctx_1 \rangle) = (\langle s'_1, ctx'_1 \rangle, r_1, \epsilon, \epsilon)$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle$ and $c_2 = \|x \leftarrow q\|$, it follows that $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v)]$, $m'_2 = m_2[x \mapsto r_2]$, $q'_2 = q[v_1 \mapsto [\![v_1]\!](m_2), \ldots, v_n \mapsto [\![v_n]\!](m_2)]$ $\tau_2 = \epsilon$, $c'_2 = \varepsilon$, where $[\![q'_2]\!](\langle s_2, ctx_2 \rangle) = (\langle s'_2, ctx'_2 \rangle, r_2, \epsilon, \epsilon)$. Note that the execution of the query q alters only the database configuration; it does not modify the content of the database. Assume, for contradiction's sake, that $\langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \not\approx_\ell \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle$. There are three cases:
  – $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_2(x) \not\sqsubseteq \ell$ (or vice versa). From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{v \in vars(q)} \Delta_2(v) \sqsubseteq \ell$. From this, it follows that $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v) \sqsubseteq \ell$. From this and $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v)]$, it follows that $\Delta'_2(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
  – $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_2(x) \sqsubseteq \ell$, and $\Delta'_1(x) \neq \Delta'_2(x)$. From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$ and $\bigwedge_{v \in vars(q)} \Delta_1(v) = \Delta_2(v)$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v) \sqsubseteq \ell$. From this and $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v)]$, it follows that $\Delta'_1(\mathtt{pc}_u) = \Delta'_2(\mathtt{pc}_u)$, leading to a contradiction.
  – $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_2(x) \sqsubseteq \ell$, and $m'_1(x) \neq m'_2(x)$. From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v) \sqsubseteq \ell$. From this, it follows that $\bigwedge_{v \in vars(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in vars(q)} m_1(v) = m_2(v)$. From this, it follows that $q'_1 = q'_2$. From this and (1), it follows that $r_1 = r_2$. From this, $m'_1 = m_1[x \mapsto r_1]$, and $m'_2 = m_2[x \mapsto r_2]$, it follows that $m'_1(x) = m'_2(x)$, leading to a contradiction.

Induction Step. The proof of the induction step directly follows from the induction hypothesis for all rules except F-AsUser. For the F-AsUser rule, the proof can be done by case distinction on the executed query. The proofs for the various cases are similar to that of the rules F-Select, F-UpdateDatabaseOk, and F-UpdateConfigurationOk. □

Lemma D.19 states that, under appropriate conditions, performing a step of execution in two ℓ-equivalent states with the same initial code results in the same code.

Lemma D.19. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $c_1 = c_2$,
5. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
6. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,

then $c_1' = c_2'$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold:

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $c_1 = c_2$,
5. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
6. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$.

We prove our claim by induction on the rules defining $\rightsquigarrow_u$.

Base Case. The proof of most of the rules, e.g., F-Assign or F-Out, is trivial. The only interesting cases are the branching statements and the expansion procedure.

- Rule F-IfTrue. From the rule, it follows that $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From this and (4), it follows that $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, $m_1 = m_1'$, $\tau_1 = \epsilon$, $c_1' = c'$; **set pc to** $\Delta_1(\mathtt{pc}_u)$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From the rule, it also follows that $[\![e]\!](m_1) = \textbf{tt}$. From (3) and $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_1(e) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in vars(e)} m_1(e) = m_2(e)$. From this and $[\![e]\!](m_1) = \textbf{tt}$, it follows that $[\![e]\!](m_2) = \textbf{tt}$. Therefore, by applying the rule F-IfTrue to $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, we obtain that $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = c'$; **set pc to** $\Delta_2(\mathtt{pc}_u)$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. Furthermore, from $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$. Therefore, $c_1' = c_2'$.
- Rule F-IfFalse. The proof is similar to that for the F-IfTrue case.
- Rule F-WhileTrue. The proof is similar to that for the F-IfTrue case.
- Rule F-WhileFalse. The proof is similar to that for the F-IfTrue case.
- Rule F-Expand. From the rule, it follows that $c_1 = x \leftarrow q$. From this and (4), it follows that $c_2 = x \leftarrow q$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = x \leftarrow q$, it follows that $\Delta_1' = \Delta_1$, $m_1 = m_1'$, $\tau_1 = \epsilon$, $c_1' = expand(s_1, x, q, u)$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = x \leftarrow q$, it follows that $\Delta_2' = \Delta_2$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = expand(s_2, x, q, u)$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. There are a number of cases depending on q:
  - q is `SELECT` φ. For `SELECT` queries, the result of the expansion procedure is the same for any two database states s_1 and s_2. Therefore, it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.
  - q is `INSERT` $\overline{e}$ `INTO` T. The expansion procedure relies only on the *allowed* and *apply* procedures, which, in turn, depend only on the configuration of the database state. From this and (1), it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.
  - q is `DELETE` $\overline{e}$ `FROM` T. The proof of this case is similar to that of `INSERT` $\overline{e}$ `INTO` T.
  - q is `GRANT` p `TO` u. The expansion procedure relies only on the *allowed* and *apply* procedures, which, in turn, depend only on the configuration of the database state. From this and (1), it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.
  - q is `GRANT` p `TO` u `WITH GRANT OPTION`. The proof of this case is similar to that of `GRANT` p `TO` u.
  - q is `REVOKE` p `FROM` u. The proof of this case is similar to that of `GRANT` p `TO` u.
  - q is a `CREATE` queries. For `CREATE` queries, the result of the expansion procedure is the same for any two database states s_1 and s_2. Therefore, it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.

– q is `ADD USER` u'. For `ADD USER` queries, the result of the expansion procedure is the same for any two database states s_1 and s_2. Therefore, it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.

Induction Step. The proof of the induction step directly follows from the induction hypothesis. □

Lemma D.20 presents some results about computations involving **if** statements.

Lemma D.20. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ *or* $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5. $c_1 = c_2$,
6. $first(c_1) =$ ***if*** e ***then*** c' ***else*** c'',
7. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
8. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,

then one of the following conditions hold:

(a) $[\![e]\!](m_1) =$ ***tt*** *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(b) $[\![e]\!](m_1) =$ ***ff*** *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c'', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(c) $[\![e]\!](m_2) =$ ***tt*** *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(d) $[\![e]\!](m_2) =$ ***ff*** *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c'', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(e) there are $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *and* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *and* $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_1'' = c_2''$, *and* $\Delta_1''(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2''(\mathtt{pc}_u) \sqsubseteq \ell$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that:

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ or $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5. $c_1 = c_2$,
6. $first(c_1) =$ **if** e **then** c' **else** c'',
7. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
8. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,

From (5) and (6), it follows that $first(c_2) =$ **if** e **then** c' **else** c''. Without loss of generality, we assume that $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$. From this, $\Delta_1'(\mathtt{pc}_u) = \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)$. From this, (3), and $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$, it follows that $\Delta_1(e) \not\sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(e) \not\sqsubseteq \ell$. From this, (3), and $\Delta_2'(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)$, it follows that $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$. Without loss of generality, we assume that $c_1 =$ **if** e **then** c' **else** c'' (the proof in case $c_1 =$ **if** e **then** c' **else** c'' ; c_3 is similar). There are four cases:

- $[\![e]\!](m_1) = [\![e]\!](m_2) =$ **tt**. From the rules F-IfTrue and F-IfFalse, it follows that $c_1' = [c'$; **set pc to** $\Delta_1(\mathtt{pc}_u)]$ and $c_2 = [c'$; **set pc to** $\Delta_1(\mathtt{pc}_u)]$. There are three cases:
  - For all $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ such that $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$ (i.e., c' never terminates, produces an exception, or stucks starting from $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle$). In this case our claim is trivially satisfied.
  - For all $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ such that $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$ (i.e., c' never terminates, produces an exception, or stucks starting from $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle$). In this case our claim is trivially satisfied.
  - There exist $\langle \Delta_1'', \varepsilon, m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2'', \varepsilon, m_2'', \langle s_2'', ctx_2''\rangle\rangle$ such that $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', \varepsilon, m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', \varepsilon, m_2'', \langle s_2'', ctx_2''\rangle\rangle$.

From this, it follows that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle \overset{\tau_1'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_1''', [\varepsilon \ ; \ \textbf{set pc to } \Delta_1(\mathtt{pc}_u)], m_1'', \langle s_1'', ctx_1'' \rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle \overset{\tau_2'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_2''', [\varepsilon \ ; \ \textbf{set pc to } \Delta_2(\mathtt{pc}_u)], m_2'', \langle s_2'', ctx_2'' \rangle\rangle$. By applying the F-ExpandedCode and the F-SeqEmpty rules, we obtain $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle \overset{\tau_1'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_1''', [\textbf{set pc to } \Delta_1(\mathtt{pc}_u)], m_1'', \langle s_1'', ctx_1'' \rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle \overset{\tau_2'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_2''', [\textbf{set pc to } \Delta_2(\mathtt{pc}_u)], m_2'', \langle s_2'', ctx_2'' \rangle\rangle$. By applying the F-ExpandedCode and the F-UpdateLabels rules, we obtain $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle \overset{\tau_1'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_1'''[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u)], [\varepsilon], m_1'', \langle s_1'', ctx_1'' \rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle \overset{\tau_2'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_2'''[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u)], [\varepsilon], m_2'', \langle s_2'', ctx_2'' \rangle\rangle$ (observe that $\Delta_1'''[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u)] = \Delta_1'''$ and $\Delta_2'''[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u)] = \Delta_2'''$). From this, it directly follows our claim (since the code is the same in both final configurations and from (3), it follows that the $\mathtt{pc}_u$ is below ℓ in both configurations).

- $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{ff}$. The proof of this case is similar to that of $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{tt}$.
- $[\![e]\!](m_1) = \mathbf{tt}$ and $[\![e]\!](m_2) = \mathbf{ff}$. The proof of this case is similar to that of $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{tt}$.
- $[\![e]\!](m_1) = \mathbf{ff}$ and $[\![e]\!](m_2) = \mathbf{tt}$. The proof of this case is similar to that of $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{tt}$.

This completes the proof of our claim. □

Lemma D.20 presents some results about computations involving **while** statements.

Lemma D.21. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle\rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ *or* $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5. $c_1 = c_2$,
6. $first(c_1) = \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c'$,
7. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle\rangle \overset{\tau_1}{\rightsquigarrow}{}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle$,
8. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle\rangle \overset{\tau_2}{\rightsquigarrow}{}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle$,

then one of the following conditions hold:

(a) $[\![e]\!](m_1) = \textbf{\textit{tt}}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1'' \rangle\rangle$ *such that* $\langle \Delta_1', c', m_1', \langle s_1', ctx_1' \rangle\rangle \overset{\tau_1'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1'' \rangle\rangle$, $c_1'' \neq \varepsilon$,

(b) $[\![e]\!](m_2) = \textbf{\textit{tt}}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2'' \rangle\rangle$ *such that* $\langle \Delta_2', c', m_2', \langle s_2', ctx_2' \rangle\rangle \overset{\tau_2'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2'' \rangle\rangle$, $c_2'' \neq \varepsilon$,

(c) there are $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1'' \rangle\rangle$ *and* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2'' \rangle\rangle$ *such that* $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle \overset{\tau_1'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1'' \rangle\rangle$ *and* $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle \overset{\tau_2'}{\rightsquigarrow}{}_u^{\,*} \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2'' \rangle\rangle$, $c_1'' = c_2''$, *and* $\Delta_1''(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2''(\mathtt{pc}_u) \sqsubseteq \ell$.

Proof. The proof is similar to that of Lemma D.20. □

D.3.7 Lemmas about the global semantics

Here we present some auxiliary results about the global semantics of our enforcement mechanism.

Lemma D.22 states that, under appropriate conditions, performing a step of (global) execution in two ℓ-equivalent states with the same initial code and scheduler results in configurations with the same code and scheduler.

Lemma D.22. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, \mathcal{S}_1 \rangle$, $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle$, $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, \mathcal{S}_2 \rangle$, and $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2' \rangle, \mathcal{S}_2' \rangle$ be four global configurations, $\ell \in \mathcal{L}$ be a label, n' be the first value in $\mathcal{S}$, $n = (n' \bmod |C_1|) + 1$, and u be the user associated with the n-th program in C_1. If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, \mathcal{S}_1 \rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, \mathcal{S}_2 \rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$,
5. $C_1 = C_2$,
6. $\mathcal{S}_1 = \mathcal{S}_2$,
7. $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, \mathcal{S}_1 \rangle \overset{\tau_1}{\rightsquigarrow}{}_u \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle$,

8. $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$,

then $C_1' = C_2'$ *and* $\mathcal{S}_1' = \mathcal{S}_2'$.

Proof. The claim directly follows from (4), (5), and Lemma D.19 (together with (1) and (3)). □

Lemma D.23 states some properties of the execution of **if** statements in the global semantics.

Lemma D.23. *Let* sec_0 *be the policy used to initialize the monitor,* $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle$, $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$, $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$, *and* $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$ *be four global configurations, and* $\ell \in \mathcal{L}$ *be a label,* n' *be the first value in* $\mathcal{S}$, $n = (n' \bmod |C_1|) + 1$, *and* u *be the user associated with the* n*-th program in* C_1. *If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ *or* $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5. $C_1 = C_2$,
6. $first(C_1(n)) = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c'\ \textbf{\textit{else}}\ c''$,
7. $\mathcal{S}_1 = \mathcal{S}_2$,
8. $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$,
9. $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$,

then one of the following conditions hold:

(a) $[\![e]\!](m_1) = \textbf{\textit{tt}}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(b) $[\![e]\!](m_1) = \textbf{\textit{ff}}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c'', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(c) $[\![e]\!](m_2) = \textbf{\textit{tt}}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(d) $[\![e]\!](m_2) = \textbf{\textit{ff}}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c'', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(e) *there are* $\langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$ *and* $\langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2''\rangle, \mathcal{S}_2''\rangle$ *such that* $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$ *and* $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2''\rangle, \mathcal{S}_2''\rangle$, $C_1'' = C_2''$, $\mathcal{S}_1'' = \mathcal{S}_2''$, *and* $\Delta_1''(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2''(\mathtt{pc}_u) \sqsubseteq \ell$.

Proof. From (6) and the rules F-IfTrue and F-IfFalse, it follows that the only applicable rule in the global semantics is F-Atomic-Statement. Our claim directly follows from this, (3), (4), Lemma D.20, and the fact that the F-Atomic-Statement rule does not modify the scheduler. □

Lemma D.24 states some properties of the execution of **while** statements in the global semantics.

Lemma D.24. *Let* sec_0 *be the policy used to initialize the monitor,* $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle$, $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$, $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$, *and* $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$ *be four global configurations, and* $\ell \in \mathcal{L}$ *be a label,* n' *be the first value in* $\mathcal{S}$, $n = (n' \bmod |C_1|) + 1$, *and* u *be the user associated with the* n*-th program in* C_1. *If the following conditions hold:*

1. $s_1 \equiv^{cfg} s_2$,
2. $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$,
3. $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4. $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ *or* $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5. $C_1 = C_2$,
6. $first(C_1(n)) = \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c'$,
7. $\mathcal{S}_1 = \mathcal{S}_2$,
8. $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$,
9. $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$,

then one of the following conditions hold:

(a) $[\![e]\!](m_1) = \textbf{\textit{tt}}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(b) $[\![e]\!](m_1) = \textbf{\textit{ff}}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c'', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(c) $[\![e]\!](m_2) = \textbf{\textit{tt}}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(d) $[\![e]\!](m_2) = \mathit{fff}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2'' \rangle \rangle$ *such that* $\langle \Delta_2', c'', m_2', \langle s_2', ctx_2' \rangle \rangle \overset{\tau_2'}{\rightsquigarrow}{}_u^{*} \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2'' \rangle \rangle$, $c_2'' \neq \varepsilon$,

(e) *there are* $\langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1'' \rangle, \mathcal{S}_1'' \rangle$ *and* $\langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2'' \rangle, \mathcal{S}_2'' \rangle$ *such that* $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle \overset{\tau_1'}{\rightsquigarrow}{}_u^{*} \langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1'' \rangle, \mathcal{S}_1'' \rangle$ *and* $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2' \rangle, \mathcal{S}_2' \rangle \overset{\tau_2'}{\rightsquigarrow}{}_u^{*} \langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2'' \rangle, \mathcal{S}_2'' \rangle$, $C_1'' = C_2''$, $\mathcal{S}_1'' = \mathcal{S}_2''$, *and* $\Delta_1''(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2''(\mathtt{pc}_u) \sqsubseteq \ell$.

Proof. From (6) and the rules F-WhileTrue and F-WhileFalse, it follows that the only applicable rule in the global semantics is F-Atomic-Statement. Our claim directly follows from this, (3), (4), Lemma D.21, and the fact that the F-Atomic-Statement rule does not modify the schedulers. □

D.3.8 Bisimulations

Here we introduce bisimulations for our setting, and we prove some key results about them.

We first introduce some machinery. Let $\sigma_1 = \langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, \mathcal{S}_1 \rangle$ and $\sigma_2 = \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, \mathcal{S}_2 \rangle$ be two global configurations and $i \in \mathbb{N}$ be an integer. We say that σ_2 is reachable in at most i steps from σ_1, denoted $reach^i(\sigma_1, \sigma_2)$, iff there exists an $i' \leq i$ such that $\sigma_1 \overset{\tau}{\rightsquigarrow}{}^{i'} \sigma_2$. The current program in σ_1, denoted $currPrg(\sigma_1)$, is c and the current memory $currMem(\sigma_1)$ is m, where n' is the first element in $\mathcal{S}_1$, $n = (n' \bmod |C_1|) + 1$, $C_1(n) = \langle u, c \rangle$, and $M(n) = \langle u, c \rangle$. Furthermore, the current pc in σ_1, denoted $currPc(\sigma_1)$, is $\Delta_1(\mathtt{pc}_u)$, where n' is the first element in $\mathcal{S}_1$, $n = (n' \bmod |C_1|) + 1$, and $C_1(n) = \langle u, c \rangle$. Finally, the current user in σ_1, denoted $currUsr(\sigma_1)$, is u, where $currPrg(\sigma_1) = \langle u, c \rangle$. Given a local configuration σ and a user u, $term(\sigma, u) = \top$ iff there exists a $\langle \Delta, c, m, \langle s, ctx \rangle \rangle$ such that $\sigma \overset{\tau}{\rightsquigarrow}{}_u^{*} \langle \Delta, c, m, \langle s, ctx \rangle \rangle$ and $c = \varepsilon$. Given a label ℓ and a user u, we denote by $notBelow(\sigma, \ell, u) = \top$ iff for all $\langle \Delta, c, m, \langle s, ctx \rangle \rangle$ such that $\sigma \overset{\tau}{\rightsquigarrow}{}_u^{*} \langle \Delta, c, m, \langle s, ctx \rangle \rangle$, then $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.

We are now ready to formalize bisimulations.

Definition D.4. Let $\sigma_1 = \langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, \mathcal{S}_1 \rangle$ and $\sigma_2 = \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, \mathcal{S}_2 \rangle$ be two global configurations, $i, j \in \mathbb{N}$ be integers, and $\ell \in \mathcal{L}$ be a label. Furthermore, let R be a binary relation over global configurations. We say that R is a $(\sigma_1, \sigma_2, i, j, \ell)$-*bisimulation* iff for all $\sigma_1' = \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle$ and $\sigma_2' = \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2' \rangle, \mathcal{S}_2' \rangle$ such that $\sigma_1' R \sigma_2'$, then the following conditions hold:

1. $reach^i(\sigma_1, \sigma_1')$ and $reach^j(\sigma_2, \sigma_2')$.
2. $\sigma_1' \approx_\ell \sigma_2'$.
3. $\sigma_1' \equiv^{cfg} \sigma_2'$.
4. $C_1' = C_2'$.
5. $\mathcal{S}_1' = \mathcal{S}_2'$.
6. $currPc(\sigma_1) \sqsubseteq \ell$ and $currPc(\sigma_2) \sqsubseteq \ell$.
7. If $\sigma_1' \overset{\tau_1'}{\rightsquigarrow} \sigma_1''$, $\sigma_2' \overset{\tau_2'}{\rightsquigarrow} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, and $currPc(\sigma_1'') \sqsubseteq \ell \wedge currPc(\sigma_2'') \sqsubseteq \ell$, then $\sigma_1'' R \sigma_2''$.
8. If $\sigma_1' \overset{\tau_1'}{\rightsquigarrow} \sigma_1''$, $\sigma_2' \overset{\tau_2'}{\rightsquigarrow} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, and $currPc(\sigma_1'') \not\sqsubseteq \ell \vee currPc(\sigma_2'') \not\sqsubseteq \ell$, then one of the following conditions hold:
   (a) for all σ_1'' and σ_2'' such that $\sigma_1' \overset{\tau_1''}{\rightsquigarrow}{}^{*} \sigma_1''$, $\sigma_2' \overset{\tau_2''}{\rightsquigarrow}{}^{*} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, $currPc(\sigma_1) \not\sqsubseteq \ell$ or $currPc(\sigma_2) \not\sqsubseteq \ell$, or
   (b) there are σ_1'' and σ_2'' such that $\sigma_1' \overset{\tau_1''}{\rightsquigarrow}{}^{*} \sigma_1''$, $\sigma_2' \overset{\tau_2''}{\rightsquigarrow}{}^{*} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, and $\sigma_1'' R \sigma_2''$. □

Lemmas D.25 and D.26 state that, under certain conditions, we can construct bisimulations.

Lemma D.25. *Let* sec_0 *be the policy used to initialize the monitor,* $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0 \rangle, \mathcal{S}_1^0 \rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0 \rangle, \mathcal{S}_2^0 \rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1 \rangle, \mathcal{S}_1^1 \rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1 \rangle, \mathcal{S}_2^1 \rangle$ *be four global configurations,* τ_1, τ_2 *be two traces,* u *be a user, and* $\ell \in \mathcal{L}$ *be a label. If the following conditions hold:*

1. $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,
2. $\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,
3. $\sigma_1^0 \approx_\ell \sigma_2^0$,
4. $\sigma_1^0 \equiv^{cfg} \sigma_2^0$,
5. $C_1^0 = C_2^0$,
6. $\mathcal{S}_1^0 = \mathcal{S}_2^0$,
7. $currPc(\sigma_1^0) \sqsubseteq \ell$ *and* $currPc(\sigma_2^0) \sqsubseteq \ell$,
8. $currPc(\sigma_1^1) \sqsubseteq \ell$ *and* $currPc(\sigma_2^1) \sqsubseteq \ell$,
9. $cl(auth(sec_0, ATK)) \sqsubseteq \ell$,

then $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$ *is a* $(\sigma_1^0, \sigma_2^0, 1, 1, \ell)$*-bisimulation.*

Proof. Let sec_0 be the policy used to initialize the monitor, $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0 \rangle, \mathcal{S}_1^0 \rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0 \rangle, \mathcal{S}_2^0 \rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1 \rangle, \mathcal{S}_1^1 \rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1 \rangle, \mathcal{S}_2^1 \rangle$ be four global configurations, τ_1, τ_2 be two traces, u be a user, and $\ell \in \mathcal{L}$ be a label. Furthermore, we assume that following conditions hold:

1. $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,
2. $\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,
3. $\sigma_1^0 \approx_\ell \sigma_2^0$,
4. $\sigma_1^0 \equiv^{cfg} \sigma_2^0$,
5. $C_1^0 = C_2^0$,
6. $\mathcal{S}_1^0 = \mathcal{S}_2^0$,
7. $currPc(\sigma_1^0) \sqsubseteq \ell$ and $currPc(\sigma_2^0) \sqsubseteq \ell$,
8. $currPc(\sigma_1^1) \sqsubseteq \ell$ and $currPc(\sigma_2^1) \sqsubseteq \ell$,
9. $cl(auth(sec_0, ATK)) \sqsubseteq \ell$.

We now show that $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$ is a $(\sigma_1^0, \sigma_2^0, 1, 1, \ell)$-bisimulation. We first need to show that for all $\sigma_1' = \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle$ and $\sigma_2' = \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2' \rangle, \mathcal{S}_2' \rangle$ such that $\sigma_1' R \sigma_2'$, the following conditions hold: (a) $reach^1(\sigma_1^0, \sigma_1')$ and $reach^1(\sigma_2^0, \sigma_2')$, (b) $\sigma_1' \approx_\ell \sigma_2'$, (c) $\sigma_1' \equiv^{cfg} \sigma_2'$, (d) $C_1' = C_2'$, (e) $\mathcal{S}_1' = \mathcal{S}_2'$, (f) $currPc(\sigma_1) \sqsubseteq \ell$ and $currPc(\sigma_2) \sqsubseteq \ell$. There are two cases:

- $(\sigma_1', \sigma_2') = (\sigma_1^0, \sigma_2^0)$. Then, (a) trivially follows since $reach^1(\sigma, \sigma)$ always holds. Moreover, (b)–(f) directly follow from (3)–(7).
- $(\sigma_1', \sigma_2') = (\sigma_1^1, \sigma_2^1)$. Then, (a) directly follows from (1) and (2). There are two cases:
  - (1) is obtained by applying the M-Eval-End rule. From this and (5), it follows that also (2) is obtained using the M-Eval-End rule. From this, (5), and the rule, we eliminate in both run the same components. From this and (3)–(7), it directly follows that (b)–(f) are satisfied.
  - (1) is obtained by applying the M-Eval-Step or M-Atomic-Statement rules. From this, (5), and (6), it follows that we perform one step of the local semantics for the same program in both runs. From this and (6), (e) directly follows. From (3)–(10), Lemmas D.17, D.18, and D.19, conditions (b)–(d) follow. Finally, condition (f) immediately follows from (8).

Therefore, the fact that R is a bisimulation directly follows from (i) the fact that (a)–(f) hold for (σ_1^0, σ_2^0) and (σ_1^1, σ_2^1), (ii) assumptions (1), (2), and $R = \{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$, and (3) there are no configurations that are reachable in 1 step from σ_1^0 and σ_2^0 other than σ_1^1 and σ_2^1. □

Lemma D.26. *Let* sec_0 *be the policy used to initialize the monitor,* $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0 \rangle, \mathcal{S}_1^0 \rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0 \rangle, \mathcal{S}_2^0 \rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1 \rangle, \mathcal{S}_1^1 \rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1 \rangle, \mathcal{S}_2^1 \rangle$ *be four global configurations,* τ_1, τ_2 *be two traces,* $u = currUsr(\sigma_1^1)$ *be a user, and* $\ell \in \mathcal{L}$ *be a label. If the following conditions hold:*

1. $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,
2. $\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,
3. $\sigma_1^0 \approx_\ell \sigma_2^0$,
4. $\sigma_1^0 \equiv^{cfg} \sigma_2^0$,
5. $C_1^0 = C_2^0$,
6. $\mathcal{S}_1^0 = \mathcal{S}_2^0$,
7. $currPc(\sigma_1^0) \sqsubseteq \ell$ *and* $currPc(\sigma_2^0) \sqsubseteq \ell$,
8. $currPc(\sigma_1^1) \not\sqsubseteq \ell$ *or* $currPc(\sigma_2^1) \not\sqsubseteq \ell$,
9. $cl(auth(sec_0, ATK)) \sqsubseteq \ell$,
10. *for all users* $u' \neq currUsr(\sigma_1^0)$, $\Delta_1^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$ *and* $\Delta_2^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$,
11. *whenever* $first(currPrg(\sigma_1^0)) = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c'\ \textbf{\textit{else}}\ c''$ *and* $[\![e]\!](M_1^0) = \textbf{\textit{tt}}$, *then* $term(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ *and* $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,
12. *whenever* $first(currPrg(\sigma_1^0)) = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c'\ \textbf{\textit{else}}\ c''$ *and* $[\![e]\!](M_1^0) = \textbf{\textit{ff}}$, *then* $term(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ *and* $notBelow(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,
13. *whenever* $first(currPrg(\sigma_2^0)) = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c'\ \textbf{\textit{else}}\ c''$ *and* $[\![e]\!](M_2^0) = \textbf{\textit{tt}}$, *then* $term(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ *and* $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,
14. *whenever* $first(currPrg(\sigma_2^0)) = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c'\ \textbf{\textit{else}}\ c''$ *and* $[\![e]\!](M_2^0) = \textbf{\textit{ff}}$, *then* $term(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ *and* $notBelow(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,
15. *whenever* $first(currPrg(\sigma_1^0)) = \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c'$ *and* $[\![e]\!](M_1^0) = \textbf{\textit{tt}}$, *then* $term(\langle \Delta_1^1, c'; \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ *and* $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,
16. *whenever* $first(currPrg(\sigma_2^0)) = \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c'$ *and* $[\![e]\!](M_2^0) = \textbf{\textit{tt}}$, *then* $term(\langle \Delta_2^1, c'; \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ *and* $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,
17. *whenever* $first(currPrg(\sigma_1^0)) = \textbf{\textit{set pc to}}\ \ell'$, $\ell' \sqsubseteq currPc(\sigma_1^0)$,

then there are i, j such that $\sigma_1^0 \overset{\pi_1}{\rightsquigarrow}^i \sigma_1^i$, $\sigma_2^0 \overset{\pi_2}{\rightsquigarrow}^j \sigma_2^j$, and $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^i, \sigma_2^j)\}$ is a $(\sigma_1^0, \sigma_2^0, i, j, \ell)$-bisimulation.

Proof. Let sec_0 be the policy used to initialize the monitor, $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0\rangle, \mathcal{S}_1^0\rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0\rangle, \mathcal{S}_2^0\rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1\rangle, \mathcal{S}_1^1\rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1\rangle, \mathcal{S}_2^1\rangle$ be four global configurations, τ_1, τ_2 be two traces, $u = currUsr(\sigma_1^1)$ be a user, and $\ell \in \mathcal{L}$ be a label. Furthermore, we assume the following conditions hold:

1. $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,
2. $\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,
3. $\sigma_1^0 \approx_\ell \sigma_2^0$,
4. $\sigma_1^0 \equiv^{cfg} \sigma_2^0$,
5. $C_1^0 = C_2^0$,
6. $\mathcal{S}_1^0 = \mathcal{S}_2^0$,
7. $currPc(\sigma_1^0) \sqsubseteq \ell$ and $currPc(\sigma_2^0) \sqsubseteq \ell$,
8. $currPc(\sigma_1^1) \not\sqsubseteq \ell$ or $currPc(\sigma_2^1) \not\sqsubseteq \ell$,
9. $cl(auth(sec_0, ATK)) \sqsubseteq \ell$,
10. for all users $u' \neq currUsr(\sigma_1^0)$, $\Delta_1^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$ and $\Delta_2^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$,
11. whenever $first(currPrg(\sigma_1^0)) = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$ and $[\![e]\!](M_1^0) = \mathbf{tt}$, then $term(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1\rangle\rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1\rangle\rangle, \ell, u) = \top$,
12. whenever $first(currPrg(\sigma_1^0)) = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$ and $[\![e]\!](M_1^0) = \mathbf{ff}$, then $term(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1\rangle\rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1\rangle\rangle, \ell, u) = \top$,
13. whenever $first(currPrg(\sigma_2^0)) = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$ and $[\![e]\!](M_2^0) = \mathbf{tt}$, then $term(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1\rangle\rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1\rangle\rangle, \ell, u) = \top$,
14. whenever $first(currPrg(\sigma_2^0)) = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$ and $[\![e]\!](M_2^0) = \mathbf{ff}$, then $term(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1\rangle\rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1\rangle\rangle, \ell, u) = \top$,
15. whenever $first(currPrg(\sigma_1^0)) = \textbf{while } e \textbf{ do } c'$ and $[\![e]\!](M_1^0) = \mathbf{tt}$, then $term(\langle \Delta_1^1, c'; \textbf{while } e \textbf{ do } c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1\rangle\rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1\rangle\rangle, \ell, u) = \top$,
16. whenever $first(currPrg(\sigma_2^0)) = \textbf{while } e \textbf{ do } c'$ and $[\![e]\!](M_2^0) = \mathbf{tt}$, then $term(\langle \Delta_2^1, c'; \textbf{while } e \textbf{ do } c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1\rangle\rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1\rangle\rangle, \ell, u) = \top$,
17. whenever $first(currPrg(\sigma_1^0)) = \textbf{set pc to } \ell'$, $\ell' \sqsubseteq currPc(\sigma_1^0)$.

Let $c_1 = currPrg(\sigma_1^0)$, $u_1 = currUsr(\sigma_1^0)$, $c_2 = currPrg(\sigma_2^0)$, and $u_2 = currUsr(\sigma_2^0)$. From (5) and (6), it follows that $\langle u_1, c_1\rangle = \langle u_2, c_2\rangle$. In the following, we denote u_1 and u_2 using u and c_1 and c_2 using c. Furthermore, we denote by n the value such that $C_1^0(n) = \langle currUsr(\sigma_1^0), currPrg(\sigma_1^0)\rangle$. From Lemma D.12 and (17), there are only two cases:

- $first(c_1) = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. We assume that $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''; c_3$, where c_3 can be an empty program (the proof for the other cases is almost identical). From this and the F-IfTrue and F-IfFalse rules, it follows that $currPrg(\sigma_1^1) = [c_1^*; \textbf{set pc to } \Delta_1^0(\mathtt{pc}_u)]; c_3$ and $currPrg(\sigma_2^1) = [c_2^*; \textbf{set pc to } \Delta_2^0(\mathtt{pc}_u)]; c_3$, where $c_1^* \in \{c', c''\}$, and $c_2^* \in \{c', c''\}$. Furthermore, from (3) and (7), it follows that $\Delta_1^0(\mathtt{pc}_u) = \Delta_2^0(\mathtt{pc}_u) = \ell'$. Therefore, $currPrg(\sigma_1^1) = [c_1^*; \textbf{set pc to } \ell']; c_3$ and $currPrg(\sigma_2^1) = [c_2^*; \textbf{set pc to } \ell']; c_3$, where $c_1^* \in \{c', c''\}$, and $c_2^* \in \{c', c''\}$. From this and (11)–(14), it follows that there are $\langle \Delta_1^i, [\varepsilon]; c_3, m_1^i, \langle s_1^i, ctx_1^i\rangle\rangle$ and $\langle \Delta_2^j, [\varepsilon]; c_3, m_2^j, \langle s_2^j, ctx_2^j\rangle\rangle$ such that $\langle \Delta_1^0, [c_1^*; \textbf{set pc to } \ell']; c_3, currMem(\sigma_1^0), \langle s_1^0, ctx_1^0\rangle\rangle \overset{\tau}{\rightsquigarrow}_u^i \langle \Delta_1^i, \varepsilon; c_3, m_1^i, \langle s_1^i, ctx_1^i\rangle\rangle$ and $\langle \Delta_2^0, [c_2^*; \textbf{set pc to } \ell']; c_3, currMem(\sigma_2^0), \langle s_2^0, ctx_2^0\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u^j \langle \Delta_2^j, \varepsilon; c_3, m_2^j, \langle s_2^j, ctx_2^j\rangle\rangle$, where $\Delta_1^i(\mathtt{pc}_u) = \ell'$ and $\Delta_2^i(\mathtt{pc}_u) = \ell'$. From (11)–(14), it follows that during both computations $\mathtt{pc}_u$ is never below ℓ before executing the last **set pc to** ℓ' statement. From this and (8), it follows that $\Delta_1^{i'}(\mathtt{pc}_u) \not\sqsubseteq \ell$ and $\Delta_2^{j'}(\mathtt{pc}_u) \not\sqsubseteq \ell$ for all $1 \leq i' \leq i-1$ and $1 \leq j' \leq j-1$. By repeatedly applying Lemma D.13 and Lemma D.14 and $\Delta_1^{i'}(\mathtt{pc}_u) \not\sqsubseteq \ell$ and $\Delta_2^{j'}(\mathtt{pc}_u) \not\sqsubseteq \ell$ for all $1 \leq i' \leq i-1$ and $1 \leq j' \leq j-1$, we obtain that $\langle \Delta_1^{i'}, c_1^{i'}, m_1^{i'}, \langle s_1^{i'}, ctx_1^{i'}\rangle\rangle \approx_\ell \langle \Delta_2^{j'}, c_2^{j'}, m_2^{j'}, \langle s_2^{j'}, ctx_2^{j'}\rangle\rangle$ for all $1 \leq i' \leq i-1$ and $1 \leq j' \leq j-1$. From this and the fact that in the last step of the execution we set $\mathtt{pc}_u$ to ℓ' in both runs, $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i\rangle\rangle \approx_\ell \langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j\rangle\rangle$. Similarly, by repeatedly applying Lemma D.15, we obtain that $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i\rangle\rangle \equiv^{cfg} \langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j\rangle\rangle$. By repeatedly applying the F-Atomic-Statement rule both to σ_1^0 and σ_2^0, we obtain that $\sigma_1^0 \overset{\tau}{\rightarrow}^i \langle \Delta_1^i, C_1^i, M_1^i, \langle s_1^i, ctx_1^i\rangle, \mathcal{S}_1^0\rangle$ and $\sigma_2^0 \overset{\tau'}{\rightsquigarrow}^i \langle \Delta_2^j, C_2^j, M_2^j, \langle s_2^j, ctx_2^j\rangle, \mathcal{S}_2^0\rangle$, where $C_1^i = C_1^i(1)\cdot\ldots\cdot C_1^i(n-1)\cdot\langle u, [\varepsilon]; c_3\rangle\cdot C_1^i(n+1)\ldots\cdot C_1^i(|C_1^i|)$, $C_2^i = C_2^i(1)\cdot\ldots\cdot C_2^i(n-1)\cdot\langle u, [\varepsilon]; c_3\rangle\cdot C_2^i(n+1)\ldots\cdot C_2^i(|C_2^i|)$, $M_1^i = M_1^i(1)\cdot\ldots\cdot M_1^i(n-1)\cdot\langle u, m_1^i\rangle\cdot M_1^i(n+1)\ldots\cdot M_1^i(|M_1^i|)$, and $M_2^i = M_2^i(1)\cdot\ldots\cdot M_2^i(n-1)\cdot\langle u, m_2^i\rangle\cdot M_2^i(n+1)\ldots\cdot M_2^i(|M_2^i|)$. In the following, let $\sigma_1^i = \langle \Delta_1^i, C_1^i, M_1^i, \langle s_1^i, ctx_1^i\rangle, \mathcal{S}_1^0\rangle$ and $\sigma_2^j = \langle \Delta_2^j, C_2^j, M_2^j, \langle s_2^j, ctx_2^j\rangle, \mathcal{S}_2^0\rangle$. We now show that $R = \{(\sigma_1^0, \sigma_2^0), (\sigma_1^i, \sigma_2^j)\}$ is a $(\sigma_1^0, \sigma_2^0, i, j, \ell)$-bisimulation. We first need to show that for all $\sigma_1' = \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$ and $\sigma_2' = \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$ such that $\sigma_1' R \sigma_2'$, the following conditions hold:

(a) $reach^i(\sigma_1^0, \sigma_1')$ and $reach^j(\sigma_2^0, \sigma_2')$, (b) $\sigma_1' \approx_\ell \sigma_2'$, (c) $\sigma_1' \equiv^{cfg} \sigma_2'$, (d) $C_1' = C_2'$, (e) $\mathcal{S}_1' = \mathcal{S}_2'$, (f) $currPc(\sigma_1') \sqsubseteq \ell$ and $currPc(\sigma_2') \sqsubseteq \ell$. There are two cases:

- $(\sigma_1', \sigma_2') = (\sigma_1^0, \sigma_2^0)$. Then, (a) trivially follows since $reach^k(\sigma, \sigma)$ always holds for all $k > 0$ and both $i, j > 0$. Morever, (b)–(f) directly follow from (3)–(7).
- $(\sigma_1', \sigma_2') = (\sigma_1^i, \sigma_2^j)$. Then, (a) directly follows from $\sigma_1^0 \overset{\tau}{\rightarrow}{}^i \sigma_1^i$ and $\sigma_2^0 \overset{\tau'}{\rightarrow}{}^i \sigma_2^j$. Condition (b) follows from (3), $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i \rangle \rangle \approx_\ell \langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j \rangle \rangle$, and $\Delta_1^i(\mathtt{pc}_u) = \Delta_2^j(\mathtt{pc}_u) = \ell'$. Condition (c) follows from $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i \rangle \rangle \equiv^{cfg} \langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j \rangle \rangle$. Condition (d) follows from (6), $\mathcal{S}_1^i = \mathcal{S}_1^0$, and $\mathcal{S}_2^j = \mathcal{S}_2^0$. Condition (e) follows from (5) and the fact that we applied only the F-AtomicStatement rule, which does not modify the scheduler. Condition (f) follows from (7), $\Delta_1^i(\mathtt{pc}_u) = \Delta_1^0(\mathtt{pc}_u)$, and $\Delta_2^i(\mathtt{pc}_u) = \Delta_2^0(\mathtt{pc}_u)$.

Therefore, the fact that R is a bisimulation directly follows from (i) the fact that (a)–(f) hold for (σ_1^0, σ_2^0) and (σ_1^1, σ_2^1), (ii) assumptions (1), (2), and $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$, and (3) there are no configurations that are reachable in i steps from σ_1^0 and j steps from σ_2^0 other than σ_1^i and σ_2^j.

- $first(c_1) = \textbf{while}\ e\ \textbf{do}\ c'$. The proof of this case is similar to that of $first(c_1) = \textbf{if}\ e\ \textbf{then}\ c'\ \textbf{else}\ c''$.

This completes the proof of our claim. □

Finally, Lemma D.27 states a composition result for bisimulations.

Lemma D.27. *Let R_1 be a $(\sigma_0^0, \sigma_1^0, i, j, \ell)$-bisimulation and R_2 be a $(\rho_0^0, \rho_1^0, x, y, \ell')$-bisimulation. If the following conditions hold:*

1. $(\sigma_0^0, \sigma_1^0) \in R_1$,
2. $(\rho_0^0, \rho_1^0) \in R_1 \cap R_2$,
3. $\ell = \ell'$,
4. $\sigma_0^0 \overset{\tau}{\rightarrow}{}^i \rho_0^0$, *and*
5. $\sigma_1^0 \overset{\tau}{\rightarrow}{}^j \rho_1^0$,

then $R_1 \cup R_2$ is a $(\sigma_0^0, \sigma_1^0, i + x, j + y, \ell)$-bisimulation.

Proof. Let R_1 be a $(\sigma_0^0, \sigma_1^0, i, j, \ell)$-bisimulation and R_2 be a $(\rho_0^0, \rho_1^0, x, y, \ell')$-bisimulation. Assume, for contradiction's sake, that $R_1 \cup R_2$ is not a $(\sigma_0^0, \sigma_1^0, i + x, j + y, \ell)$-bisimulation. This happens iff there is a $(\nu_0, \nu_1) \in R_1 \cup R_2$ that violates one of the following constraints:

1. $\neg reach^{i+x}(\sigma_0^0, \nu_0)$ or $\neg reach^{j+y}(\sigma_1^0, \nu^1)$. Without loss of generality, we assume that $reach^{i+x}(\sigma_0^0, \nu_0)$ does not hold. If $(\nu_0, \nu_1) \in R_1$, then $reach^i(\sigma_0^0, \nu_0)$ holds and $reach^{i+x}(\sigma_0^0, \nu_0)$ follows, leading to a contradiction. If $(\nu_0, \nu_1) \in R_2 \setminus R_1$, then from $\sigma_0^0 \overset{\tau}{\rightarrow}{}^i \rho_0^0$ and $reach^x(\rho_0^0, \nu_0)$, it follows that $reach^{i+x}(\sigma_0^0, \nu_0)$, leading to a contradiction.
2. $\nu_0 \not\approx_\ell \nu_1$. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
3. $\nu_0 \not\equiv^{cfg} \nu_1$. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
4. $C_0 \neq C_1$, where C_0 is the code in ν_0 and C_1 is the code in ν_1. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
5. $\mathcal{S}_0 \neq \mathcal{S}_1$, where $\mathcal{S}_0$ is the scheduler in ν_0 and $\mathcal{S}_1$ is the scheduler in ν_1. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
6. $currPc(\nu_0) \not\sqsubseteq \ell$ or $currPc(\nu_1) \not\sqsubseteq \ell$. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
7. $\nu_0 \overset{\tau_0}{\rightsquigarrow} \nu_0'$, $\nu_1 \overset{\tau_1}{\rightsquigarrow} \nu_1'$, $reach^{i+x}(\sigma_0^0, \nu_0')$, $reach^{j+y}(\sigma_1^0, \nu_1')$, and $currPc(\nu_0') \sqsubseteq \ell$, but $(\nu_0', \nu_1') \notin R_1 \cup R_2$. There are three cases:
   - $(\nu_0, \nu_1) \in R_0$, $reach^i(\sigma_0^0, \nu_0')$, and $reach^j(\sigma_1^0, \nu_1')$. From this and R_1 is a $(\sigma_0^0, \sigma_1^0, i, j, \ell)$-bisimulation, it follows that $(\nu_0', \nu_1') \in R_1$. Hence, $(\nu_0', \nu_1') \in R_1 \cup R_2$, leading to a contradiction.
   - $(\nu_0, \nu_1) \in R_1$, $reach^x(\rho_0^0, \nu_0')$, and $reach^y(\rho_1^0, \nu_1')$. From this and R_2 is a $(\rho_0^0, \rho_1^0, x, y, \ell)$-bisimulation, it follows that $(\nu_0', \nu_1') \in R_2$. Hence, $(\nu_0', \nu_1') \in R_1 \cup R_2$, leading to a contradiction.
   - $(\nu_0, \nu_1) \in R_0$, $reach^{i+x}(\sigma_0^0, \nu_0')$, $reach^{j+y}(\sigma_1^0, \nu_1')$, but $\neg reach^i(\sigma_0^0, \nu_0')$ or $\neg reach^j(\sigma_1^0, \nu_1')$. This happens iff $(\nu_0, \nu_1) = (\rho_0, \rho_1)$. From this, $(\rho_0, \rho_1) \in R_1 \cap R_2$, $\sigma_0^0 \overset{\tau}{\rightarrow}{}^i \rho_0^0$, $\sigma_1^0 \overset{\tau}{\rightarrow}{}^j \rho_1^0$, $reach^{i+x}(\sigma_0^0, \nu_0')$, and $reach^{j+y}(\sigma_1^0, \nu_1')$, it follows that $reach^x(\rho_0^0, \nu_0')$ and $reach^y(\rho_1^0, \nu_1')$. From this and R_2 is a $(\rho_0^0, \rho_1^0, x, y, \ell)$-bisimulation, it follows that $(\nu_0', \nu_1') \in R_2$. Hence, $(\nu_0', \nu_1') \in R_1 \cup R_2$, leading to a contradiction.
8. $\nu_0 \overset{\tau_0}{\rightsquigarrow} \nu_0'$, $\nu_1 \overset{\tau_1}{\rightsquigarrow} \nu_1'$, $reach^{i+x}(\sigma_0^0, \nu_0')$, $reach^{j+y}(\sigma_1^0, \nu_1')$, $currPc(\nu_0') \not\sqsubseteq \ell$, and there are ν_0'' and ν_1'' such that $\nu_0 \overset{\tau_0'}{\rightsquigarrow}{}^* \nu_0''$, $\nu_1 \overset{\tau_1'}{\rightsquigarrow}{}^* \nu_1''$, $reach^{i+x}(\sigma_0^0, \nu_0'')$, $reach^{j+y}(\sigma_1^0, \nu_1'')$, $currPc(\nu_0'') \sqsubseteq \ell$, $currPc(\nu_1'') \sqsubseteq \ell$, and $(\nu_0'', \nu_1'') \notin R_1 \cup R_2$. If $reach^i(\sigma_0^0, \nu_0'')$ and $reach^j(\sigma_1^0, \nu_1'')$, then $(\nu_0'', \nu_1'') \in R_1$ and, therefore, $(\nu_0'', \nu_1'') \in R_1 \cup R_2$, leading to a contradiction. If $reach^x(\rho_0^0, \nu_0'')$ and $reach^y(\rho_1^0, \nu_1'')$, then $(\nu_0'', \nu_1'') \in R_2$ and, therefore, $(\nu_0'', \nu_1'') \in R_1 \cup R_2$, leading to a contradiction. Note that from $(\rho_0, \rho_1) \in R_1 \cap R_2$, it follows that the above cases are the only possible.

Since all cases lead to a contradiction, this completes the proof of our claim. □

D.3.9 Proof of the main result

We are now ready to prove the main result of Section 7.6, namely that our mechanism provides security with respect to an external attacker ATK.

Theorem D.7. *For all programs $C = c_1 \cdot \ldots \cdot c_k \in Com^k_{UID}$, scheduler $\mathcal{S}$, memories $M = m_1 \cdot \ldots \cdot m_k \in Mem^k_{UID}$, and initial runtime state s, whenever $r = \langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}^n \langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $K^{\rightsquigarrow}_{ATK}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{ATK,sec}(M, s) \subseteq K^{\rightsquigarrow}_{ATK}(\langle M, s\rangle, C, \mathcal{S}, trace(r^i))$, where the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$ and $K^{\rightsquigarrow}_{ATK}$ refers to Definition 7.3 with $\rightsquigarrow$ as the underlying evaluation relation.*

Proof. Let $k \in \mathbb{N}$, $C_0 = c_1 \cdot \ldots \cdot c_k \in Com^k_{UID}$ be WhileSql programs, $\mathcal{S}_0$ be a scheduler, $M_0 = m_1 \cdot \ldots \cdot m_k \in Mem^k_{UID}$ be memories, s_0 be a database state, σ_0 be the global state $\langle \Delta_0, C_0, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}_0\rangle$. Let σ_1 be a global state and n be a value in $\mathbb{N}$ such that $r = \sigma_0 \overset{\tau}{\rightsquigarrow}^n \sigma_1$. Assume, for contradiction's sake, that our claim does not hold. Namely, there exists a value $1 \leq i \leq n$ such that $K^{\rightsquigarrow}_{ATK}(\langle M_0, s_0\rangle, C_0, \mathcal{S}_0, trace(r^{i-1})) \cap A_{ATK,sec}(M_0, s_0) \not\subseteq K^{\rightsquigarrow}_{ATK}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, trace(r^i))$, where sec is the security policy in the $(i-1)$-th configuration in r. From this, it follows that there is a state $\langle M_1, s_1\rangle$ such that $\langle M_1, s_1\rangle \in K^{\rightsquigarrow}_{ATK}(\langle M_0, s_0\rangle, C_0, \mathcal{S}_0, trace(r^{i-1})) \cap A_{ATK,sec}(M_0, s_0)$ and $\langle M_1, s_1\rangle \notin K^{\rightsquigarrow}_{ATK}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, trace(r^i))$. From $\langle M_1, s_1\rangle \in K^{\rightsquigarrow}_{ATK}(\langle M_0, s_0\rangle, C_0, \mathcal{S}_0, trace(r^{i-1}))$, it follows that $s_0 \approx_{ATK} s_1$, $M_0 \approx_{ATK} M_1$, and for all ctx', Δ', τ', C', M', s', $\mathcal{S}'$ such that $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \overset{\tau'}{\rightsquigarrow}^* \langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, $trace(r^{i-1}) \sim_{ATK} \tau'$. From $\langle M_1, s_1\rangle \in A_{ATK,sec}(M, s)$, it follows that $s_1 \approx_{sec,ATK} s$ and $M_1 \approx_{ATK} M$. Finally, from $\langle M_1, s_1\rangle \notin K^{\rightsquigarrow}_{ATK}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, trace(r^i))$, it follows that $s_0 \not\approx_{ATK} s_1$, $M_0 \not\approx_{ATK} M_1$, or there are ctx', Δ', τ', C', M', s', $\mathcal{S}'$ such that $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \overset{\tau'}{\rightsquigarrow}^* \langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ and $trace(r^i) \not\sim_{ATK} \tau'$. Note that only the last case is interesting (since $s_0 \not\approx_{ATK} s_1$ and $M_0 \not\approx_{ATK} M_1$ immediately contradict $\langle M_1, s_1\rangle \in K^{\rightsquigarrow}_{ATK}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, trace(r^{i-1}))$). Therefore, the following conditions hold:

1. $s_1 \approx_{ATK} s_0$,
2. $s_1 \approx_{sec,ATK} s_0$,
3. $M_1 \approx_{ATK} M_0$,
4. there are j, ctx^j_1, Δ^j_1, τ^j_1, C^j_1, M^j_1, s^j_1, $\mathcal{S}^j_1$ such that:
   (a) $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \overset{\tau'}{\rightsquigarrow}^j \langle \Delta^j_1, C^j_1, M^j_1, \langle s^j_1, ctx^j_1\rangle, \mathcal{S}^j_1\rangle$,
   (b) $trace(r^{i-1}) \sim_{ATK} \tau'$, and
   (c) $trace(r^i) \not\sim_{ATK} \tau'$.

In the following, let sec_0 be the policy in the state s_0 (which, from $s_1 \approx_{ATK} s_0$, is the same as in s_1), sec be the policy in the $(i-1)$-th configuration in r, and $Vars_{ATK}$ be the variables occurring in ATK's program. Furthermore, let ℓ be the label $cl(auth(sec_0, ATK) \cup auth(sec, ATK) \cup \bigcup_{x \in Vars_{ATK}} MEM_x)$ (observe that $cl(auth(sec_0, ATK)) \sqsubseteq \ell$ holds), $\sigma^{i-1}_0 = \langle \Delta^{i-1}_0, C^{i-1}_0, M^{i-1}_0, \langle s^{i-1}_0, ctx^{i-1}_0\rangle, \mathcal{S}^{i-1}_0\rangle$ be $(i-1)$-th configuration in r, and σ^{j-1}_1 be the configuration $\langle \Delta^{j-1}_1, C^{j-1}_1, M^{j-1}_1, \langle s^{j-1}_1, ctx^{j-1}_1\rangle, \mathcal{S}^{j-1}_1\rangle$. From (4.b) and (4.c), it follows that the only interesting cases are those for which $trace(r^i)\restriction_{ATK} = trace(r^{i-1})\restriction_{ATK} \cdot obs_0$ (since if $trace(r^i)\restriction_{ATK} = trace(r^{i-1})\restriction_{ATK}$, then (4.b) and (4.c) are contradictory statements), where obs_1 is an observation. Therefore, there is a non-empty trace π_1 such that $trace(r^{i-1})\restriction_{ATK} = \pi_1$ and $trace(r^i)\restriction_{ATK} = \pi_1 \cdot obs_1$. Let σ^i_0 and σ^{i-1}_0 be the last global states in r^i and r^{i-1}. From $trace(r^i)\restriction_{ATK} = trace(r^{i-1})\restriction_{ATK} \cdot obs_0$, it follows that $\sigma^{i-1}_0 \overset{obs_0}{\rightsquigarrow} \sigma^i_0$. From $trace(r^{i-1}) \sim_{ATK} \tau'$, $trace(r^i) \not\sim_{ATK} \tau'$, and $trace(r^i)\restriction_u = trace(r^{i-1})\restriction_u \cdot obs_0$, it follows that $\tau'\restriction_{ATK} = \pi_1 \cdot obs_1 \cdot \pi_4$, where obs_1 is an observation different from obs_0 (it this is not the case, this would contradict (4.b) and (4.c) since this would imply $trace(r^i) \sim_{ATK} \tau'$). Without loss of generality, we assume that $\pi_4 = \epsilon$ and that obs_1 is produced in the last step of $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \overset{\tau'}{\rightsquigarrow}^j \langle \Delta^j_1, C^j_1, M^j_1, \langle s^j_1, ctx^j_1\rangle, \mathcal{S}^j_1\rangle$, i.e., $\sigma^{j-1}_1 \overset{obs_1}{\rightsquigarrow} \sigma^j_1$. We claim that that (1) $\sigma^{i-1}_0 \approx_\ell \sigma^{j-1}_1$, (2) $C^{i-1}_0 = C^{j-1}_1$, (3) $\mathcal{S}^{i-1}_0 = \mathcal{S}^{j-1}_1$, (4) $s^{i-1}_0 \equiv^{cfg} s^{j-1}_1$. Furthermore, we also claim that $currPc(\sigma^{i-1}_0) \sqsubseteq \ell$ and $L_{\mathcal{U}}(s^{i-1}_0, ATK) \sqsubseteq \ell$. From (1) and $currPc(\sigma^{i-1}_0) \sqsubseteq \ell$, it follows that $currPc(\sigma^{j-1}_1) \sqsubseteq \ell$. From (4) and $L_{\mathcal{U}}(s^{i-1}_0, ATK) \sqsubseteq \ell$, it follows that $L_{\mathcal{U}}(s^{j-1}_1, ATK) \sqsubseteq \ell$. From this and Lemma D.16, it follows that $obs_0 = obs_1$, leading to a contradiction.

Below, we prove our claims together with other intermediate facts.

Fact 1. We now prove that $\sigma^0_0 \approx_\ell \sigma^0_1$, where $\sigma^0_0 = \langle \Delta_0, C_0, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}_0\rangle$, $\sigma^0_1 = \langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle$, $s_0 = \langle db_0, U_0, sec_0, T_0, V_0\rangle$, and $s_1 = \langle db_1, U_1, sec_1, T_1, V_1\rangle$. To do so, we need to show that:

- For all queries $q \in RC$ such that $L_{\mathcal{Q}}(\Delta_0, q) \sqsubseteq \ell$, $[q]^{db_0} = [q]^{db_1}$. From $\ell = cl(auth(sec_0, ATK) \cup auth(sec, ATK) \cup \bigcup_{x \in Vars_{ATK}} MEM_x)$ and q is a query (i.e., it does not refer to MEM_x for any x), it follows that $L_{\mathcal{Q}}(\Delta_0, q) \sqsubseteq cl(auth(sec_0, ATK) \cup auth(sec, ATK))$. From $L_{\mathcal{Q}}$'s definition, it follows that $L_{\mathcal{Q}}(\Delta_0, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta_0(q')$. From this and Δ_0's

definition, it follows that $\Delta_0(q) = cl(q)$ for all $q \in RC^{pred}$. From this and $L_{\mathcal{Q}}(\Delta_0, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta_0(q')$, it follows that $L_{\mathcal{Q}}(\Delta_0, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} cl(q')$. From this and $L_{\mathcal{Q}}(\Delta_0, q) \sqsubseteq cl(auth(sec_0, ATK) \cup auth(sec, ATK))$, it follows $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} cl(q') \sqsubseteq cl(auth(sec_0, ATK) \cup auth(sec, ATK))$. Furthermore, from the definition of $supp_{D,\Gamma}(q)$, it follows that $cl(q) \sqsubseteq \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} cl(q')$. Therefore, the tables and views in $auth(sec_0, ATK) \cup auth(sec, ATK)$ determine the values of the queries in $supp_{D,\Gamma}(q)$, which in turn determine the value of q, i.e., $cl(q) \sqsubseteq cl(auth(sec_0, ATK) \cup auth(sec, ATK))$. From $s_1 \approx_{ATK} s_0$, it follows that the content of all tables and views in $auth(sec_0, ATK)$ is the same in s_0 and s_1. From $s_1 \approx_{sec,ATK} s_0$, it follows that the content of all tables and views in $auth(sec, ATK)$ is the same in s_0 and s_1. From this and $cl(q) \sqsubseteq cl(auth(sec_0, ATK) \cup auth(sec, ATK))$, it follows that $[q]^{db_0} = [q]^{db_1}$.

- For all variables $x \in Vars$ such that $\Delta_0(x) \sqsubseteq \ell$, $[\![M_0]\!](x) = [\![M_1]\!](x)$. From $\ell = cl(auth(sec_0, ATK) \cup auth(sec, ATK) \cup \bigcup_{x \in Vars_{ATK}} MEM_x)$, it follows that $\Delta_0(x) \sqsubseteq cl(auth(sec_0, ATK) \cup auth(sec, ATK) \cup \bigcup_{x \in Vars_{ATK}} MEM_x)$. From this and $\Delta_0(x) = \top$ if $x \notin Vars_{ATK}$ and $\Delta_0(x) = MEM_x$ otherwise, it follows that $\Delta_0(x) = MEM_x$. From this, it follows that $x \in Vars_{ATK}$. From this and $s_1 \approx_{ATK} s_0$, it follows that $[\![M_0]\!](x) = [\![M_1]\!](x)$.

These facts together with the fact that the monitor state is the same in σ_0^0 and σ_1^0, leads to $\sigma_0^0 \approx_\ell \sigma_1^0$.

Fact 2. We now prove that $s_0 \equiv^{cfg} s_1$. Let $s_0 = \langle db_0, U_0, S_0, T_0, V_0 \rangle$ and $s_1 = \langle db_1, U_1, S_1, T_1, V_1 \rangle$. From $s_1 \approx_{ATK} s_0$ and $\approx_{ATK}$'s definition, it follows that $U_0 = U_1$, $S_0 = S_1$, $T_0 = T_1$, and $V_0 = V_1$. From this and $\equiv^{cfg}$'s definition, it follows that $s_0 \equiv^{cfg} s_1$.

Fact 3. We now show that $currPc(\sigma_0^{i-1}) \sqsubseteq \ell$ and $L_{\mathcal{U}}(s_0^{i-1}, ATK) \sqsubseteq \ell$. From $\ell = cl(auth(sec_0, ATK) \cup auth(sec, ATK) \cup \bigcup_{x \in Vars_{ATK}} MEM_x)$ and sec is the policy in σ_0^{i-1}, it immediately follows that $L_{\mathcal{U}}(s_0^{i-1}, ATK) \sqsubseteq \ell$. Assume, for contradiction's sake, that $currPc(\sigma_0^{i-1}) \not\sqsubseteq \ell$. From $\sigma_0^{i-1} \overset{obs_0}{\rightsquigarrow} \sigma_0^i$ and $obs_0 \neq \epsilon$, it follows that the executed rule produced an event. In the following, let $currUsr(\sigma_0^{i-1}) = u$. Observe that $\Delta_0^{i-1}(\mathtt{pc}_u) = currPc(\sigma_0^{i-1})$. There are two cases:

- Rule F-Eval-Step. From the rule, it follows that $\langle \Delta_0^{i-1}, c_0^{i-1}, m_0^{i-1}, \langle s_0^{i-1}, ctx_0^{i-1} \rangle \rangle \overset{obs_0}{\rightsquigarrow}_u \langle \Delta_0^i, c_0^i, m_0^i, \langle s_0^i, ctx_0^i \rangle \rangle$ (which we denote r). From this and Lemma D.11, it follows that $\Delta_0^{i-1}(deps(obs_0, r)) \sqcup \Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, user(obs_0))$. From this, it follows that $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, user(obs_0))$. Furthermore, since $trace(r^i)\restriction_{ATK} = trace(r^{i-1})\restriction_{ATK} \cdot obs_0$, it follows that $user(obs_0) = ATK$. From this and $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, user(obs_0))$, it follows that $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, ATK)$. From this and $L_{\mathcal{U}}(s_0^{i-1}, ATK) \sqsubseteq \ell$, it follows that $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
- Rule F-Atomic-Statement. The proof is similar to that of F-Eval-Step.

Since all cases lead to a contradiction, this completes the proof of Fact 3.

Fact 4. We prove that $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$. From $\sigma_0^0 \overset{\tau_0}{\rightsquigarrow}^* \sigma_0^{i-1} \overset{obs_0}{\rightsquigarrow} \sigma_0^i$, $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow}^* \sigma_1^{j-1} \overset{obs_1}{\rightsquigarrow} \sigma_1^j$, $\tau_0\restriction_{ATK} = \tau_1\restriction_{ATK}$, all configuration changes are associated with public events, and the code produced by the expansion process either terminates or gets stuck, it follows that the configuration has been modified in the same way in $\sigma_0^0 \overset{\tau_0}{\rightsquigarrow}^* \sigma_0^{i-1}$ and $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow}^* \sigma_1^{j-1}$. Therefore, $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$.

Fact 5. We now prove that $currPc(\sigma_1^{j-1}) \sqsubseteq \ell$. From $\sigma_1^{j-1} \overset{obs_1}{\rightsquigarrow} \sigma_1^j$, there are two cases:

1. We applied the F-Out rule. From the rule, it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq L_{\mathcal{U}}(s_1^{j-1}, u')$. Furthermore, since obs_1 is visible to ATK, it follows that $u' = ATK$. Hence, $currPc(\sigma_1^{j-1}) \sqsubseteq L_{\mathcal{U}}(s_1^{j-1}, ATK)$. From this and $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$ (Fact 4), it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, ATK)$. From this and $L_{\mathcal{U}}(s_0^{i-1}, ATK) \sqsubseteq \ell$ (Fact 3), it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq \ell$
2. We applied the F-DbOut rule. The proof of this case is similar to that of the F-Out rule.

Fact 6. We now prove that (1) $\sigma_0^{i-1} \approx_\ell \sigma_1^{j-1}$, (2) $C_0^{i-1} = C_1^{j-1}$, and (3) $\mathcal{S}_0^{i-1} = \mathcal{S}_1^{j-1}$. From Facts 1 and 2, it follows that initially $\sigma_0^0 \approx_\ell \sigma_1^0$, $C_0^0 = C_1^0$, $\mathcal{S}_0^0 = \mathcal{S}_1^0$, and $s_0^0 \equiv^{cfg} s_1^0$. Furthermore, $currPc(\sigma_0^0) = currPc(\sigma_1^0) = \bot$ given that both runs start from the initial monitor state Δ_0. Therefore, we can repeatedly apply Lemmas D.25 and D.26 (depending on whether we are in a low or high context given ℓ) to construct bisimulations among states in the two runs and use Lemma D.27 to compose the various bisimulations in a unique bisimulation R. We remark that during the construction of the bisimulation we can always apply either Lemma D.25 or Lemma D.26. In particular, $currPc(\sigma_0^{i-1}) \sqsubseteq \ell$ (Fact 3) and $currPc(\sigma_1^{j-1}) \sqsubseteq \ell$ (Fact 5) ensure that the execution of branching statements leading to high contexts with respect to ℓ always terminates before σ_0^{j-1} and σ_1^{j-1}. Finally, observe that $\{(\sigma_0^0, \sigma_0^1), (\sigma_0^{i-1}, \sigma_1^{j-1})\} \subseteq R$ by construction. From this, $\sigma_0^{i-1} \approx_\ell \sigma_1^{j-1}$, $C_0^{i-1} = C_1^{j-1}$, and $\mathcal{S}_0^{i-1} = \mathcal{S}_1^{j-1}$ directly follow. □

Bibliography

[1] 1000 Genomes – A Deep Catalog of Human Genetic Variation. Online at http://www.internationalgenome.org/ (accessed June 2017).

[2] 23andMe – DNA Genetic Testing and Analysis. Online at https://www.23andme.com/ (accessed June 2017).

[3] Linq: .net language-integrated query, *Microsoft MSDN Library*. Online at https://msdn.microsoft.com/en-us/library/bb308959.aspx (accessed June 2017).

[4] Manage trigger security, *Microsoft MSDN Library*. online at http://msdn.microsoft.com/en-us/library/ms191134.aspx/ (accessed June 2017).

[5] openSNP. Online at http://opensnp.org/ (accessed June 2017).

[6] ProbLog – Probabilistic Programming. Online at http://dtai.cs.kuleuven.be/problog/index.html (accessed June 2017).

[7] The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at https://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm (accessed June 2017), 1996.

[8] New Security Features in Sybase Adaptive Server Enterprise. *Sybase Technical White Paper*, 2003.

[9] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 254–263. ACM, 1998.

[10] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[11] Nabil R. Adam and John C. Worthmann. Security-control methods for statistical databases: a comparative study. *ACM Computing Surveys*, 21(4):515–556, 1989.

[12] Foto Afrati, Rada Chirkova, Manolis Gergatsoulis, and Vassia Pavlaki. View selection for real conjunctive queries. *Acta Informatica*, 44(5):289–321, 2007.

[13] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *Proceedings of the 21st International Conference on Data Engineering*, pages 1013–1022. IEEE, 2005.

[14] K. Ahmed, A. A. Emran, T. Jesmin, R. F. Mukti, M. Z. Rahman, and F. Ahmed. Early detection of lung cancer risk using data mining. *Asian Pacific Journal of Cancer Prevention*, 14(1):595–598, 2013.

[15] Alessandro Aldini. Probabilistic information flow in a process algebra. In *Proceedings of the 12th International Conference on Concurrency Theory*, pages 152–168. Springer, 2001.

[16] Mário Alvim, Miguel Andrés, and Catuscia Palamidessi. Probabilistic information flow. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*, pages 314–321. IEEE, 2010.

[17] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, pages 569–583, 2015.

[18] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 25th IEEE Symposium on Computer Security Foundations*, pages 308–322. IEEE, 2012.

[19] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348. Springer, 2008.

[20] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, pages 207–221. IEEE, 2007.

[21] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 22nd IEEE Symposium on Computer Security Foundations*, pages 43–59. IEEE, 2009.

[22] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 113–124. ACM, 2009.

[23] Musard Balliu. A logic for information flow analysis of distributed programs. In *Proceedings of the 18th Nordic Conference on Secure IT Systems*, pages 84–99. Springer-Verlag, 2013.

[24] Musard Balliu, Benjamin Liebe, Daniel Schoepe, and Andrei Sabelfeld. Jslinq: Building secure applications across tiers. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, pages 307–318. ACM, 2016.

[25] Vince Bárány, Balder Ten Cate, and Martin Otto. Queries with guarded negation. In *Proceedings of the 38th International Conference on Very Large Data Bases*, pages 1328–1339. VLDB Endowment, 2012.

[26] Gabriel Bender, Lucja Kot, and Johannes Gehrke. Explainable security for relational databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, pages 1411–1422. ACM, 2014.

[27] Gabriel Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 869–880, 2013.

[28] Joachim Biskup. For unknown secrecies refusal is better than lying. *Data & Knowledge Engineering*, 33(1):1–23, 2000.

[29] Joachim Biskup and Piero A. Bonatti. Lying versus refusal for known potential secrets. *Data & Knowledge Engineering*, 38(2):199–222, 2001.

[30] Joachim Biskup and Piero A Bonatti. Controlled query evaluation for known policies by combining lying and refusal. *Annals of Mathematics and Artificial Intelligence*, 40(1-2):37–62, 2004.

[31] Joachim Biskup, Christian Gogolin, Jens Seiler, and Torben Weibert. Requirements and protocols for inference-proof interactions in information systems. In *Proceedings of the 14th European Symposium on Research in Computer Security*, pages 285–302. Springer, 2009.

[32] Joachim Biskup, Jens Seiler, and Torben Weibert. Controlled query evaluation and inference-free view updates. In *Proceedings of the 23rd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 1–16. Springer, 2009.

[33] Joachim Biskup, Cornelia Tadros, and Lena Wiese. Towards controlled query evaluation for incomplete first-order databases. In *Proceedings of the 6th International Symposium on Foundations of Information and Knowledge Systems*, pages 230–247. Springer, 2010.

[34] Joachim Biskup and Torben Weibert. Confidentiality policies for controlled query evaluation. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 1–13. Springer, 2007.

[35] Joachim Biskup and Torben Weibert. Keeping secrets in incomplete databases. *International Journal of Information Security*, 7(3):199–217, 2008.

[36] Piero A. Bonatti, Sarit Kraus, and V. S. Subrahmanian. Foundations of secure deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):406–422, 1995.

[37] Egon Börger, Erich Grädel, and Yuri Gurevich. *The classical decision problem*. Springer Verlag, 2001.

[38] Bernadette Bouchon-Meunier, Giulianella Coletti, and Christophe Marsala. Independence and possibilistic conditioning. *Annals of Mathematics and Artificial Intelligence*, 35(1):107–123, 2002.

[39] Niklas Broberg, Bart van Delft, and David Sands. The anatomy and facets of dynamic policies. In *Proceedings of the 28th IEEE Symposium on Computer Security Foundations*, pages 122–136. IEEE, 2015.

[40] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Transactions on Knowledge and Data Engineering*, 12(6):900–919, 2000.

[41] Kristy Browder and Mary Ann Davidson. The virtual private database in Oracle9iR2. *Oracle Technical White Paper, Oracle Corporation*, 500, 2002.

[42] Terence A Brown. *Genomes, 3rd edition*. Garland Science, 2006.

[43] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer Science & Business Media, 1990.

[44] Yu Chen and Wesley W Chu. Database security protection via inference detection. In *International Conference on Intelligence and Security Informatics*, pages 452–458. Springer, 2006.

[45] Yu Chen and Wesley W Chu. Protection of database security via collaborative inference detection. *IEEE Transactions on Knowledge and Data Engineering*, 20(8):1013–1027, 2008.

[46] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 403–416. ACM, 2013.

[47] Francis Y. Chin and Gultekin Ozsoyoglu. Auditing and inference control in statistical databases. *IEEE Transactions on Software Engineering*, 8(6):574–582, 1982.

[48] Rada Chirkova. Query containment. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2249–2253. Springer US, Boston, MA, 2009.

[49] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 105–118. USENIX Association, 2010.

[50] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium*, pages 1:1–1:16. USENIX Association, 2007.

[51] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[52] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, pages 31–45. IEEE, 2005.

[53] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, 2009.

[54] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In *Proceedings of the 2003 International Conference on Rewriting Techniques and Applications*. Springer, 2003.

[55] Edgar F. Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972.

[56] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, pages 266–296. Springer-Verlag, 2007.

[57] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 269–282. ACM, 2009.

[58] Ernesto Damiani, Majirus Fansi, Alban Gabillon, and Stefania Marrara. A general approach to securely querying XML. *Computer standards & interfaces*, 30(6):379–389, 2008.

[59] Benjamin Davis and Hao Chen. DBTaint: cross-application information flow tracking via databases. In *Proceedings of the 2010 USENIX Conference on Web Application Development*. USENIX Association, 2010.

[60] Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

[61] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2468–2473. Morgan Kaufmann, 2007.

[62] Dorothy E. Denning. Secure statistical databases with random sample queries. *ACM Transactions on Database Systems*, 5(3):291–315, 1980.

[63] Dorothy E. Denning and Teresa F. Lunt. A multilevel relational data model. In *Proceedings of the 8th IEEE Symposium on Security and Privacy*, pages 220–220. IEEE, 1987.

[64] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.

[65] David Dobkin, Anita K. Jones, and Richard J. Lipton. Secure databases: protection against user influence. *ACM Transactions on Database Systems*, 4(1):97–106, 1979.

[66] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.

[67] Josep Domingo-Ferrer. *Inference control in statistical databases: From theory to practice*, volume 2316. Springer, 2002.

[68] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*, pages 1–12. Springer, 2006.

[69] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[70] Hamid Ebadi, David Sands, and Gerardo Schneider. Differential privacy: Now it's getting personal. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 69–81. ACM, 2015.

[71] Alexandre Evfimievski, Ronald Fagin, and David Woodruff. Epistemic privacy. *Journal of ACM*, 58(1):2:1–2:45, 2010.

[72] Csilla Farkas and Sushil Jajodia. The inference problem: a survey. *ACM SIGKDD Explorations Newsletter*, 4(2):6–11, 2002.

[73] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(03):358–401, 2015.

[74] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering*, pages 196–205. IEEE, 2005.

[75] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2012.

[76] Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic inference for probabilistic programs. In *Proceedings of the 28th International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

[77] Lise Getoor and Ben Taskar. *Introduction to statistical relational learning*. MIT press, 2007.

[78] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197. ACM, 2004.

[79] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 47–60, 2012.

[80] Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. Tabular: A schema-driven probabilistic programming language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 321–334. ACM, 2014.

[81] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the Conference on The Future of Software Engineering*, pages 167–181. ACM, 2014.

[82] Bernardo Cuenca Grau, Evgeny Kharlamov, Egor V. Kostylev, and Dmitriy Zheleznyakov. Controlled query evaluation over owl 2 rl ontologies. In *Proceedings of the 12th International Semantic Web Conference*, pages 49–65. Springer, 2013.

[83] Patricia P Griffiths and Bradford W Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3), 1976.

[84] Marco Guarnieri and David Basin. Optimal security-aware query processing. In *Proceedings of the 40th International Conference on Very Large Data Bases*, pages 1307–1318. VLDB Endowment, 2014.

[85] Marco Guarnieri, Srdjan Marinovic, and David Basin. Securing databases from probabilistic inference — prototype. Online at `http://www.infsec.ethz.ch/research/projects/FDAC.html`.

[86] Marco Guarnieri, Srdjan Marinovic, and David Basin. Strong and Provably Secure Database Access Control — Prototype and Maude models. Online at `http://www.infsec.ethz.ch/research/projects/FDAC.html`.

[87] Marco Guarnieri, Srdjan Marinovic, and David Basin. Strong and provably secure database access control. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, pages 163–178. IEEE, 2016. **© 2016 IEEE. Reprinted with permission.**

[88] Marco Guarnieri, Srdjan Marinovic, and David Basin. Securing databases from probabilistic inference. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium*, pages 343–359. IEEE, 2017. **© 2017 IEEE. Reprinted with permission.**

[89] Theodore Hailperin. Probability logic. *Notre Dame Journal of Formal Logic*, 25(3):198–212, 1984.

[90] Raju Halder and Agostino Cortesi. Fine grained access control for relational databases by abstract interpretation. In *Proceedings of the 2010 International Conference on Software and Data Technologies*, pages 235–249. Springer, 2010.

[91] John Hale and Sujeet Shenoi. Catalytic inference analysis: Detecting inference threats due to knowledge discovery. In *Proceedings of the 18th IEEE Symposium on Security and Privacy*, pages 188–199. IEEE, 1997.

[92] Jianming He, Wesley W Chu, and Zhenyu Victor Liu. Inferring privacy information from social networks. In *International Conference on Intelligence and Security Informatics*, pages 154–165. Springer, 2006.

[93] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Proceedings of the 2011 Marktoberdorf Summer School*, 2011.

[94] Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic updating of information-flow policies. In *Proceedings of the 2005 International Workshop on Foundations of Computer Security*, pages 7–18, June 2005.

[95] Thomas H. Hinke, Harry S. Delugach, and Randall P. Wolf. Protecting databases from inference attacks. *Computers & Security*, 16(8):687–708, 1997.

[96] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, pages 40–52. ACM, 2004.

[97] Mathias Humbert, Erman Ayday, Jean-Pierre Hubaux, and Amalio Telenti. Addressing the concerns of the lacks family: Quantification of kin genomic privacy. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*, pages 1141–1152. ACM, 2013.

[98] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 79–90. ACM, 2006.

[99] Sushil Jajodia and Ravi Sandhu. Polyinstantiation integrity in multilevel relations. In *Proceedings of the 11th IEEE Symposium on Security and Privacy*, pages 104–115. IEEE, 1990.

[100] Somesh Jha, Ninghui Li, Mahesh Tripunitara, Qihua Wang, and William Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4):242–255, 2008.

[101] Govind Kabra, Ravishankar Ramamurthy, and S. Sudarshan. Redundancy and information leakage in fine-grained access control. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM, 2006.

[102] Vasilios Katos, Dimitris Vrakas, and Panagiotis Katsaros. A framework for access control with inference constraints. In *Proceedings of 35th IEEE Annual Conference on Computer Software and Applications*, pages 289–297. IEEE, 2011.

[103] Krishnaram Kenthapadi, Nina Mishra, and Kobbi Nissim. Simulatable auditing. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 118–127. ACM, 2005.

[104] Robert J. Klein, Caroline Zeiss, Emily Y. Chew, Jen-Yue Tsai, Richard S. Sackler, Chad Haynes, Alice K. Henning, John Paul SanGiovanni, Shrikant M. Mane, Susan T. Mayne, et al. Complement factor h polymorphism in age-related macular degeneration. *Science*, 308(5720):385–389, 2005.

[105] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[106] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 286–296. ACM, 2007.

[107] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Query-based data pricing. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 167–178. ACM, 2012.

[108] Johan Kwisthout, Hans Bodlaender, and Linda van der Gaag. The necessity of bounded treewidth for efficient inference in bayesian networks. In *Proceedings of the 19th European Conference on Artificial Intelligence*, volume 215, pages 237–242, 2010.

[109] Steffen L Lauritzen and Nuala A Sheehan. Graphical models for genetic analyses. *Statistical Science*, 18(4):489–514, 2003.

[110] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 108–119. VLDB Endowment, 2004.

[111] Peng Li and Steve Zdancewic. Practical information flow control in web-based information systems. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, pages 2–15. IEEE, 2005.

[112] Leonid Libkin. Incomplete information and certain answers in general data models. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 59–70. ACM, 2011.

[113] Leonid Libkin. *Elements of finite model theory*. Springer Science & Business Media, 2013.

[114] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334. ACM, 2009.

[115] Gavin Lowe. Quantifying information flow. In *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, pages 18–31. IEEE, 2002.

[116] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, 21(4):463–532, 2013.

[117] Wesley Mathew, Ruben Raposo, and Bruno Martins. Predicting future locations with hidden markov models. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 911–918. ACM, 2012.

[118] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30. ACM, 2009.

[119] Nancy R. Mead and Ted Stehney. Security quality requirements engineering (square) methodology. In *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems—Building Trustworthy Applications*, pages 1–7. ACM, 2005.

[120] Matthew Morgenstern. Security and inference in multilevel database and knowledge-base systems. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of data*, pages 357–373. ACM, 1987.

[121] Matthew Morgenstern. Controlling logical inference in multilevel database systems. In *Proceedings of the 9th IEEE Symposium on Security and Privacy*, pages 245–255. IEEE, 1988.

[122] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142. ACM, 1997.

[123] Shubha U. Nabar, Bhaskara Marthi, Krishnaram Kenthapadi, Nina Mishra, and Rajeev Motwani. Towards robustness in query auditing. In *Proceedings of the 32nd international Conference on Very Large Data Bases*, pages 151–162. VLDB Endowment, 2006.

[124] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Transactions on Database Systems*, 35(3):21, 2010.

[125] Said Oulmakhzoune, Nora Cuppens-Boulahia, Frédéric Cuppens, and Stephane Morucci. fQuery: SPARQL query rewriting to enforce data confidentiality. In *Proceedings of the 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy*, pages 146–161. Springer, 2010.

[126] Aris Pagourtzis and Stathis Zachos. The complexity of counting functions with easy decision version. In *International Symposium on Mathematical Foundations of Computer Science*, pages 741–752. Springer, 2006.

[127] Daniel Pasailă. Conjunctive queries determinacy and rewriting. In *Proceedings of the 14th International Conference on Database Theory*, pages 220–231. ACM, 2011.

[128] Xiaolei Qian, Mark E. Stickel, Peter D. Karp, Teresa F. Lunt, and Thomas D. Garvey. Detection and elimination of inference channels in multilevel relational database systems. In *Proceedings of the 14th IEEE Symposium on Security and Privacy*, pages 196–205. IEEE, 1993.

[129] Vibhor Rastogi, Dan Suciu, and Evan Welbourne. Access control over uncertain data. In *Proceedings of the 34th International Conference of Very Large Data Bases*, pages 821–832. VLDB Endowment, 2008.

[130] General Data Protection Regulation. Regulation (EU) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46. *Official Journal of the European Union (OJ)*, 59:1–88, 2016.

[131] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of data*, pages 551–562. ACM, 2004.

[132] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, volume 10, pages 297–312. USENIX Association, 2010.

[133] John Rushby. Noninterference, transitivity, and channel-control security policies. SRI International, Computer Science Laboratory, Technical Report CSL-92-02 (http://www.csl.sri.com/papers/csl-92-2/), 1992.

[134] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 186–199, 2010.

[135] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1), 2003.

[136] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, pages 200–214. IEEE, 2000.

[137] Pierangela Samarati. Recursive revoke. In *Encyclopedia of Cryptography and Security*, pages 1035–1037. Springer, 2011.

[138] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design: Tutorial Lectures*, pages 137–196. Springer Berlin Heidelberg, 2001.

[139] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122. ACM, 2014.

[140] Ravi Sandhu and Fang Chen. The multilevel relational (MLR) data model. *ACM Transactions on Information and System Security*, 1(1), 1998.

[141] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. Selinq: Tracking information across application-database boundaries. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 25–38. ACM, 2014.

[142] Daniel Schoepe and Andrei Sabelfeld. Understanding and enforcing opacity. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, pages 539–553. IEEE, 2015.

[143] David Schultz and Barbara Liskov. IFDB: decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 43–56. ACM, 2013.

[144] Chung-chieh Shan and Norman Ramsey. Exact bayesian inference by symbolic disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 130–144. ACM, 2017.

[145] Jie Shi, Hong Zhu, Ge Fu, and Tao Jiang. On the Soundness Property for SQL Queries of Fine-grained Access Control in DBMSs. In *8th IEEE/ACIS International Conference on Computer and Information Science*, pages 469–474, 2009.

[146] George L. Sicherman, Wiebren De Jonge, and Reind P. Van de Riet. Answering queries without revealing secrets. *ACM Transactions on Database Systems*, 8(1):41–59, 1983.

[147] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[148] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 46–59. ACM, 2017.

[149] Kenneth Smith and Marianne Winslett. Multilevel secure rules: Integrating the multilevel secure and active data models. In *Database Security VI: Status and Prospects*. North-Holland, 1993.

[150] Michael Stonebraker and Eugene Wong. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 Annual Conference - Volume 1*, pages 180–186. ACM, 1974.

[151] Tzong-An Su and Gultekin Ozsoyoglu. Data dependencies and inference control in multilevel relational database systems. In *Proceedings of the 7th IEEE Symposium on Security and Privacy*, pages 202–211. IEEE, 1987.

[152] Tzong-An Su and Gultekin Ozsoyoglu. Controlling FD and MVD inferences in multilevel relational database systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):474–485, 1991.

[153] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[154] Charles Sutton. GRMM: GRaphical Models in Mallet. Online at http://mallet.cs.umass.edu/grmm/ (accessed June 2017).

[155] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, pages 369–383. IEEE, 2008.

[156] Symantec. State of Privacy Report 2015. 2015.

[157] Bhavani Thuraisingham. Security checking in relational database management systems augmented with inference engines. *Computers & Security*, 6(6):479–492, 1987.

[158] Tyrone S. Toland, Csilla Farkas, and Caroline M. Eastman. The inference problem: Maintaining maximal availability in the presence of database updates. *Computers & Security*, 29(1):88–103, 2010.

[159] I. A. Tondel, M. G. Jaatun, and P. H. Meland. Security requirements for the rest of us: A survey. *IEEE Software*, 25(1):20–27, Jan 2008.

[160] Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus. *ACM Transactions on Database Systems*, 16(2):235–278, 1991.

[161] P. J. Villeneuve and Y. Mao. Lifetime probability of developing lung cancer, by smoking status, Canada. *Cancer Journal of Public Health*, 85(6):385–388, 1994.

[162] Sabrina De Capitani di Vimercati and Giovanni Livraga. Sql access control model. In *Encyclopedia of Cryptography and Security*, pages 1248–1251. Springer, 2011.

[163] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3), 1996.

[164] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3):231–253, 1999.

[165] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 555–566. VLDB Endowment, 2007.

[166] Lena Wiese. Keeping secrets in possibilistic knowledge bases with necessity-valued privacy policies. In *Computational Intelligence for Knowledge-Based Systems Design*, pages 655–664. Springer, 2010.

[167] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 631–647. ACM, 2016.

[168] Raymond W. Yip and Karl N. Levitt. Data level inference detection in database systems. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 179–189. IEEE, 1998.

[169] Stephan Arthur Zdancewic. *Programming Languages for Information Security.* PhD thesis, Cornell University, Ithaca, NY, USA, 2002.

[170] Zheng Zhang and Alberto O. Mendelzon. Authorization views and conditional query containment. In *Proceedings of the 10th International Conference on Database Theory*, pages 259–273. Springer, 2005.

Resume

Marco Guarnieri
Born on 6 March 1988 in Ponte San Pietro, Italy.
Citizen of Italy.

Education

ETH Zurich, Zurich, Switzerland
PhD in Computer Science — October 2012 – present

Università degli Studi di Bergamo, Bergamo, Italy
Laurea Magistrale in Ingegneria Informatica — September 2010 – July 2012
(Master of Science in Computer Engineering)

Laurea in Ingegneria Informatica — September 2007 – September 2010
(Bachelor of Science in Computer Engineering)

Professional Experience

ETH Zurich, Zurich, Switzerland
Research Assistant — October 2012 – present

Università degli Studi di Bergamo, Bergamo, Italy
Research Assistant — August 2012 – September 2012

SAP Labs France, Sophia Antipolis, France
R&D Intern — June 2010 – September 2010

Salvaneschi & **Partners, Software and System Engineering**, Bergamo, Italy
Testing Consultant — October 2009 – March 2010