Strong and Provably Secure Database Access Control

Marco Guarnieri
Institute of Information Security
Department of Computer Science
ETH Zurich, Switzerland
marco.guarnieri@inf.ethz.ch

Srdjan Marinovic
The Wireless Registry, Inc.
Washington DC, US
srdjan@wirelessregistry.com

David Basin
Institute of Information Security
Department of Computer Science
ETH Zurich, Switzerland
basin@inf.ethz.ch

***Abstract*—Existing SQL access control mechanisms are extremely limited. Attackers can leak information and escalate their privileges using advanced database features such as views, triggers, and integrity constraints. This is not merely a problem of vendors lagging behind the state-of-the-art. The theoretical foundations for database security lack adequate security definitions and a realistic attacker model, both of which are needed to evaluate the security of modern databases. We address these issues and present a provably secure access control mechanism that prevents attacks that defeat popular SQL database systems.**

1. Introduction

It is essential to control access to databases that store sensitive information. To this end, the SQL standard defines access control rules and all SQL database vendors 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. 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 [3], [11], [29], [37] 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 §2.

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.

Contributions. We develop a comprehensive formal framework for the design and analysis of database access control. We use it to design and verify an access control mechanism that prevents confidentiality and integrity attacks that defeat existing mechanisms.

First, we develop an operational semantics for databases that supports SQL's core features, as well as triggers, views, and integrity constraints. Our semantics models both the security-critical aspects of these features and the database's dynamic behaviour at the level needed to capture realistic attacks. Our semantics is substantially more detailed than those used in previous works [29], [37], which ignore the database's dynamics.

Second, we develop a novel attacker model that, in addition to SQL's core features, incorporates advanced features such as triggers, views, and integrity constraints. Furthermore, our attacker can infer information based on the semantics of these features. Note that our attacker model subsumes the `SELECT`-only attacker considered in previous works [29], [37]. We also develop an executable version of our operational semantics and attacker model using the Maude term-rewriting framework [12]. The executable model acts as a reference implementation for our semantics. Given the complexity of databases and their features, having an executable version of our models provides a way to validate them against existing database systems and against the examples we use in this paper.

Third, we present two security definitions—database integrity and data confidentiality—that reflect two principal 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 our attacker from exploiting modern SQL databases.

Finally, using our framework, 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 report on in §2.

Related Work. Surprisingly, and in contrast to other areas of information security, 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 just SELECT commands. A number of access control mechanisms, such as [3], [7], [8], [29], [37], 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 [37], explicitly state that update commands are not supported, whereas others [3], [7], [8], [29] ignore what the attacker can learn from update commands. Works on Inference Control [11], [17], [36] and Controlled Query Evaluation [10] 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 [36] supports update commands, it treats them just as a way of increasing data availability, rather than considering them as a possible attack vector.

Database access control mechanisms can be classified into two distinct families [29]. Mechanisms in the *Truman model* [3], [37] transparently modify query results to restrict the user's access to the data authorized by the policy. In contrast, mechanisms in the *Non-Truman model* [7], [8], [29] either accept or reject queries without modifying their results. Different notions of security have been proposed for these models [21], [29], [37]. They are, however, 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. We refer the reader to §7 for further comparison with related work.

Organization. In §2 we present attacks that illustrate serious weaknesses in existing Database Management Systems (DBMSs). In §3 we introduce background and notation about queries, views, triggers, and access control. In §4 we formalize our system and attacker models, and in §5 we define the desired security properties. In §6 we present our access control mechanism, and in §7 we discuss related work. Finally, we draw conclusions in §8. An extended version of this paper, with the system's operational semantics, the attacker model, and complete proofs of all results, is available at [23]. A prototype of our enforcement mechanism and its executable semantics are available at [22].

2. 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 at the end of this section.

2.1. 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 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 [1] 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 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. ■

[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.

DBMS	Integrity Attacks			Confidentiality Attacks	
	Triggers with activator's privileges	Granting views	Revoking views	Table updates and integrity constraints	Triggers with owner's privileges
IBM DB2 (v. 10.5)	†	$\mathcal{X}$	✓	✓	✓
Oracle (v. 11g)	†	$\mathcal{X}$	$\mathcal{X}$	✓	✓
PostgreSQL (v. 9.3.5)	✓	✓	✓	✓	✓
MySQL (v. 14.14)	†	$\mathcal{X}$	✓	✓	✓
SQL Server (v. 12.0)	✓	†	†	✓	✓
Firebird (v. 2.5.2)	✓	$\mathcal{X}$	✓	✓	✓

Figure 1: The ✓ symbol indicates a successful attack, whereas $\mathcal{X}$ indicates a failed attack. The † symbol indicates that the DBMS does not support the features necessary to launch the attack.

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 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 behaviour 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 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.

2.2. 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 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 access control 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. ■

Even though similar attacks have been identified before [24], [34], 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 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 behaviour and users are usually aware of them.

2.3. Discussion

We manually carried out all five attacks against IBM DB2, Oracle Database, PostgreSQL, MySQL, SQL Server, and Firebird. Figure 1 summarizes 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 1 cannot be carried out due to missing features. IBM DB2, MySQL, and Firebird prevent just one of the three attacks, namely Attack 2. However, they all fail to prevent Attack 3. Note that Firebird also fails to prevent Attack 1. In contrast, Attack 1 cannot be carried out against MySQL and IBM DB2 due to missing features. SQL Server also fails to prevent Attack 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.

3. Database Model

We now formalize databases including features like views, access control policies, and triggers. Our formalization of databases and queries follows [2], and our access control policies formalize SQL policies.

3.1. Overview

In this paper we consider the following SQL features: `SELECT`, `INSERT`, `DELETE`, `GRANT`, `REVOKE`, `CREATE TRIGGER`, `CREATE VIEW`, and `ADD USER` commands.

For `SELECT` commands, rather than using SQL, we use the relational calculus (RC), i.e., function-free first-order logic, which has a simple and well-defined semantics [2]. We support `GRANT` commands with the `GRANT OPTION` and `REVOKE` commands with the `CASCADE OPTION`, i.e., when a user revokes a privilege, he also revokes all the privileges that depend on it. 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.

We support only `AFTER` triggers on `INSERT` and `DELETE` events, i.e., triggers that are executed in response to `INSERT` and `DELETE` commands. The triggers' `WHEN` conditions are arbitrary boolean queries and their actions are `GRANT`, `REVOKE`, `INSERT`, or `DELETE` commands. Note that DBMSs usually impose severe restrictions on the `WHEN` clause, such as it must not contain sub-queries. However, most DBMSs 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. Note that all triggers used in §2 belong to this class.

We support two kinds of integrity constraints: functional dependencies and inclusion dependencies [2]. They model the most widely used families of SQL integrity constraints, namely the `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints. We also support views with both the owner's privileges and the activator's privileges.

The SQL fragment we support 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 paper are general and can be extended to the entire SQL standard.

3.2. Databases and Queries

Let $\mathcal{R}$, $\mathcal{U}$, $\mathcal{V}$, and $\mathcal{T}$ be mutually disjoint, countably infinite sets, respectively representing identifiers of relation schemas, users, views, and triggers.

A *database schema* D is a pair $\langle \Sigma, \mathbf{dom} \rangle$, where Σ is a first-order signature and $\mathbf{dom}$ is a fixed countably infinite domain. The signature Σ consists of a set of *relation schemas* $R \in \mathcal{R}$, also called *tables*, with arity $|R|$ and sort $sort(R)$. A *state* s of D is a finite Σ-structure over $\mathbf{dom}$. We denote by Ω_D the set of all states. Given a table $R \in D$, $s(R)$ denotes the set of tuples that belong to R in s.

A *query* q over a schema D is of the form $\{\overline{x} \mid \phi\}$, where $\overline{x}$ is a sequence of variables, ϕ is a relational calculus formula over D, and ϕ's free variables are those in $\overline{x}$. A *boolean query* is a query $\{ \mid \phi\}$, also written as ϕ, where ϕ is a sentence. The result of executing a query q on a state s, denoted by $[q]^s$, is a boolean value in $\{\top, \bot\}$, if q is a boolean query, or a set of tuples otherwise. We denote by RC (respectively RC_{bool}) the set of all relational calculus queries (respectively sentences). We consider only *domain-independent queries* as is standard, and we employ the standard relational calculus semantics [2].

Let $D = \langle \Sigma, \mathbf{dom} \rangle$ be a schema, s be a state in Ω_D, R be a table in D, and $\overline{t}$ be a tuple in $\mathbf{dom}^{|R|}$. The result of inserting (respectively deleting) $\overline{t}$ in R in the state s is the state s', denoted by $s[R \oplus \overline{t}]$ (respectively $s[R \ominus \overline{t}]$), where $s'(T) = s(T)$ for all $T \in \Sigma$ such that $T \neq R$, and $s'(R) = s(R) \cup \{\overline{t}\}$ (respectively $s'(R) = s(R) \setminus \{\overline{t}\}$).

An *integrity constraint over* D is a relational calculus sentence γ over D. Given a state s, we say that *s satisfies the constraint* γ iff $[\gamma]^s = \top$. Given a set of constraints Γ,

Ω_D^Γ denotes the set of all states satisfying the constraints in Γ, i.e., $\Omega_D^\Gamma = \{s \in \Omega_D \mid \bigwedge_{\gamma\in\Gamma}[\gamma]^s = \top\}$. We consider two types of integrity constraints: *functional dependencies*, which are sentences of the form $\forall \overline{x}, \overline{y}, \overline{y}', \overline{z}, \overline{z}'. ((R(\overline{x},\overline{y},\overline{z}) \wedge R(\overline{x},\overline{y}',\overline{z}')) \Rightarrow \overline{y} = \overline{y}')$, and *inclusion dependencies*, which are sentence of the form $\forall \overline{x},\overline{y}. (R(\overline{x},\overline{y}) \Rightarrow \exists \overline{z}. S(\overline{x},\overline{z}))$.

3.3. Views

Let D be a schema. A *view V over D* is a tuple $\langle \mathit{id}, o, q, m\rangle$, where $\mathit{id} \in \mathcal{V}$ 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 [2].

3.4. Access Control Policies

We now formalize the SQL access control model. 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.

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 $\Omega_{U,D}^{sec}$ 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)-access control policy S* as a finite set of (U, D)-grants. We denote by $\mathcal{S}_{U,D}$ the set of all (U, D)-policies.

Example 3.1. Consider the policy described in Attack 5. The database D has three tables: N, P, and T. The set U is $\{u, \mathit{admin}\}$ and the policy S contains the following grants: $\langle\oplus,u,\langle\texttt{SELECT},P\rangle,\mathit{admin}\rangle$, $\langle\oplus,u,\langle\texttt{INSERT},P\rangle,\mathit{admin}\rangle$, $\langle\oplus,u,\langle\texttt{DELETE},P\rangle,\mathit{admin}\rangle$, $\langle\oplus,u,\langle\texttt{SELECT},N\rangle,\mathit{admin}\rangle$, $\langle\oplus,u,\langle\texttt{INSERT},N\rangle,\mathit{admin}\rangle$, and $\langle\oplus,u,\langle\texttt{DELETE},N\rangle,\mathit{admin}\rangle$. ■

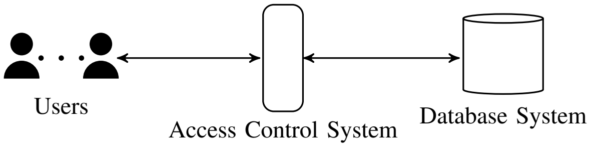


Figure 2: System model.

3.5. Triggers

Let D be a database schema. A *trigger over D* is a tuple $\langle \mathit{id}, u, e, R, \phi, a, m\rangle$, where $\mathit{id} \in \mathcal{T}$ is the trigger identifier, $u \in \mathcal{U}$ is the trigger's owner, $e \in \{\mathit{INS}, \mathit{DEL}\}$ is the trigger event (where INS stands for `INSERT` and DEL stands for `DELETE`), $R \in D$ is a relation schema, the trigger condition ϕ is a relational calculus formula such that $\mathit{free}(\phi) \subseteq \{x_1, \ldots, x_{|R|}\}$, and the trigger action a is one of: (1) $\langle \texttt{INSERT}, R', \overline{t}\rangle$, where $R' \in D$ and $\overline{t}$ is a $|R'|$-tuple of values in $\mathbf{dom}$ and variables in $\{x_1, \ldots, x_{|R|}\}$, (2) $\langle \texttt{DELETE}, R', \overline{t}\rangle$, where R' and $\overline{t}$ are as before, or (3) $\langle \mathit{op}, u, p\rangle$, where $\mathit{op} \in \{\oplus, \oplus^*, \ominus\}$, $u \in \mathcal{U}$, 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.

We assume that any command a is executed atomically together with all the triggers activated by a. We also assume that triggers do not recursively activate other triggers. Hence all executions terminate. We enforce this condition syntactically at the trigger's creation time; see [23] for additional details. The trigger $\langle t, \mathit{admin}, \mathit{INS}, P, T(x_1), \langle \texttt{INSERT}, N, x_1\rangle, O\rangle$ models the trigger in Attack 5. Here, x_1 is bound, at run-time, to the value inserted in P by the trigger's invoker.

4. System and Attacker Model

We next present our system and attacker models. Executable versions of these models, built in the Maude framework [12], are available at [22]. The 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 behaviour. We have executed and validated all of our examples using these models.

4.1. Overview

In our system model, shown in Figure 2, 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 `SELECT`, `INSERT`, `DELETE`, `GRANT`, `REVOKE`, `CREATE TRIGGER`, and `CREATE VIEW` commands.

The *system administrator* is a distinguished user responsible for defining the database schema and the access control 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 $admin$.

The *attacker* is a user, other than the administrator, with an assigned user account who attempts to violate the access control 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. The attacker's inference capabilities are specified using deduction rules.

Access Control System. The access control system protects the confidentiality and integrity of the data in the database. It is configured with an access control 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 DBMS, 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 [29]; see related work for more details.

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.

4.2. System Model

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 behaviour as a small step operational semantics, including all features necessary to reason about security, even in the presence of attacks like those illustrated in §2.

A *system configuration* is a tuple $\langle D, \Gamma \rangle$ such that D is a schema and Γ is a finite set of integrity constraints over D. 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:

- $\langle u, \mathtt{ADD\_USER}, u' \rangle$, where $u = admin$ and $u' \in \mathcal{U} \setminus \{admin\}$,
- $\langle u, \mathtt{SELECT}, q \rangle$, where q is a boolean query[2] over D,
- $\langle u, \mathtt{INSERT}, R, \overline{t} \rangle$, where $R \in D$ and $\overline{t} \in \mathbf{dom}^{|R|}$,
- $\langle u, \mathtt{DELETE}, R, \overline{t} \rangle$, where R and $\overline{t}$ are as above,
- $\langle op, u', p, u \rangle$, where $\langle op, u', p, u \rangle \in \Omega^{sec}_{D,\mathcal{U}}$, or
- $\langle u, \mathtt{CREATE}, o \rangle$, 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 $\bigcup_{u \in U} \mathcal{A}_{D,u}$.

An M*-context* 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.

An M*-state* is a tuple $\langle db, U, sec, T, V, c \rangle$ such that $db \in \Omega^{\Gamma}_D$ is a database state, $U \subset \mathcal{U}$ is a finite set of users such that $admin \in U$, $sec \in \mathcal{S}_{U,D}$ is a security policy, T is a finite set of triggers over D owned by users in U, V is a finite set of views over D owned by users in U, and $c \in \mathcal{C}_M$ is an M-context. We denote by Ω_M the set of all M-states. An M-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.

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 state s and action a to an access control 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.

We now define the LTS representing the system model.

Definition 4.1. 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 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.

The relation $\to_f$ formalizes the system's small step operational semantics. Figure 3 shows three rules describing the successful execution of `SELECT` and `INSERT` commands, as well as triggers. In the rules, we represent context changes using the update function upd, which takes as input an M-state and an action $a \in \mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D$, and returns

[2] Without loss of generality, we focus only on boolean queries [2]. 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.

$$\frac{\begin{array}{c} s = \langle db, sec, U, T, V, c\rangle \;\; f(s, \langle u, \texttt{SELECT}, q\rangle) = \top \;\; trg(s) = \epsilon \\ s' = \langle db, sec, U, T, V, c'\rangle \qquad c' = upd(s, \langle u, \texttt{SELECT}, q\rangle) \end{array}}{s \xrightarrow{\langle u, \texttt{SELECT}, q\rangle}_f s'} \;\; \begin{array}{c}\texttt{SELECT}\\ \text{Success}\end{array}$$

$$\frac{\begin{array}{c} s = \langle db, sec, U, T, V, c\rangle \quad f(s, \langle u, \texttt{INSERT}, R, \bar{t}\rangle) = \top \\ s' = \langle db[R \oplus \bar{t}], sec, U, T, V, c'\rangle \qquad db[R \oplus \bar{t}] \in \Omega_D^\Gamma \\ c' = upd(s, \langle u, \texttt{INSERT}, R, \bar{t}\rangle) \qquad trg(s) = \epsilon \end{array}}{s \xrightarrow{\langle u, \texttt{INSERT}, R, \bar{t}\rangle}_f s'} \;\; \begin{array}{c}\texttt{INSERT}\\ \text{Success}\end{array}$$

$$\frac{\begin{array}{c} s = \langle db, sec, U, T, V, c\rangle \qquad \overline{v} = tpl(s) \\ u = user(m, owner, invoker(s)) \\ trg(s) = \langle id, owner, ev, R', \phi, st, m\rangle \\ f(s, \langle u, \texttt{SELECT}, \phi[\overline{x} \mapsto \overline{v}]\rangle) = \top \quad [\phi[\overline{x} \mapsto \overline{v}]]^{db} = \top \\ \langle u, \texttt{INSERT}, R, \overline{v}'\rangle = act(st, u, \overline{v}) \\ f(s, \langle u, \texttt{INSERT}, R, \overline{v}'\rangle) = \top \qquad c' = upd(s, trg(s)) \\ s' = \langle db[R \oplus \overline{v}'], sec, U, T, V, c'\rangle \quad db[R \oplus \overline{v}'] \in \Omega_D^\Gamma \end{array}}{s \xrightarrow{trg(s)}_f s'} \;\; \begin{array}{c}\text{Trigger}\\ \texttt{INSERT}\\ \text{Success}\end{array}$$

Figure 3: Examples of system model's rules.

the updated context. This function, for instance, updates the system's history stored in the context. The function trg takes as input a system state s and returns the first trigger in the list of scheduled triggers stored in s's context. If there are no triggers to be executed, then $trg(s) = \epsilon$. The rule *`SELECT` Success* models the system's behaviour when the user u issues a `SELECT` query q that is authorized by the PDP f. The only component of the M-state s that changes is the context c. Namely, c' is obtained from c by updating the history and storing q's result. Similarly, the rule *`INSERT` Success* describes how the system behaves after a successful `INSERT` command, i.e., one that neither violates the integrity constraints nor causes security exceptions. The database state db is updated by adding the tuple $\bar{t}$ to R and the context is updated from c to c' by (a) storing the action's result, (b) storing the triggers that must be executed in response to the `INSERT` event, and (c) keeping track of the previous state in case a roll-back is needed.

The *Trigger `INSERT` Success* rule describes how the system executes a trigger whose action is an `INSERT`. The system extracts from the context the trigger t to be executed, i.e., $t = trg(s)$. It determines, using the function $user$, the user u under whose privileges the trigger t is executed, which is, depending on t's security mode, either the invoker $invoker(s)$ or t's owner. It then checks that u is authorized to execute the `SELECT` statement associated with t's `WHEN` condition, and that this condition is satisfied. Afterwards, it computes the actual action using the function act, which instantiates the free variables in t's definition with the values in the tuple $tpl(s)$, i.e., the tuple associated with the action that fired t. Finally, the system updates the database state db by adding the tuple $\overline{v}'$ to R and the context by storing the results of t's execution and removing t from the list of scheduled triggers.

In [23], we give the complete formalization of our labelled transition system. This includes formalizing contexts and all the rules defining the transition relation $\rightarrow_f$. Our operational semantics can be tailored to model the behaviour of specific DBMSs. Thus, using our executable model, available at [22], it is possible to validate our operational semantics against different existing DBMSs.

4.3. Attacker Model

We model attackers that interact with the system through SQL commands and infer information from the system's behaviour 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 §5, and the mechanism we develop in §6, can 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-state $\langle db, U, sec, T, V, c\rangle$, 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 §6 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 behaviour. 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 behaviour, 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 behaviour. The idea of representing the attacker's knowledge using

$$\frac{\begin{array}{c} r^i = r^{i-1} \cdot \langle u, \texttt{DELETE}, R, \overline{t}\rangle \cdot s \quad 1 < i \le |r| \\ s \in \Omega_M \quad \mathit{secEx}(s) = \bot \quad \mathit{Ex}(s) = \emptyset \end{array}}{r, i \vdash_u \neg R(\overline{t})} \;\begin{array}{c}\texttt{DELETE}\\ \text{Success}\end{array} \qquad \frac{\begin{array}{c} r^i = r^{i-1} \cdot \langle u, \texttt{SELECT}, \phi\rangle \cdot s \quad 1 < i \le |r| \quad s \in \Omega_M \\ \mathit{secEx}(s) = \bot \quad \mathit{Ex}(s) = \emptyset \quad \mathit{res}(s) = \top \end{array}}{r, i \vdash_u \phi} \;\begin{array}{c}\texttt{SELECT}\\ \text{Success}\end{array}$$

$$\frac{\begin{array}{c} r^{i+1} = r^i \cdot t \cdot s \quad \mathit{invoker}(\mathit{last}(r^i)) = u \quad s \in \Omega_M \quad 1 \le i < |r| \\ \mathit{secEx}(s) = \bot \quad \mathit{Ex}(s) = \emptyset \quad r, i \vdash_u \neg\psi \quad r, i+1 \vdash_u \psi \\ t = \langle id, ow, ev, R', \phi(\overline{x}), \langle \texttt{INSERT}, R, \overline{t}\rangle, m\rangle \end{array}}{r, i \vdash_u \phi[\overline{x} \mapsto \mathit{tpl}(\mathit{last}(r^i))]} \;\begin{array}{c}\text{Learn}\\ \texttt{INSERT}\\ \text{Backward}\end{array} \qquad \frac{\begin{array}{c} r^{i+1} = r^i \cdot \langle u, \texttt{SELECT}, \phi\rangle \cdot s \\ r, i+1 \vdash_u \psi \quad s \in \Omega_M \quad 1 \le i < |r| \end{array}}{r, i \vdash_u \psi} \;\begin{array}{c}\text{Propagate}\\ \text{Backward}\\ \texttt{SELECT}\end{array}$$

$$\frac{\begin{array}{c} r, i-1 \vdash_u \phi \quad r^i = r^{i-1} \cdot \langle u, op, R, \overline{t}\rangle \cdot s \quad s \in \Omega_M \quad 1 < i \le |r| \\ \mathit{secEx}(s) = \bot \quad \mathit{Ex}(s) = \emptyset \quad \mathit{revise}(r^{i-1}, \phi, r^i) = \top \quad op \in \{\texttt{INSERT}, \texttt{DELETE}\} \end{array}}{r, i \vdash_u \phi} \;\begin{array}{c}\text{Propagate Forward}\\ \text{Update Success}\end{array}$$

Figure 4: Example of attacker inference rules, where $r, i \vdash_u \phi$ denotes that this judgment holds in $\mathcal{ATK}_u$.

sentences ϕ is inspired by existing formalisms for Inference Control [11], [17] and Controlled Query Evaluation [10].

Definition 4.2. 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 $r \in \mathit{traces}(L)$, $1 \le i \le |r|$, and $\phi \in RC_{bool}$. 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 paper. 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 4 shows five representative rules. The complete formalization of all rules is given in [23]. In the following, when we say that a judgment $r, i \vdash_u \phi$ holds, we always mean with respect to the attacker model $\mathcal{ATK}_u$.

Note that $\mathcal{ATK}_u$ 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 behaviour, 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 model.

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 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.

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 $\mathit{invoker}(s)$ denotes the user who fired the trigger that is executed in the state s, whereas $\mathit{tpl}(s)$ denotes the tuple associated with the action that fired the trigger that is executed in the state s.

Example 4.1. Let the schema, the set of users U, and the policy S be as in Example 3.1. The database state db is $db(N) = \{v\}$, $db(P) = \emptyset$, and $db(T) = \{v\}$. 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.

$$\begin{tikzcd}
\langle db, U, S, \{t\}, \emptyset, c_1\rangle \arrow[r, "{\langle u, \texttt{DELETE}, N, v\rangle}"] & \langle db[N \ominus v], U, S, \{t\}, \emptyset, c_2\rangle \arrow[d, "{\langle u, \texttt{INSERT}, P, v\rangle}"'] \\
\langle db[P \oplus v], U, S, \{t\}, \emptyset, c_4\rangle \arrow[d, "{\langle u, \texttt{SELECT}, N(v)\rangle}"] & \langle db[P \oplus v, N \ominus v], U, S, \{t\}, \emptyset, c_3\rangle \arrow[l, "t"'] \\
\langle db[P \oplus v], U, S, \{t\}, \emptyset, c_5\rangle &
\end{tikzcd}$$

Figure 5 models u's reasoning in Attack 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

$$\dfrac{\dfrac{\dfrac{}{r,2 \vdash_u \neg N(v)}\ \texttt{DELETE}\text{ Success}}{r,3 \vdash_u \neg N(v)}\ \begin{array}{c}\text{Propagate Forward}\\ \text{Update Success}\end{array} \qquad \dfrac{\dfrac{}{r,5 \vdash_u N(v)}\ \texttt{SELECT}\text{ Success}}{r,4 \vdash_u N(v)}\ \text{Propagate Backward }\texttt{SELECT}}{r,3 \vdash_u T(v)}\ \text{Learn }\texttt{INSERT}\text{ Backward}$$

Figure 5: Template Derivation of Attack 5 (contains just selected subgoals)

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 [22] to derive the judgments. ■

5. Security Properties

Here we define two security properties: database integrity and data confidentiality. These properties capture the two essential aspects of database security. Database integrity states that all actions modifying the system's state are authorized by the system's policy. In contrast, data confidentiality states that all information that an attacker can learn by observing the system's behaviour is authorized.

These two properties formalize security guarantees with respect to the two different classes of attacks previously identified. An access control mechanism providing database integrity prevents non-authorized changes to the system's state and, thereby, prevents integrity attacks. Similarly, by preventing the leakage of sensitive data, a mechanism providing data confidentiality prevents confidentiality attacks.

5.1. Database Integrity

Database integrity requires a formalization of authorized actions. We therefore define the relation $\rightsquigarrow_{auth}$ between states and actions, modelling which actions are authorized in a given state. Let $P = \langle M, f\rangle$ be an extended configuration, where $M = \langle D, \Gamma\rangle$ and f is an M-PDP. The relation $\rightsquigarrow_{auth} \subseteq \Omega_M \times (\mathcal{A}_{D,\mathcal{U}} \cup \mathcal{TRIGGER}_D)$ is defined by a set of rules given in [23]. Figure 6 shows three representative rules. The *GRANT* rule says that the owner o of a view v with owner's privileges is authorized to delegate the SELECT privilege over v to a user u in the state s, if o has the SELECT privilege with grant option over a set of tables and views that determine v's materialization [28], denoted by $hasAccess(s, v, o, \oplus^*)$. The *TRIGGER* rule says that the execution of an enabled trigger, i.e., one 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 $\rightsquigarrow_{auth}$. Note that the *act* 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 $\mathbf{dom}$. Finally, the *REVOKE* rule says that a REVOKE statement is authorized if the resulting state, obtained using the function *apply*, has a consistent policy, namely one in which all the GRANTs are authorized by $\rightsquigarrow_{auth}$.

$$\dfrac{\begin{array}{c} s = \langle db, U, sec, T, V, c\rangle \quad u, o \in U \quad op \in \{\oplus, \oplus^*\} \\ priv = \langle \texttt{SELECT}, v\rangle \quad v = \langle id, o, q, O\rangle \quad v \in V \\ hasAccess(s, v, o, \oplus^*) \end{array}}{s \rightsquigarrow_{auth} \langle op, u, priv, o\rangle}\ \texttt{GRANT}$$

$$\dfrac{\begin{array}{c} s = \langle db, U, sec, T, V, c\rangle \quad t = \langle id, ow, ev, R, \phi, st, A\rangle \\ [\phi[\overline{x} \mapsto tpl(s)]]^{db} = \top \quad s \rightsquigarrow_{auth} act(st, ow, tpl(s)) \\ s \rightsquigarrow_{auth} act(st, invoker(s), tpl(s)) \qquad t \in T \end{array}}{s \rightsquigarrow_{auth} t}\ \texttt{TRIGGER}$$

$$\dfrac{\begin{array}{c} s = \langle db, U, sec, T, V, c\rangle \quad s' = \langle db, U, sec', T, V, c\rangle \\ s' = apply(\langle \ominus, u, p, u'\rangle, s) \quad \forall g \in sec'.\, s' \rightsquigarrow_{auth} g \end{array}}{s \rightsquigarrow_{auth} \langle \ominus, u, p, u'\rangle}\ \texttt{REVOKE}$$

Figure 6: Examples of $\rightsquigarrow_{auth}$ rules.

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 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 5.1. 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 5.1. We consider a run corresponding to Attack 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.

$$\begin{tikzcd}
\langle db, U, sec, \emptyset, \emptyset, c_1\rangle \arrow[r, "{\langle u_1, \texttt{CREATE}, t\rangle}"] & \langle db, U, sec, \{t\}, \emptyset, c_2\rangle \arrow[d, "{\langle u_2, \texttt{INSERT}, P, v\rangle}"'] \\
\langle db[P \oplus v, S \ominus z], U, sec, \{t\}, \emptyset, c_4\rangle & \langle db[P \oplus v], U, sec, \{t\}, \emptyset, c_3\rangle \arrow[l, "t"]
\end{tikzcd}$$

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 $\langle db[P \oplus v], U, sec, \{t\}, \emptyset, c_3\rangle \not\to_{auth} t$. ■

5.2. Data Confidentiality

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.

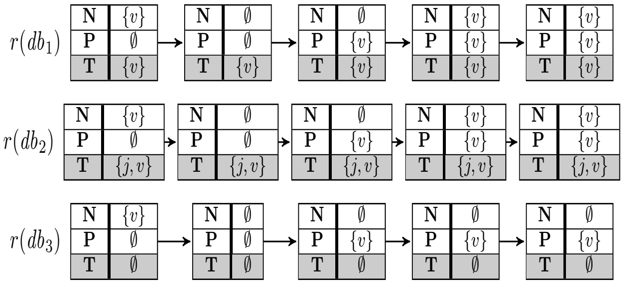
We now define the concept of a secure judgment, which is a judgment that does not leak sensitive information or, equivalently, one that cannot be used to differentiate between indistinguishable runs.

Definition 5.2. 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 $last(r^i) = \langle db, U, S, T, V, c\rangle$ and $last(r') = \langle db', U', S', T', V', c'\rangle$. □

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 5.3. 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. □

We now define the indistinguishability relation that we use in the rest of the paper, which captures what each user can observe (as stated in §4.3) and the effects of the system's access control 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 a 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 $\cong_{P,u}$, which is formally defined in [23]. Intuitively, 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 views, the triggers, the users, and the data disclosed by the policy are the same in r and r'.



$r(db_1)$	N	$\{v\}$	$\emptyset$	$\emptyset$	$\{v\}$	$\{v\}$
	P	$\emptyset$	$\emptyset$	$\{v\}$	$\{v\}$	$\{v\}$
	T	$\{v\}$	$\{v\}$	$\{v\}$	$\{v\}$	$\{v\}$
$r(db_2)$	N	$\{v\}$	$\emptyset$	$\emptyset$	$\{v\}$	$\{v\}$
	P	$\emptyset$	$\emptyset$	$\{v\}$	$\{v\}$	$\{v\}$
	T	$\{j,v\}$	$\{j,v\}$	$\{j,v\}$	$\{j,v\}$	$\{j,v\}$
$r(db_3)$	N	$\{v\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	P	$\emptyset$	$\emptyset$	$\{v\}$	$\{v\}$	$\{v\}$
	T	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 7: The runs $r(db_1)$ and $r(db_2)$ are indistinguishable, whereas $r(db_1)$ and $r(db_3)$ are not.

We remark that there is a close relation between $\cong_{P,u}$ and state-based indistinguishability [21], [29], [37]. 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 under existing state-based notions [21], [29], [37].

Example 5.2 illustrates our indistinguishability notion.

Example 5.2. Let the schema, the set of users, the policy, and the triggers be as in Example 4.1. 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 4.1. Figure 7 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`, 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. ■

Example 5.3 shows that existing PDPs leak sensitive information and therefore do not provide data confidentiality.

Example 5.3. In Example 4.1, 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 5.2, in which $T(v)$ does not hold. ■

Example 5.4 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* [5], [6].

Example 5.4. Consider a database with two tables T and Z and a view $V = \langle v, admin, \{x \mid T(x) \land Z(x)\}, O\rangle$. 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.

$$\begin{tikzcd} \langle db, U, S, \emptyset, \{V\}, c_1\rangle \arrow[r, "{\langle u, \texttt{INSERT}, T, 27\rangle}"] & \langle db[T \oplus 27], U, S, \emptyset, \{V\}, c_2\rangle \arrow[d, "{\langle u, \texttt{SELECT}, V(27)\rangle}"'] \\ & \langle db[T \oplus 27], U, S, \emptyset, \{V\}, c_3\rangle \end{tikzcd}$$

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 [21], [29], [37], we have defined data confidentiality over runs. This is essential to model and detect attacks, such as those in Examples 5.3 and 5.4, where an attacker infers sensitive information from the transitions between states. For instance, the leakage in Example 5.4 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. 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.

Figure 8 depicts our PDP f together with the functions f_{int} and f_{conf}. Additional details about the PDP are given in [23]. 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 results in [21], [25], [28], 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.

6.1. 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. The function $auth$, which checks if a is authorized with respect to s, is a sound and computable under-approximation of $\rightsquigarrow_{auth}$. Thus, any action authorized by f_{int} is authorized according to $\rightsquigarrow_{auth}$. This ensures database integrity. Note that $\rightsquigarrow_{auth}$ relies on the concept of *determinacy* [28] to decide whether a query is determined by a set of views. Since determinacy is undecidable [28], in $auth$ we implement a sound under-approximation of it, given in [23], that checks syntactically if a query is determined by a set of views.

Example 6.1. 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 $V = \langle v, admin, \{x \mid T(x) \land Z(x)\}, O\rangle$ and $W = \langle w, u, \{x \mid R(x) \lor V(x)\}, O\rangle$. 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 $\{x \mid R(x) \lor (T(x) \land Z(x))\}$ and u is authorized to delegate the read access for R, T, and Z. ■

6.2. Enforcing Data Confidentiality

The function f_{conf}, shown in Figure 8, takes as input an action a, a state s, and a user u. Note that any user other than

```
$\triangleright$ $s$ is a state and $a$ is an action
function $f(s, a)$
  return $f_{int}(s, a) \wedge f_{conf}(s, a, user(s, a))$
```

```
$\triangleright$ $s$ is a state and $a$ is an action
function $f_{int}(s, a)$
  if $trg(s) = \epsilon$ return $auth(s, a)$
    else if $a = cond(trg(s), s)$ return $\top$
      else if $a = act(trg(s), s)$ return $auth(s, trg(s))$
        else return $\bot$
```

```
   $\triangleright$ $s$ 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 $leak(a, s, u) \vee \neg secure(u, getInfo(a), s)$ return $\bot$
5.     for $\gamma \in Dep(a, \Gamma)$
6.       if $(\neg secure(u, getInfoS(\gamma, a), s) \vee \neg secure(u, getInfoV(\gamma, a), s))$
7.         return $\bot$
8.   case $\langle \oplus, u'', pr, u'\rangle, \langle \oplus^*, u'', pr, u'\rangle$ : return $\neg leak(a, s, u)$
9. return $\top$
```

Figure 8: 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.

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 5.3. 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 3–7), f_{conf} checks (line 4), using the *leak* procedure, whether a's execution may leak sensitive information through the views that u can read, as in Example 5.4. Afterwards, f_{conf} also checks (line 4) whether the information u can learn from a's execution, modelled by the sentence computed by the procedure $getInfo(a)$, is secure. In line 5–7, 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 (line 8), f_{conf} checks whether a's successful execution discloses sensitive information to u. In the remaining cases (line 9), f_{conf} authorizes a.

Secure judgments. Determining if a given judgment is secure is undecidable for RC [21], [25]. 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 [23].

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 given in [23]. 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 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 $\langle op, u, \langle \texttt{SELECT}, K'\rangle, u'\rangle \in sec$ 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:

$$\phi^{v}_{s,u} = \begin{cases} \bigvee_{S \in R^{\top}_{s,u}} S(\overline{x}) & \text{if } \phi = R(\overline{x}) \text{ and } v = \top \\ \bigwedge_{S \in R^{\bot}_{s,u}} S(\overline{x}) & \text{if } \phi = R(\overline{x}) \text{ and } v = \bot \\ \neg\psi^{\neg v}_{s,u} & \text{if } \phi = \neg\psi \\ \psi^{v}_{s,u} * \gamma^{v}_{s,u} & \text{if } \phi = \psi * \gamma \text{ and } * \in \{\vee, \wedge\} \\ Q\, x.\, \psi^{v}_{s,u} & \text{if } \phi = Q\, x.\, \psi \text{ and } Q \in \{\exists, \forall\} \\ \phi & \text{otherwise} \end{cases}$$

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

$S_y{}^{\top}_{s,u} = \{V_y\}$ $R^{\top}_{s,u} = \emptyset$

$S_y{}^{\perp}_{s,u} = \emptyset$ $R^{\perp}_{s,u} = \{W\}$

$S_x = \{y \mid \exists x.\, S(x, y)\}$ $V_x = \{y \mid \exists x.\, V(x, y)\}$

$S_y = \{x \mid \exists y.\, S(x, y)\}$ $V_y = \{x \mid \exists y.\, V(x, y)\}$

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

$\phi_{rw} := \neg\phi^{\top}_{s,u} \wedge \phi^{\perp}_{s,u}$

$\phi^{\top}_{s,u} := S_y(2)^{\top}_{s,u} \wedge (\neg R(5)^{\perp}_{s,u} \vee S_y(4)^{\top}_{s,u}) \equiv V_y(2) \wedge (\neg W(5) \vee V_y(4))$

$\phi^{\perp}_{s,u} := S_y(2)^{\perp}_{s,u} \wedge (\neg R(5)^{\top}_{s,u} \vee S_y(4)^{\perp}_{s,u}) \equiv \top$

Figure 9: 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 6.2.

The formulae are such that if $\phi^{\top}_{s,u}$ holds, then ϕ holds and if $\neg\phi^{\perp}_{s,u}$ holds, then $\neg\phi$ holds. To compute the sets $R^{\top}_{s,u}$ and $R^{\perp}_{s,u}$, we check the containment between queries. Since query containment is undecidable [2], we implement a sound under-approximation of it, described in [23]. Other sound under-approximations can be used as well.

Our $\phi^{\top}_{s,u}$ and $\phi^{\perp}_{s,u}$ rewritings share similarities with the *low* and *high evaluations* of Wang et al. [37]. Both try to approximate the result of a query just by looking at the authorized data. However, we use $\phi^{\top}_{s,u}$ and $\phi^{\perp}_{s,u}$ to determine a judgment's security, whereas Wang et al. use evaluations to restrict the query's results only to authorized data.

Example 6.2. 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 $\{\langle\oplus, u, \langle \texttt{SELECT}, V\rangle, admin\rangle, \langle\oplus, u, \langle \texttt{SELECT}, W\rangle, admin\rangle\}$. 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 9 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 also in Figure 9, proceeds as follows. We first rewrite the formula ϕ as $S_y(2) \wedge (\neg R(5) \vee S_y(4))$. The sets $S_y{}^{\top}_{s,u}$, $S_y{}^{\perp}_{s,u}$, $R^{\top}_{s,u}$, and $R^{\perp}_{s,u}$ are respectively $\{V_y\}$, $\emptyset$, $\emptyset$, and $\{W\}$. The formulae $\phi^{\top}_{s,u}$ and $\phi^{\perp}_{s,u}$ are respectively $S_y(2)^{\top}_{s,u} \wedge (\neg R(5)^{\perp}_{s,u} \vee S_y(4)^{\top}_{s,u})$, which is equivalent to $V_y(2) \wedge (\neg W(5) \vee V_y(4))$, and $S_y(2)^{\perp}_{s,u} \wedge (\neg R(5)^{\top}_{s,u} \vee S_y(4)^{\perp}_{s,u})$, 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 $\neg\phi^{\top}_{s,u} \wedge \phi^{\perp}_{s,u}$. Since ϕ_{rw} does not hold in s, it follows that $r, 1 \vdash_u \phi$ is secure. ■

6.3. Theoretical Evaluation

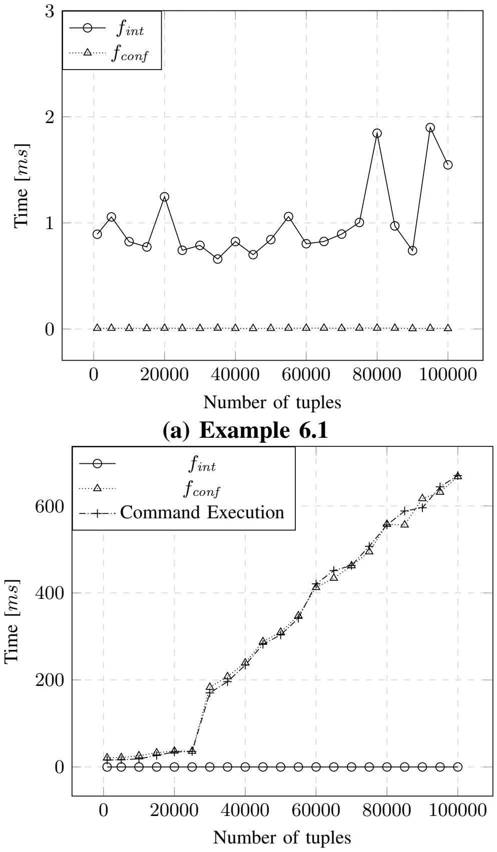
Our PDP provides the desired security guarantees and its data complexity, i.e., the complexity of executing f when the action, the policy, the triggers, and the views are fixed, is AC^0. This means that f can be evaluated in logarithmic space in the database's size, as $AC^0 \subseteq LOGSPACE$, and 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 [2]. 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 [23].

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 6.3 and 6.4 below show, f is more permissive than existing PDPs for those actions that violate neither database integrity nor data confidentiality.

Example 6.3. 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 6.1. 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 6.4. 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 6.2 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. [29] rejects the query in Example 6.2 as insecure. However, our solution and the proposal of Rizvi et al. [29] are incomparable in terms of permissiveness, i.e., some secure `SELECT` queries are authorized by one mechanism and not by the other. ■



(a) Example 6.1

(b) Example 6.2

Figure 10: PDP Execution time.

6.4. Implementation

To evaluate the feasibility and security of our approach in practice, we implemented our PDP in Java. The prototype, available at [22], 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 §2 against our prototype. As expected, our PDP prevents all the attacks. For (2), we simulated Examples 6.1 and 6.2 on database states where the number of tuples ranges from 1,000 to 100,000. Figure 10 shows the PDP's execution time. 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 6.2, f_{conf}'s execution time is comparable to the execution time of the user's query. As Figure 11 shows, f_{conf}'s query rewriting time does not depend on the database's size, whereas f_{conf}'s query execution time does.

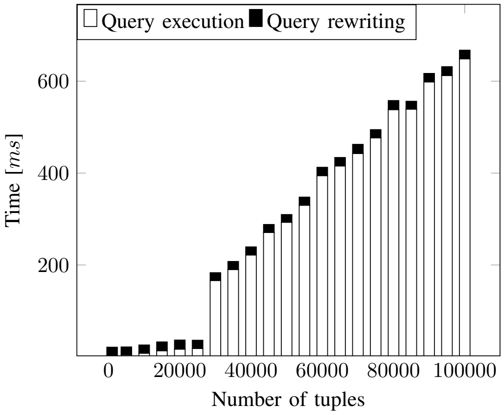


Figure 11: Example 8 : f_{conf}'s execution time.

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. We will investigate further optimizations as a future work.

7. Related Work and Discussion

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.

7.1. Database Access Control

Discretionary Database Access Control. Our framework builds on prior research in database access control [21], [29], [37] as well as established notions from database theory, such as determinacy [28] and instance-based determinacy [25].

Specifically, our notion of secure judgments extends instance-based determinacy from database states to runs, while data confidentiality extends existing security notions [21], [29], [37] to dynamic settings, where both the database and the policy may change. Similarly, our indistinguishability notion extends those in [21], [37] 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 [20] that prevents Attacks 2 and 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) [15], 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 [31]. The policy is, in general, fixed (cf. the *tranquillity principle* [31]).

With respect to semantics, existing MLS solutions are based on the so-called *Truman model* [29], where 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. This is called the *Non-Truman model* [29]. For an in-depth comparison of these access control models, see [21], [29]. Operationally, MLS mechanisms use poly-instantiation [24], which is neither supported by the relational model nor by the SQL standard, and requires adhoc extensions [15], [32]. 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 [15], [32] with MLS and BIBA properties [31].

MLS systems prevent attacks similar to Attacks 4 and 5 using poly-instantiated tuples and triggers [32], [35], whereas attacks similar to Attack 1 cannot be carried out because triggers with activator's privileges are not supported [35]. The SeaView system [15], which combines discretionary access control and MLS, additionally prevents attacks similar to Attacks 2 and 3 by relying on Griffiths and Wade's PDP [20]. However, these solutions cannot be applied to SQL databases for the aforementioned reasons.

7.2. Information Flow Control

Various authors have applied ideas from information flow control to databases. Davis and Chen [14] study how cross-application information flows can be tracked through databases. Other researchers [13], [26], [33] 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 paper.

Schultz and Liskov [34] extend decentralized information flow control [27] to databases, based on concepts from multi-level security [15]. They identify one attack on data confidentiality that exploits integrity constraints. Their solution relies on poly-instantiation [24] 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 [4], [18]. Giacobazzi and Mastroeni [18] model attackers as data-flow analysers that observe the program's behaviour, whereas Askarov and Chong [4] 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.

7.3. Discussion

Historically, database access control and information flow control rely on different foundations, formalisms, security notions, and techniques. We see our paper as a starting point for bridging these topics: we combine database access control theory with an operational semantics and an attacker model, which are common in information flow control, but have been largely ignored in database access control. We thereby give a precise logical characterization of the attacker's capabilities and of a judgment's security. Furthermore, our indistinguishability notion has similarities with the low-equivalence notions used in [5], [6], [9], whereas both data confidentiality and the notion of secure judgments have a precise characterization as instances of non-interference [19], [30]; see [23] for more details.

We believe our framework provides a basis for (1) further investigating the connections between these two topics, (2) applying information flow mechanisms, such as type systems or multi-execution [16], to database access control, and (3) investigating how integrity notions used in information flow control can best be applied to databases.

8. Conclusion

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 analysing 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. As a future work, we will extend our framework and our PDP to directly support the SQL language, and we will investigate efficiency improvements for our PDP.

Acknowledgments. We thank Ùlfar Erlingsson, Erwin Fang, Andreas Lochbihler, Ognjen Maric, Mohammad Torabi Dashti, Dmitriy Traytel, Petar Tsankov, Thilo Weghorn, Der-Yeuan Yu, Eugen Zalinescu, as well as the anonymous reviewers for their comments.

References

[1] (2014, Sep.) Manage trigger security, *Microsoft MSDN Library*. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms191134.aspx/

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley Reading, 1995, vol. 8.

[3] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi, "Extending relational database systems to automatically enforce privacy policies," in *Proc. 2005 IEEE Int. Conf. Data Engineering*.

[4] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *Proc. 2012 IEEE Symp. Computer Security Foundations*.

[5] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *Proc. 2007 IEEE Symp. Security and Privacy*.

[6] ——, "Tight enforcement of information-release policies for dynamic languages," in *Proc. 2009 IEEE Symp. Computer Security Foundations*.

[7] G. Bender, L. Kot, and J. Gehrke, "Explainable security for relational databases," in *Proc. 2014 ACM Intl. Conf. Management of data*.

[8] G. M. Bender, L. Kot, J. Gehrke, and C. Koch, "Fine-grained disclosure control for app ecosystems," in *Proc. 2013 ACM Intl. Conf. Management of data*.

[9] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *Proc. 2009 ACM Conf. Computer and Communications Security*.

[10] P. A. Bonatti, S. Kraus, and V. Subrahmanian, "Foundations of secure deductive databases," *IEEE Trans. Knowl. Data Eng.*, vol. 7, no. 3, 1995.

[11] A. Brodsky, C. Farkas, and S. Jajodia, "Secure databases: Constraints, inference channels, and monitoring disclosures," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 6, 2000.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, "The maude 2.0 system," in *Rewriting Techniques and Applications*. Springer, 2003.

[13] B. J. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, label-based security enforcement for web applications," in *Proc. 2009 ACM Intl. Conf. Management of data*.

[14] B. Davis and H. Chen, "DBTaint: cross-application information flow tracking via databases," *Proc. 2010 USENIX Conf. Web Application Development*.

[15] D. E. Denning and T. F. Lunt, "A multilevel relational data model," in *Proc. 1987 IEEE Symp. Security and Privacy*.

[16] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proc. 2010 IEEE Symp. Security and Privacy*.

[17] C. Farkas and S. Jajodia, "The inference problem: a survey," *ACM SIGKDD Explorations*, vol. 4, no. 2, 2002.

[18] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: Parameterizing non-interference by abstract interpretation," in *Proc. 2004 ACM Symp. Principles of Programming Languages*.

[19] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symp. Security and Privacy*, 1982.

[20] P. P. Griffiths and B. W. Wade, "An authorization mechanism for a relational database system," *ACM Trans. on Database Syst.*, vol. 1, no. 3, 1976.

[21] M. Guarnieri and D. Basin, "Optimal security-aware query processing," in *Proc. 2014 Int. Conf. Very Large Data Bases*.

[22] M. Guarnieri, S. Marinovic, and D. Basin. Strong and Provably Secure Database Access Control — Prototype and Maude models. [Online]. Available: http://www.infsec.ethz.ch/research/projects/FDAC.html

[23] ——, "Strong and provably secure database access control," Tech. Rep., 2015. [Online]. Available: http://arxiv.org/abs/1512.01479

[24] S. Jajodia and R. Sandhu, "Polyinstantiation integrity in multilevel relations," in *Proc. 1990 IEEE Symp. Security and Privacy*.

[25] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu, "Query-based data pricing," in *Proc. 2012 ACM Symp. Principles of Database Systems*.

[26] P. Li and S. Zdancewic, "Practical information flow control in web-based information systems," in *Proc. 2005 IEEE Workshop on Computer Security Foundations*.

[27] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proc. 1997 ACM Symp. Operating Systems Principles*.

[28] A. Nash, L. Segoufin, and V. Vianu, "Views and queries: Determinacy and rewriting," *ACM Trans. Database Syst.*, vol. 35, no. 3, 2010.

[29] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *Proc. 2004 ACM Int. Conf. Management of data*.

[30] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, 2003.

[31] P. Samarati and S. Capitani de Vimercati, "Access Control: Policies, Models, and Mechanisms," *Springer Lecture Notes in Computer Science*, vol. 2171, 2001.

[32] R. Sandhu and F. Chen, "The multilevel relational (MLR) data model," *ACM Trans. Inf. Syst. Sec.*, vol. 1, no. 1, 1998.

[33] D. Schoepe, D. Hedin, and A. Sabelfeld, "SeLINQ: tracking information across application-database boundaries," in *Proc. 2014 ACM Intl. Conf. Functional Programming*.

[34] D. Schultz and B. Liskov, "IFDB: decentralized information flow control for databases," in *Proc. 2013 ACM European Conf. Computer Systems*.

[35] K. Smith and M. Winslett, "Multilevel secure rules: Integrating the multilevel secure and active data models," in *Database Security VI: Status and Prospects*. North-Holland, 1993.

[36] T. S. Toland, C. Farkas, and C. M. Eastman, "The inference problem: Maintaining maximal availability in the presence of database updates," *Computers & Security*, vol. 29, no. 1, 2010.

[37] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun, "On the correctness criteria of fine-grained access control in relational databases," in *Proc. 2007 Int. Conf. Very Large Data Bases*.