Synthesis of Sound and Precise Leakage Contracts for Open-Source RISC-V Processors

Zilong Wang*
IMDEA Software Institute
Madrid, Spain
zilong.wang@imdea.org

Gideon Mohr
Saarland University
Saarbrücken, Germany
s8gimohr@stud.uni-saarland.de

Klaus von Gleissenthall
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
k.freiherrvongleissenthal@vu.nl

Jan Reineke
Saarland University
Saarbrücken, Germany
reineke@cs.uni-saarland.de

Marco Guarnieri
IMDEA Software Institute
Madrid, Spain
marco.guarnieri@imdea.org

Abstract

Leakage contracts have been proposed as a new security abstraction at the instruction set architecture level. Leakage contracts aim to capture the information that processors may leak via microarchitectural side channels. Recently, the first tools have emerged to verify whether a processor satisfies a given contract. However, coming up with a contract that is both sound and precise for a given processor is challenging, time-consuming, and error-prone, as it requires in-depth knowledge of the timing side channels introduced by microarchitectural optimizations.

In this paper, we address this challenge by proposing LeaSyn, the first tool for automatically synthesizing leakage contracts that are both *sound* and *precise* for processor designs at register-transfer level. Starting from a user-provided contract template that captures the space of possible contracts, LeaSyn automatically constructs a contract, alternating between contract synthesis, which ensures precision based on an empirical characterization of the processor's leaks, and contract verification, which ensures soundness.

Using LeaSyn, we automatically synthesize contracts for six open-source RISC-V CPUs for a variety of contract templates. Our experiments indicate that LeaSyn's contracts are sound and more precise (*i.e.*, represent the actual leaks in the target processor more faithfully) than contracts constructed by existing approaches.

CCS Concepts

• **Security and privacy → Logic and verification**; **Security in hardware**.

Keywords

Side channels, hardware verification, leakage contracts

ACM Reference Format:
Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2025. Synthesis of Sound and Precise Leakage Contracts for Open-Source RISC-V Processors. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765148

*Also with Universidad Politécnica de Madrid.

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765148

1 Introduction

Microarchitectural attacks [11, 12, 17, 27, 35–37, 43–46, 49, 53, 56, 57] rely on subtle but measurable hardware side effects on a computation's execution time to compromise otherwise secure programs. The exploited side effects result from optimizations—like caching and speculative execution—implemented in a CPU's microarchitecture. Defending against such attacks is challenging since they are "invisible" at the level of the instruction set architecture (ISA), the traditional hardware-software interface familiar to programmers.

Leakage contracts [30] have recently been proposed as a new security abstraction at ISA-level to fill this gap and to serve as a rigorous foundation for writing programs that are resistant against microarchitectural attacks. In a nutshell, a leakage contract specifies at ISA-level what information a program may leak via microarchitectural side channels. Leakage contracts have been successfully applied to reason about the security of hardware [26, 41, 50] and software [14, 24, 25, 29, 42] against microarchitectural leaks.

To prevent microarchitectural attacks, programmers can use leakage contracts as a guideline for secure programming by ensuring that secret information does not influence any of the leakage sources identified in the contracts. To be useful for this, however, leakage contracts should be *sound*, *i.e.*, they should capture all leaks in the underlying processor, and *precise*, *i.e.*, they should expose as part of the contract only the information that is actually leaked. Unsound contracts are problematic since they could lead to insecure code. At the same time, imprecise contracts may unnecessarily rule out programs as "insecure" and introduce performance overheads, as they lead to overly defensive code.

Deriving a sound and precise leakage contract for a given processor is challenging. Manually constructing a leakage contract for a modern processor is a time-consuming [31] and error-prone endeavour: it requires a deep understanding of the processor's microarchitecture and the leakage introduced by its optimizations. Automatic synthesis of leakage contracts promises to address the impracticalities of manually constructing a contract [21, 22, 38]. However, existing approaches for synthesis are limited: The approach by Mohr et al. [38] can synthesize a precise contract directly

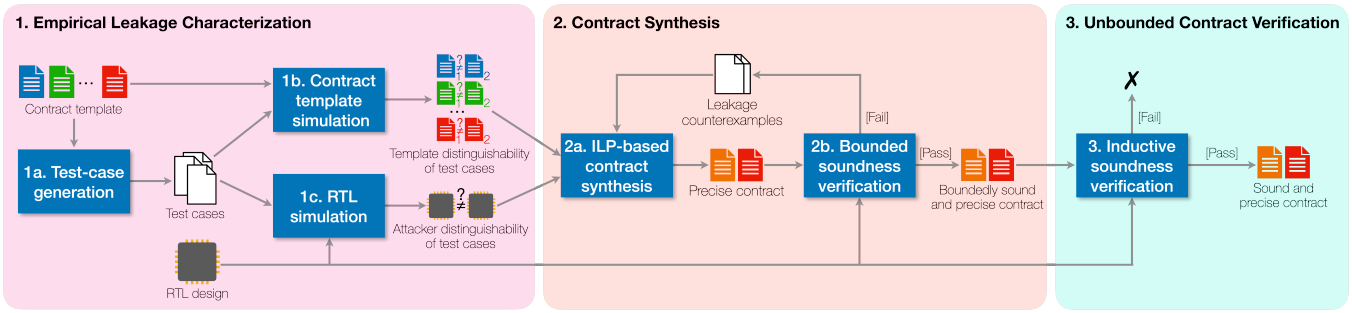


Figure 1: Overview of LeaSyn synthesis approach

from a register-transfer level (RTL) processor design, but it cannot guarantee the contract's soundness, *i.e.*, the contract may miss leaks, compromising its usefulness for secure programming. The approach by Dinesh et al. [21, 22], on the other hand, synthesizes sound but (very) imprecise contracts, *e.g.*, the synthesized contracts entirely rule out the use of any branch or memory instructions. Wang et al. [50] have proposed LeaVe, a tool that can verify the soundness of leakage contracts against RTL designs, and they have used LeaVe to synthesize sound contracts with a human in the loop. As we show in §6, the resulting contracts are still fairly imprecise.

In this paper, we propose a methodology to automatically synthesize leakage contracts that are *sound* and *precise* given an RTL processor design. Next, we describe our main contributions.

Synthesis methodology: We propose a methodology (§4) to synthesize sound and precise contracts given an RTL processor design. Our methodology starts from (a) the RTL design of the target processor, and (b) a user-provided *contract template* that consists of a set of *contract atoms*, each capturing a potential instruction-level leak. Crucially, the template defines the search space for contract synthesis and any subset of the template's atoms is a candidate contract. Figure 1 illustrates our approach, which consists of three phases:

(1) *Empirical Leakage Characterization:* We start by characterizing a processor's leakage based on a set of test cases (§4.1), where a test case consists of a pair of programs with their corresponding inputs. These test cases are automatically generated based on the given contract template, and they aim at exercising the processor to identify leaks. The test cases are classified in terms of attacker distinguishability, *i.e.*, whether the attacker can distinguish the pair of executions or not, and template distinguishability, *i.e.*, which sets of atoms from the template distinguish each test case.

(2) *Contract Synthesis:* We build a candidate contract for the target processor by alternating between a synthesis phase (§4.2) and a verification phase (§4.3). We start by synthesizing the most precise candidate contract that captures all leaks exposed in the test cases, based on the attacker and template distinguishability relations. To ensure that the candidate contract is sound, *i.e.*, it captures all leaks, even those not exercised by the initial test cases, we perform a bounded verification step. Whenever we find a leakage counterexample, *i.e.*, an attacker-distinguishable test case that is contract indistinguishable under the current candidate contract, this counterexample is added to the set of test cases, and a new contract candidate is synthesized.

(3) *Unbounded Contract Verification:* When we cannot find further counterexamples with bounded verification, we perform a final unbounded verification step to ensure that indeed all possible leaks in the target processor are captured by the candidate contract (§4.4). If this passes, the contract is provably sound for the target processor, and we return it to the user.

In contrast to prior approaches for contract synthesis [21, 22, 38], our methodology ensures that the synthesized contract is both *sound*, *i.e.*, it captures all leaks, and *precise*, *i.e.*, it distinguishes as few attacker-indistinguishable test cases as possible.

LeaSyn synthesis tool: We implement our methodology in LeaSyn (§5), a tool for synthesizing leakage contracts from processor designs in Verilog. LeaSyn implements all the steps of our methodology: it uses integer-linear programming for contract synthesis, a bounded model checker for bounded soundness verification, and the LeaVe contract verification tool [50] for unbounded soundness verification. LeaSyn also implements a family of different contract templates capturing instruction-level leaks, which are implemented of top of the RISC-V Formal Interface [6] to simplify applying LeaSyn to new targets.

Evaluation: We validate our methodology by synthesizing leakage contracts for six open-source RISC-V processors from three different core families (§6). We do so by (a) defining a variety of contract templates capturing different classes of instruction-level leaks (e.g., leaks through control flow, variable-latency instructions, and memory accesses), and (b) using LeaSyn to synthesize the most precise and sound contracts satisfied by the studied target processors, against an attacker that observes when instructions retire. Our experiments confirm that LeaSyn can successfully synthesize sound and precise leakage contracts for all of our targets in less than 48 hours. Furthermore, the contracts synthesized by LeaSyn are significantly more precise than existing contracts derived either manually [50] or automatically [21, 22, 33, 38].

Summary: To summarize, we make the following contributions:

- We propose a methodology for synthesizing sound and precise leakage contracts for RISC-V processors at RTL.
- We implement our methodology in a tool called LeaSyn.
- We evaluate LeaSyn on six open-source RISC-V processors, demonstrating that it can efficiently synthesize sound and precise contracts.
- We show that the contracts synthesized by LeaSyn are significantly more precise than those derived in prior work.

2 Overview

We now present the core aspects of our approach with an example.

2.1 A simple processor

We start by introducing a simple ISA and a CPU implementing it.

Instruction Set: We consider a simple instruction set Isa that supports loading immediate values into registers and performing (integer) divisions between them. The architectural state consists of n 32-bit registers, representing signed integers, which we identify using R1, . . . , Rn. It also contains a dedicated register PC holding the program counter. The instruction set consists of two instructions:

(1) The **load immediate** instruction `li RD, imm` sets the value of the destination register RD to imm. RD is a register identifier in R1, . . . , Rn and imm is a 32-bit value.

(2) The **division** instruction `div RD, RS1, RS2` divides RS1 by RS2 and stores the result in RD. RD, RS1, RS2 are register identifiers in R1, . . . , Rn. Inspired by the RISC-V ISA [52], upon a division by zero, *i.e.*, when the value of RS2 is 0, the value of RD is set to −1.

Processor: We consider a simple two-stage pipelined implementation Impl of Isa. The *fetch/decode stage* takes one cycle. It loads the current instruction at the address in PC, decodes it into instruction type and operands, and increments the program counter.

The *execute stage* works as follows:

- Load immediate instructions `li RD, imm` are executed in a single cycle by writing the value imm to the destination register RD.
- Division instructions `div RD, RS1, RS2` are handled by a dedicated division unit. The unit takes 32 cycles to execute all divisions except when RS2 is 0 or 1, in which case it takes 1 cycle.

Whenever the execute stage takes more than one cycle, the fetch/decode stage gets stalled.

2.2 Modeling leaks with leakage contracts

Next, we show how leakage contracts can be used to capture—at ISA level—the leaks existing in the CPU Impl from §2.1.

Leakage: Executing instructions on Impl can leak information about the value of some of the registers. For instance, consider an attacker Atk that observes at which cycles Impl retires each instruction. The attacker can distinguish between the following pairs of program executions, where we represent each execution by a pair $\langle p \mid \sigma \rangle$ consisting of a program p and an initial state σ:

(1) $\langle$`div R1, R2, R3` $\mid \sigma_1\rangle$ *vs.* $\langle$`div R1, R2, R3` $\mid \sigma_2\rangle$, for σ_1, σ_2, s.t. σ_1(R3) = 2 and σ_2(R3) = 1. The two executions are attacker distinguishable since executing `div` takes 33 cycles (1 cycle for fetch/decode and 32 for the execution stage) in the first case, but only 2 cycles in the second case.

(2) $\langle$`div R1, R2, R3` $\mid \sigma_1\rangle$ *vs.* $\langle$`li R1, 0x0` $\mid \sigma_3\rangle$ for some σ_3, are also attacker distinguishable since `div` takes 33 cycles starting from s_1 whereas `li` only takes 2 cycles.

The attacker, however, cannot distinguish the following executions:

(1) $\langle$`div R1, R2, R3` $\mid \sigma_1\rangle$ *vs.* $\langle$`div R1, R2, R3` $\mid \sigma_4\rangle$, for σ_4 with σ_4(R3) = 8, since `div` will take 33 cycles in both cases, and

(2) $\langle$`div R1, R2, R3` $\mid \sigma_2\rangle$ *vs.* $\langle$`li R1, 0x0` $\mid \sigma_3\rangle$, since executing both `div` (since σ_2(R3) = 1) and `li` take 2 cycles.

Leakage contracts: Securely programming Impl requires understanding which program executions an attacker might distinguish to ensure that secret data is not leaked. Leakage contracts [30, 50] provide a way of characterizing such leaks at ISA-level, thereby enabling secure programming. For this, contracts map architectural executions, *i.e.*, executions according to the Isa, to *contract traces* that capture what information might be leaked through side channels.

A contract is *sound* for a CPU (or, equivalently, the CPU satisfies the contract), if the contract captures *all* leaks in the CPU, that is, any two program executions that are distinguishable by Atk must result in different contract traces. For instance, Impl satisfies a contract that exposes all the operands of executed `div` instructions, since distinguishable executions like the ones mentioned above would be mapped to different contract traces. While sound contracts are a prerequisite for secure programming, not all sound contracts are useful. Indeed, consider a contract that exposes the operands of *all* instructions. Despite being sound for Impl, this contract is useless from a secure programming perspective, since it only admits programs that do not process any secrets! This contract is imprecise since it over-approximates the actual leaks in Impl.

2.3 Synthesizing sound and precise contracts with LeaSyn

Figure 1 depicts LeaSyn's approach to synthesize sound and precise contracts given an RTL processor. The approach takes as input:

(1) The RTL design of the processor under synthesis Impl.

(2) A user-provided *contract template* $\mathbb{T}$ consisting of a set of *contract atoms* (§3.2), each capturing potential instruction-level leaks. The template defines the search space for contract synthesis and any subset of its atoms is a candidate contract.

Based on these inputs, LeaSyn outputs a contract Ctr that is (a) *sound*, *i.e.*, it captures *all* leaks in the target CPU Impl, and (b) *as precise as possible*, *i.e.*, among all sound contracts in the template it distinguishes the fewest of the attacker-indistinguishable test cases explored during synthesis. Next, we describe the contract template for our example, and then the three phases of our approach.

Contract template: We consider a contract template that can expose as part of the contract trace three kinds of information: (1) the value ⟦imm⟧ of the immediate imm in load-immediate instructions, (2) the destination and source registers used in `div` instructions (denoted as ⟦RD⟧, ⟦RS1⟧, ⟦RS2⟧), and (3) the values of these registers (denoted as ⟦Reg[RD]⟧, ⟦Reg[RS1]⟧, and ⟦Reg[RS2]⟧).

Furthermore, an observation corresponding to the instruction type $ITYPE = \{\texttt{li}, \texttt{div}\}$ can be added to the trace whenever an instruction of that type is executed.

Therefore, the set of atoms in our template is as follows:

$$\begin{aligned} Atoms = \{&(\texttt{li}, \langle \texttt{"li"}, - \rangle), (\texttt{div}, \langle \texttt{"div"}, - \rangle), (\texttt{li}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle), \\ &(\texttt{div}, \langle \texttt{"RD"}, [\![\texttt{RD}]\!] \rangle), (\texttt{div}, \langle \texttt{"Reg[RD]"}, [\![\texttt{Reg[RD]}]\!] \rangle), \\ &(\texttt{div}, \langle \texttt{"RS1"}, [\![\texttt{RS1}]\!] \rangle), (\texttt{div}, \langle \texttt{"Reg[RS1]"}, [\![\texttt{Reg[RS1]}]\!] \rangle), \\ &(\texttt{div}, \langle \texttt{"RS2"}, [\![\texttt{RS2}]\!] \rangle), (\texttt{div}, \langle \texttt{"Reg[RS2]"}, [\![\texttt{Reg[RS2]}]\!] \rangle)\} \end{aligned}$$

For instance, atom $(\texttt{li}, \langle \texttt{"li"}, - \rangle)$ adds to the trace the tuple $\langle \texttt{"li"}, - \rangle$ whenever a load immediate instruction is executed, whereas atom $(\texttt{div}, \langle \texttt{"Reg[RS2]"}, [\![\texttt{Reg[RS2]}]\!] \rangle)$ exposes the tuple $\langle \texttt{"Reg[RS2]"}, [\![\texttt{Reg[RS2]}]\!] \rangle$ containing the value of the second source operand whenever a `div` instruction is executed.

Phase 1: Empirical leakage characterization: To empirically characterize a processor's leakage, LeaSyn first generates a set of test cases T, where each test case consists of a pair of ISA-level programs with associated data inputs. Then, it simulates the execution of the test cases on the target processor Impl and determines which test cases are *attacker distinguishable* and which are not.

To allow the subsequent synthesis step to associate leakage with particular contract atoms in the template, LeaSyn also simulates each test case on the contract template $\mathbb{T}$. The result of this simulation is condensed into the *template distinguishability* of each test case, which compactly captures which sets of contract atoms distinguish a particular test case and which do not.

In our example, we consider an attacker Atk that observes when instructions retire and we systematically generate the following test cases, each consisting of a pairs of program executions:

(T_1) $\langle$ `li R1, 0x1234` | $\sigma\rangle$ *vs.* $\langle$ `li R1, 0x5678` | $\sigma\rangle$ for some σ.
(T_2) $\langle$ `li R1, 0x1234` | $\sigma\rangle$ *vs.* $\langle$ `li R2, 0x1234` | $\sigma\rangle$ for some σ.
(T_3) $\langle$ `li R1, 0x1234` | $\sigma\rangle$ *vs.* $\langle$ `li R1, 0x1234` | $\sigma'\rangle$ for σ, σ', s.t. $\sigma(\texttt{R1}) \neq \sigma'(\texttt{R1})$, *i.e.*, for different initial values of R1.
(T_4) $\langle$ `li R1, 0x1234` | $\sigma\rangle$ *vs.* $\langle$ `div R1, R2, R3` | $\sigma\rangle$ for some σ.
(T_5) $\langle$ `div R1, R2, R3` | $\sigma\rangle$ *vs.* $\langle$ `div R1, R2, R3` | $\sigma'\rangle$ for σ, σ', s.t. $\sigma(\texttt{R1}) \neq \sigma'(\texttt{R1})$, $\sigma(\texttt{R2}) \neq \sigma'(\texttt{R2})$, and $\sigma(\texttt{R3}) \neq \sigma'(\texttt{R3})$. Further, $\sigma(\texttt{R3}), \sigma'(\texttt{R3}) \notin \{0, 1\}$, *i.e.*, neither of the initial states assigns 0 or 1 to R3.

For the attacker, all test cases except T_4 are indistinguishable, since the program executions take the same amount of time. In terms of template distinguishability, for instance, test case T_4 is distinguished by any contract that includes at least one of the template's atoms.

Phase 2a: Contract Synthesis: Based on the empirical leakage characterization, LeaSyn synthesizes, using integer linear programming, a candidate contract Ctr that (a) distinguishes all attacker-distinguishable test cases, and (b) is as precise as possible given the template $\mathbb{T}$. In other words, it distinguishes as few of the attacker-indistinguishable test cases as possible among all contracts in $\mathbb{T}$.

In our example, based on the initial test cases, LeaSyn might synthesize the following initial contract:

$$\textsc{Ctr}_1 = \{(\texttt{li}, \langle \texttt{"li"}, - \rangle)\}$$

This contract $\textsc{Ctr}_1$ captures that the attacker Atk can distinguish `li` instructions from `div` instructions (as indicated by T_4 above). Note that $\textsc{Ctr}_1$ is the most precise contract that captures all leaks exercised by the test cases. For instance, the contract exposing also the value of the immediate in `li` instructions, *i.e.*, $(\texttt{li}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$, would still capture the leak in T_4, but it also unnecessarily distinguishes T_1 and it is, therefore, not a possible solution.

Phase 2b: Bounded verification of soundness: Since the test cases may miss some leaks, the synthesized contract is not guaranteed to be sound. To ensure soundness, LeaSyn alternates the contract synthesis step with a bounded verification step, which either provides a counterexample to soundness or it proves that the contract is sound up to a given bound on the number of executed cycles. In case of unsoundness, the counterexample is used to refine the contract by looping back to the contract synthesis step (*Phase 2a*).

In our example, $\textsc{Ctr}_1$ is unsound since it does not capture the leaks generated by `div` instructions. The bounded verification might generate the following test case as a counterexample to soundness:

(T_6) $\langle$ `div R1, R2, R3` | $\sigma\rangle$ *vs.* $\langle$ `div R1, R2, R3` | $\sigma'\rangle$ for σ, σ', s.t. $\sigma(\texttt{R3}) = 0$ and $\sigma'(\texttt{R3}) = 5$.

Next, LeaSyn simulates the contract template $\mathbb{T}$ on the newly generated counterexample to determine which of the template's contracts would distinguish it. Then, it re-executes the contract synthesis step (*Phase 2a*). This time LeaSyn generates the following contract:

$$\textsc{Ctr}_2 = \{(\texttt{div}, \langle \texttt{"Reg[RS2]"}, [\![\texttt{Reg[RS2]}]\!] \rangle)\}$$

$\textsc{Ctr}_2$ captures that the value of the second source register influences the execution time of the `div` instruction, thereby distinguishing T_6. The presence of an atom that applies only to the `div` instruction also ensures that `div` instructions are distinguishable from `li` instructions, which distinguishes T_4.

Phase 3: Unbounded verification of soundness: As soon as the bounded verification does not find any more counterexamples, LeaSyn attempts to verify the synthesized contract unboundedly. To this end, LeaSyn generates inductive invariants using the Houdini algorithm, following the verification approach of Wang et al. [50]. If the verification succeeds, LeaSyn returns the synthesized sound and precise contract to the user. In our example, $\textsc{Ctr}_2$ can be proved sound in an unbounded manner and LeaSyn terminates.

2.4 Generating more precise contracts

LeaSyn synthesizes the most precise sound contract among all possible contracts given a specific contract template. The choice of the template, however, can affect the synthesized contracts' precision.

The sound contract $\textsc{Ctr}_2$ from §2.3 is imprecise. For instance, it distinguishes the test case below, even though the CPU Impl executes both instructions in the same number of cycles:

(T_7) $\langle$ `div R1, R2, R3` | $\sigma\rangle$ *vs.* $\langle$ `div R1, R2, R3` | $\sigma'\rangle$ for σ, σ', s.t. $\sigma(\texttt{R3}) = 7$ and $\sigma'(\texttt{R3}) = 8$.

However, LeaSyn cannot generate a more precise contract since the template from §2.3 can only expose the entire value of operands for `div` instructions.

To derive more precise contracts, we need a more expressive contract template, which can be achieved by including additional atoms of the following form, where k is a constant value:

$$(\texttt{div}, \langle \texttt{"Reg[RS2]=k?"}, [\![\texttt{Reg[RS2]}]\!] == k \rangle)$$

Running LeaSyn again with the updated template and test cases $T_1, \ldots, T_7$ may result in generating the following contract at the end of *Phase 2a*:

$$\textsc{Ctr}_2 = \{(\texttt{div}, \langle \texttt{"Reg[RS2]=0?"}, [\![\texttt{Reg[RS2]}]\!] == 0 \rangle)\}$$

Even though this contract captures one of the leaks caused by `div`, it is not sound since it misses the other special case of the `div` instruction, *i.e.*, when R2 is 1. When running the bounded verification (*Phase 2b*), LeaSyn might produce the following counterexample:

(T_8) $\langle$ `div R1, R2, R3` | $\sigma\rangle$ *vs.* $\langle$ `div R1, R2, R3` | $\sigma'\rangle$ for σ, σ', s.t. $\sigma(\texttt{R2}) = 1$ and $\sigma'(\texttt{R2}) = 5$.

A new synthesis step, then, finally results in the following contract:

$$\begin{aligned}\textsc{Ctr}_3 = \{&(\texttt{div}, \langle \texttt{"Reg[RS2]=0?"}, [\![\texttt{Reg[RS2]}]\!] == 0 \rangle),\\ &(\texttt{div}, \langle \texttt{"Reg[RS2]=1?"}, [\![\texttt{Reg[RS2]}]\!] == 1 \rangle)\}\end{aligned}$$

This contract is sound and, again, it is the most precise given the updated template. Therefore, LeaSyn terminates by returning $\textsc{Ctr}_3$ to the user after the final unbounded verification.

3 Formal model

Here, we present the core components of our formal model.

3.1 Architectures and microarchitectures

Leakage contracts act as a security abstraction between the instruction set architecture (short: architecture or ISA) and a microarchitecture, that is, a concrete implementation in a processor. Next, we formalize both concepts and what it means for a microarchitecture to correctly implement an architecture [50].

Architectures: We view an architecture as a state machine that defines how the execution progresses through a sequence of architectural states, where each transition corresponds to the execution of a single instruction. Formally, an *architecture* is a pair (ArchState, Isa) where ArchState is a set of architectural states (each modeling the values of registers and data/instruction memory) and Isa : ArchState → ArchState is a transition function that maps each state $\sigma \in$ ArchState to its successor Isa(σ), obtained by executing the next instruction to be executed in σ. To capture an execution, we denote by Isa$^*(\sigma)$ the sequence of states reached from σ by successive applications of Isa, *i.e.*, Isa$^*(\sigma) = \sigma_0, \sigma_1, \sigma_2, \ldots$ where $\sigma_0 = \sigma$ and $\sigma_{i+1} =$ Isa(σ_i) for all $i \geq 0$.

Microarchitectures: We view microarchitectures as state machines modeling how the processor's state evolves at cycle level. Thus, a *microarchitecture* is a triple (ImplState, InitState, Impl) where ImplState = ArchState $\times$ μArchState is the set of microarchitectural states (some of which are initial states InitState $\subseteq$ ImplState) and Impl : ImplState → ImplState is a transition function that maps each state $\sigma \in$ ImplState to its successor Impl(σ), obtained by executing the processor for one cycle. Each microarchitectural state $\sigma \in$ ImplState consists of an architectural part $\sigma_{\text{Arch}} \in$ ArchState and a microarchitectural part $\sigma_{\mu\text{Arch}} \in \mu$ArchState modeling the state of microarchitectural components (such as caches and predictors). For simplicity, we require that there is a canonical initial microarchitectural component $\mu_0 \in \mu$ArchState such that InitState $= \{(\sigma, \mu_0) \mid \sigma \in \text{ArchState}\}$. Similarly to Isa$^*(\sigma)$, Impl$^*(\sigma)$ denotes the sequence of states reached from σ by successive applications of Impl. Finally, given a predicate ϕ over ImplState, Impl$^*|\phi(\sigma)$ denotes the sequence of elements from Impl$^*(\sigma)$ that satisfy ϕ. For simplicity, we refer to a microarchitecture only using its transition function Impl rather than the tuple (ImplState, InitState, Impl). Similarly, we refer to an architecture (ArchState, Isa) simply using its transition function Isa.

ISA compliance: To correctly implement an architecture Isa, an implementation Impl needs to change the architectural state in a manner consistent with Isa. We capture this following the ISA compliance notion from Wang et al. [50], which relies on a *retirement predicate* ϕ that indicates when Impl retires instructions. Then, we say that a microarchitecture Impl implements an architecture Isa if one can map changes of the architectural state in Impl to Isa's executions using ϕ.

Formally, a microarchitecture Impl *correctly implements* an architecture Isa [50] given a retirement predicate ϕ over ArchState, written Impl $\vdash_\phi$ Isa, if for all states $\sigma \in$ InitState:

(a) $(\text{Impl}^*|\phi(\sigma))_{\text{Arch}} = \text{Isa}^*(\sigma_{\text{Arch}})$, *i.e.*, all architectural changes witnessed by ϕ agree with Isa, and

(b) $(\text{Impl}^i(\sigma))_{\text{Arch}} = (\text{Impl}^{i-1}(\sigma))_{\text{Arch}}$ whenever $(\text{Impl}^i(\sigma))_{\text{Arch}} \not\models \phi$, *i.e.*, no architectural changes beyond those witnessed by ϕ.

3.2 Leakage contracts

Next, we introduce how we model leakage contracts in LeaSyn. We first introduce *contract atoms*, which are the basic building block for contracts. Next, we introduce *contract templates*, which capture the synthesis space for our approach as a set of possible atoms.

Contract atoms: Atoms capture potential leaks at the instruction level, and they are built from *applicability predicates* and *leakage functions*. The former determine whether a contract atom is applicable in a given architectural state, whereas the latter determine what information from the architectural state is leaked in case an atom is applicable. Formally, a *contract atom* A is a pair $A = (\pi_A, \phi_A)$ where the applicability predicate π_A is a predicate over ArchState and the leakage function ϕ_A is a function ϕ_A : ArchState → CtrObs mapping an architectural state to a contract observation in CtrObs.

Example 1. Addition and subtraction instructions `addi` and `subi` could both leak their immediate value. Contracts atoms for these instructions can be defined as follows:

- $(\texttt{addi}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$, where predicate `addi` holds whenever the instruction is an addition instruction and $\langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle$ is a leakage function that leaks the immediate value of the instruction $[\![\texttt{imm}]\!]$ together with a leakage identifier `"imm"`.
- $(\texttt{subi}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$, where predicate `subi` holds whenever the instruction is a subtraction instruction and the leakage function is the same as above.

Note that $\langle \texttt{addi R1, R2, 0x1234} \mid \sigma \rangle$ *vs.* $\langle \texttt{subi R1, R2, 0x1234} \mid \sigma \rangle$ would be indistinguishable by a contract including both atoms. Even though two different instructions are executed, both atoms would be applicable and expose $\langle \texttt{"imm"}, \texttt{0x1234} \rangle$.

Contract templates: A *contract template* $\mathbb{T}$ is a set of contract atoms. We require that the applicability predicates of contract atoms with the same leakage function are mutually exclusive, *i.e.*, at most one of these atoms is applicable in any given state. This holds in Example 1, since instructions can either be additions or subtractions.

We also require that the images of different leakage functions are disjoint. This ensures that leakage from one function cannot "cancel out" leakage from another function. For instance, a leakage function exposing an immediate operand and another function exposing the instruction's opcode must not leak the same value. A simple way to satisfy this requirement is associating a unique identifier with each leakage function, like `"imm"` in Example 1.

Leakage contracts: Given a template $\mathbb{T}$, any subset $S \subseteq \mathbb{T}$ of the template induces a *contract* LC_S : ArchState → 2^{CtrObs}, which is a function from architectural states to sets of observations.[1] Formally, LC_S is defined by evaluating each applicable atom in S as follows: $LC_S(\sigma) := \{\phi_A(\sigma) \mid (\pi_A, \phi_A) \in S \wedge \pi_A(\sigma)\}$. Given a sequence $\tau := \sigma_0, \sigma_1, \ldots$ of architectural states, $LC_S(\tau)$ denotes the corresponding sequence of observations $LC_S(\sigma_0), LC_S(\sigma_1), \ldots$. Therefore, given an architectural state $\sigma \in$ ArchState, $LC_S(\text{Isa}^*(\sigma))$ denotes the *contract trace*, *i.e.*, the sequence of contract observations associated with the execution Isa$^*(\sigma)$.

[1] We often refer to a contract LC_S directly using the set of atoms inducing it.

Finally, in LeaSyn, a *test case* $T = (\sigma, \sigma')$ is a pair of architectural states. We say that T is *contract distinguishable* for given contract LC_S if the corresponding contract traces are different, *i.e.*, $LC_S(\text{Isa}^*(\sigma)) \neq LC_S(\text{Isa}^*(\sigma'))$.

3.3 Contract satisfaction

We conclude by formalizing contract satisfaction [30] in our setting. For this, we model microarchitectural attackers and then characterize when a contract captures all leaks observable by the attacker.

Attackers: We consider (passive) microarchitectural attackers that can extract information from the microarchitectural state. Formally, we model a microarchitectural attacker as a function $\text{Atk} : \text{ImplState} \rightarrow \text{AtkObs}$ that maps microarchitectural states to attacker observations in AtkObs. Common attacker models, like the one exposing the timing of instruction retirement [48] or the one exposing the final state of caches [23, 56], can be instantiated in this setting. Given an initial microarchitectural state $\sigma \in \text{InitState}$, $\text{Atk}(\text{Impl}^*(\sigma))$ denotes the *attacker trace*, *i.e.*, the sequence of attacker observations associated with the cycle-accurate microarchitectural execution $\text{Impl}^*(\sigma)$. We say that a test case $T = (\sigma, \sigma')$ is *attacker distinguishable* if the corresponding attacker traces are different, *i.e.*, $\text{Atk}(\text{Impl}^*(\sigma)) \neq \text{Atk}(\text{Impl}^*(\sigma'))$.

Contract Satisfaction: A microarchitecture Impl satisfies the contract LC_S for an attacker Atk if Atk cannot learn more information about the initial architectural state by monitoring Impl's executions than what is exposed by the contract. That is, for any two initial states, whenever the contract traces are the same, then the attacker traces must be identical, *i.e.*, Atk cannot distinguish the two architectural executions. This is formalized in Definition 1.

Definition 1. Microarchitecture Impl *satisfies* contract LC_S for attacker Atk, written $\text{Isa}, \text{Impl} \vdash \text{CtrSat}(LC_S, \text{Atk})$, if for all initial states $\sigma, \sigma' \in \text{InitState}$, if $LC_S(\text{Isa}^*(\sigma_{\text{Arch}})) = LC_S(\text{Isa}^*(\sigma'_{\text{Arch}}))$, then $\text{Atk}(\text{Impl}^*(\sigma)) = \text{Atk}(\text{Impl}^*(\sigma'))$.

Definition 1 refers to 4 different traces: two contract traces defined over the architecture Isa and two attacker traces defined over the microarchitecture Impl. Wang et al. [50] proposed the notion of microarchitectural contract satisfaction, formalized below, which is expressed only over a pair of microarchitectural traces and allows to decouple reasoning about leakage and about ISA compliance.

Definition 2. Microarchitecture Impl *microarchitecturally-satisfies* contract LC_S for attacker Atk and predicate ϕ, written $\text{Impl} \vdash_\phi \text{CtrSat}(LC_S, \text{Atk})$, if for all initial states σ, σ', if $LC_S(\text{Impl}^*|\phi(\sigma)) = LC_S(\text{Impl}^*|\phi(\sigma'))$, then $\text{Atk}(\text{Impl}^*(\sigma)) = \text{Atk}(\text{Impl}^*(\sigma'))$.

Note that in Definition 2, $LC_S(\text{Impl}^*|\phi(\sigma))$ denotes the trace obtained by applying LC_S to the architectural parts of all states in $\text{Impl}^*|\phi(\sigma)$. As stated in [50, Theorem 1], whenever Impl correctly implements Isa, then contract satisfaction and microarchitectural contract satisfaction are equivalent. Hence, Definition 2 is the notion targeted by LeaSyn and, for conciseness, we will refer to it simply as "contract satisfaction" throughout the rest of the paper.

4 Contract synthesis

In this section, we describe LeaSyn's synthesis approach. First, we describe the empirical leakage characterization (*Phase 1* in Fig. 1; §4.1). Next, we describe how LeaSyn synthesizes a candidate contract using integer linear programming (*Phase 2a* in Fig. 1; §4.2). Then, we describe how LeaSyn checks whether a contract is sound, *i.e.*, it capture all leaks, with bounded verification (*Phase 2b* in Fig. 1; §4.3) and unbounded verification (*Phase 3* in Fig. 1; §4.4). The proofs of all propositions are given in [51].

4.1 Phase 1: Empirical leakage characterization

LeaSyn generates a set of test cases T to characterize a processor's leakage. As an input to the ILP-based contract synthesis (*Phase 2a*), LeaSyn needs to determine for each test case: (a) whether it is *attacker distinguishable*, and (b) which contracts from the contract template would make it *contract distinguishable*.

Test-case generation: Test cases aid contract synthesis in two opposing ways: (1) Attacker-distinguishable test cases uncover leaks and force the inclusion in the contract of atoms that expose these leaks. (2) Attacker-indistinguishable test cases show which atoms should not be included in the contract to avoid imprecision.

Based on these observations, our goal is to generate test cases that differ in exactly one atom. This has two benefits. First, if a test case is attacker distinguishable, there is only one possible responsible atom. Second, as test cases "differ" only in one atom, this strategy is likely to generate many attacker indistinguishable tests, in particular when targeting atoms that do not capture leakage in the CPU.

To generate such test cases, we proceed as follows. We start by generating an architectural state σ_0 that consists of a random instruction sequence and a random initial valuation of the registers. Then, we randomly pick an instruction from the sequence and an applicable leakage function and generate a pair of architectural states (σ, σ') such that the leakage function will evaluate differently in σ and σ' on the chosen instruction.

To implement this, we define a *modifier function* for each leakage function in our template, similarly to [38]. Modifier functions may, for instance, change the initial value of a register, inject new instruction to adapt the architectural state, or modify an instruction.

Recall the template introduced in §2.3. Given this template, the modifier function for the leakage function `imm` would for example transform a state σ_0 whose first instruction is `li R1, 0x1234` to the pair (σ, σ') where $\sigma = \sigma_0$ and the only difference in σ' is that its first instruction is `li R1, 0x5678`. Similarly, the modifier function for the leakage function `RD` could substitute the instruction `div R1, R2, R3` in σ_0 with `div R4, R2, R3` in σ'. For other leakage functions such as `Reg[RS1]`, the modifier function does not alter any instructions but modifies the value of a specific register in the initial architectural state of σ'. Depending on the context of an instruction, modifier functions are not guaranteed to generate a test case that only differs in the chosen leakage function, *e.g.*, if dependent instructions are present in the instruction sequence.

Attacker distinguishability: Each of the test cases $t = (\sigma, \sigma')$ in T is executed on the target CPU Impl. To derive the attacker traces for σ and σ', we construct the two initial states (σ, μ_0) and (σ', μ_0) for the CPU Impl. To determine attacker indistinguishability, we then simulate the executions $\text{Impl}^*((\sigma, \mu_0))$ and $\text{Impl}^*((\sigma', \mu_0))$ and check whether the attacker can distinguish them, *i.e.*, we check if $\text{Atk}(\text{Impl}^*((\sigma, \mu_0))) \neq \text{Atk}(\text{Impl}^*((\sigma', \mu_0)))$. In general, the executions $\text{Impl}^*((\sigma, \mu_0))$ and $\text{Impl}^*((\sigma', \mu_0))$ might be infinite, so

LeaSyn simulates them up to a fixed number of cycles n, and we pick an n that is large enough to fully simulate all our test cases.

Template distinguishability: To succinctly characterize the contract distinguishability of each test case for all possible contracts induced by the template $\mathbb{T}$, we rely on the following observation.

Let $(\sigma, \sigma') \in T$ be a test case and let LC_S be a contract. Also, let $\text{ISA}^*(\sigma) = \sigma_0, \sigma_1, \ldots$ and $\text{ISA}^*(\sigma') = \sigma'_0, \sigma'_1, \ldots$ be the architectural executions from σ and σ'. Test case (σ, σ') can be contract distinguishable under LC_S for two distinct reasons:

- *Leakage mismatch:* There are atoms $A, B \in S$ with $\phi_A = \phi_B$ and an index i s.t. $\pi_A(\sigma_i) \wedge \pi_B(\sigma'_i)$ and $\phi_A(\sigma_i) \neq \phi_B(\sigma'_i)$, *i.e.*, at some point, the atoms A and B are applicable in the two executions but evaluate differently. Note that A and B can be the same atom.
- *Applicability mismatch:* There is an atom $A \in S$ and an index i s.t. $\pi_A(\sigma_i)$ (or $\pi_A(\sigma'_i)$) and no atom $B \in S$ with $\phi_A = \phi_B$ is applicable in σ'_i (or σ_i), *i.e.*, there is an observation only in one executions.

Example 2. Consider the following test cases:

- Test case `li R1, 0x1234` *vs.* `li R1, 0x5678` would be distinguishable due to a leakage mismatch by a contract including the atom $(\texttt{li}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$, since the immediate value exposed by the atom is different in the two executions.
- Test case `li R1, 0x1234` *vs.* `add R1, R2, R3` would be distinguishable due to an applicability mismatch by a contract consisting only of the atom $(\texttt{li}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$ as no atom with the same leakage function is applicable in the second execution.

We characterize the template distinguishability of each test case $t = (\sigma, \sigma')$ via its *strongly-distinguishing atoms* $SD_t \subseteq \mathbb{T}$ and its *xor-distinguishing pairs* $XOR_t \subseteq \binom{\mathbb{T}}{2}$:

A *strongly-distinguishing atom* $A \in \mathbb{T}$ is one for which there exists an index i such that $\pi_A(\sigma_i)$ (or $\pi_A(\sigma'_i)$) holds, and no atom $B \in \mathbb{T}$ satisfies $\pi_B(\sigma'_i) \wedge \phi_A(\sigma_i) = \phi_B(\sigma'_i)$ (or $\pi_B(\sigma_i) \wedge \phi_B(\sigma_i) = \phi_A(\sigma'_i)$). Including any strongly-distinguishing atom $A \in SD_t$ in a contract will make the test case t contract distinguishable independently of which other atoms are included. This contract distinguishability may be caused by a leakage mismatch *or* an applicability mismatch.

For example, in the test case `li R1, 0x1234` *vs.* `li R1, 0x5678` the atom $(\texttt{li}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$ is strongly distinguishing, as it is applicable in both executions and the immediate values are different.

A *xor-distinguishing atom pair* $\{A, B\} \in \binom{\mathbb{T}}{2}$ with $A \neq B$ is one for which there exists an index i such that $\pi_A(\sigma_i) \wedge \pi_B(\sigma'_i)$ holds and $\phi_A(\sigma_i) = \phi_B(\sigma'_i)$. Due to our assumption that the applicability predicates of contract atoms that share a leakage function are mutually exclusive, there can be no other atom $C \in \mathbb{T}$ with $\phi_A = \phi_B = \phi_C$ that is applicable in σ_i or σ'_i. Thus, including exactly one of the two atoms A and B in a xor-distinguishing pair $\{A, B\} \in XOR_t$ will make the test case t contract distinguishable due to an applicability mismatch. For example, in the test case `addi R1, R2, 0x1234` *vs.* `subi R1, R2, 0x1234` the pair of atoms $(\texttt{addi}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$ and $(\texttt{subi}, \langle \texttt{"imm"}, [\![\texttt{imm}]\!] \rangle)$ is a xor-distinguishing pair.

The two sets SD_t and XOR_t fully characterize the contract distinguishability of a test case t under any possible contract:

Proposition 1. *Let t be a test case and let LC_S be a contract. Then, t is contract distinguishable under LC_S iff*

- *S includes a strongly-distinguishing atom,* i.e., $SD_t \cap S \neq \emptyset$, *or*
- *S includes exactly one of the two atoms in a xor-distinguishing pair,* i.e., $\exists \{A, B\} \in XOR_t$ *with* $|S \cap \{A, B\}| = 1$.

To compute the sets SD_t and XOR_t for a test case t, we simulate the execution of the test case w.r.t. the contract template $\mathbb{T}$ and check the conditions for strongly-distinguishing and xor-distinguishing atoms for each index i. To simulate $\text{ISA}^*(\sigma)$ (which might be infinite), LeaSyn only simulates executions up to a fixed number of steps.

Relation to Mohr et al. [38]: Next, we discuss how our characterization of template distinguishability in terms of strongly-distinguishing atoms and xor-distinguishing pairs relates to the one introduced by Mohr et al. [38], which relies on the notion of *distinguishing atoms*. An atom $A \in \mathbb{T}$ is *distinguishing* [38] if the contract consisting solely of atom A distinguishes the test case, *i.e.*, if $LC_{\{A\}}(\text{ISA}^*(\sigma)) \neq LC_{\{A\}}(\text{ISA}^*(\sigma'))$. Equivalently, A is distinguishing if there exists an index i such that $\pi_A(\sigma_i) \oplus \pi_A(\sigma'_i)$ or $\pi_A(\sigma_i) \wedge \pi_A(\sigma'_i) \wedge \phi_A(\sigma_i) \neq \phi_A(\sigma'_i)$.

For templates that contain multiple atoms that share the same leakage function (like the templates implemented in LeaSyn and used in §6), the set of distinguishing atoms is *not* sufficient to fully characterize a test case's contract distinguishability (as opposed to our characterization, as indicated by Proposition 1). In particular, the inclusion of two distinguishing atoms may result in contract indistinguishability as Example 1 demonstrates.

In contrast, for templates where no two distinct atoms share the same leakage function, the set of distinguishing atoms is sufficient to characterize contract distinguishability and, therefore, our characterization is equivalent to the one from [38].

Proposition 2. *Let t be a test case and let LC_S be a contract. If no two distinct atoms share the same leakage function, then t is contract distinguishable under LC_S iff S includes a distinguishing atom for t.*

To summarize, the approach by Mohr et al. [38] cannot handle templates with atoms that share the same leakage function (in addition to synthesizing contracts that are not guaranteed to be sound). However, the templates used by LeaSyn *do* contain distinct atoms that share the same leakage functions and for this reason we require the more sophisticated characterization of contract distinguishability in terms of strongly-distinguishing atoms and xor-distinguishing pairs.

4.2 Phase 2a: ILP-based contract synthesis

Given a template $\mathbb{T}$, the empirical leakage characterization from §4.1 computes the following information for each test case $t \in T$:

- $SD_t \subseteq \mathbb{T}$ the set of strongly-distinguishing atoms,
- $XOR_t \subseteq \binom{\mathbb{T}}{2}$ the set of xor-distinguishing atom pairs, and
- $d_t \in \mathbb{B}$ whether the test case is attacker distinguishable.

For convenience, we denote the set of attacker-distinguishable test cases as $T_d = \{t \in T \mid d_t = 1\}$ and the set of attacker-indistinguishable test cases as $T_{nd} = T \setminus T_d$.

Based on this information, LeaSyn uses integer linear programming (ILP) to synthesize a contract. That is, we compute a set of atoms $S \subseteq \mathbb{T}$, such that LC_S distinguishes all attacker-distinguishable test cases and as few attacker-indistinguishable test cases as possible. Next, we detail this ILP formulation.

Variables: For each atom $A \in \mathbb{T}$, we introduce a boolean variable s_A that is true *iff* the atom A is included in the synthesized contract.

We also introduce a boolean variable fp_t for each $t \in T_{nd}$ to capture whether the attacker-indistinguishable test case t is a false positive, *i.e.*, it is contract distinguishable in the synthesized contract.

Finally, we introduce a boolean variable $x_{\{A,B\}}$ for each pair $\{A, B\}$ that is xor-distinguishing for some test case t.

Objective function: The objective function of the ILP then is to minimize the number of false positives: $\min \sum_{t \in T_{nd}} fp_t$. As a secondary objective we minimize the number of atoms in the contract, *i.e.*, we minimize $\sum_{A \in \mathbb{T}} s_A$, which promises to yield a smaller contract that is likely more precise on unseen test cases.

Constraints: It remains to ensure that all attacker-distinguishable test cases are contract distinguishable and that the fp_t variables are consistent with the contract encoded by the s_A variables. From Proposition 1, we know that a test case t is contract distinguishable *iff* the contract contains (a) a strongly-distinguishing atom or (b) exactly one of the two atoms of one of its xor-distinguishing pairs.

We start by adding the following constraints enforcing $x_{\{A,B\}} = s_A \oplus s_B$ for any xor-distinguishing pair $\{A, B\} \in \bigcup_{t \in T_d} XOR_t$:

$$x_{\{A,B\}} \leq s_A + s_B, \qquad x_{\{A,B\}} \geq s_A - s_B,$$
$$x_{\{A,B\}} \geq s_B - s_A, \qquad x_{\{A,B\}} \leq 2 - s_A - s_B.$$

Following Proposition 1, we add the following constraint for every $t \in T_d$ to ensure that every attacker-distinguishable test case is contract distinguishable:

$$\sum_{A \in SD_t} s_A + \sum_{\{A,B\} \in XOR_t} x_{\{A,B\}} \geq 1. \tag{1}$$

Finally, to ensure that the fp_t variables are consistent with the contract distinguishability of the test cases we add the following constraints for each attacker-indistinguishable test case $t \in T_{nd}$:

$$fp_t \geq s_A \qquad \text{for every } A \in SD_t,$$
$$fp_t \geq x_{\{A,B\}} \qquad \text{for every } \{A, B\} \in XOR_t.$$

Proposition 3. *Any solution to the ILP corresponds to a set of atoms $S \subseteq \mathbb{T}$ such that LC_S distinguishes all attacker-distinguishable test cases and as few attacker-indistinguishable test cases as possible.*

Mohr et al. [38] employ a similar ILP for contract synthesis. As discussed in §4.1, their work lacks the notion of xor-distinguishing atoms, which is needed to support contracts where multiple atoms share the same leakage function. As a consequence, their ILP differs from ours in two ways: (a) It omits all constraints involving the $x_{\{A,B\}}$ variables. (b) To ensure that every attacker-distinguishable test case is contract distinguishable, it includes the following constraint for every $t \in T_d$: $\sum_{A \in D_t} s_A \geq 1$, where D_t is the set of distinguishing atoms for test case t.

4.3 Phase 2b: Bounded soundness verification

The ILP-based synthesis from §4.2 generates a new contract based on the attacker and template distinguishability derived from the test cases (§4.1). Since the test cases may miss some leaks in the target processor, the contract generated from *Phase 2a* may be unsound. To discover leaks that are missed by the candidate contract, LeaSyn performs a bounded verification step. This is another key difference between LeaSyn and the approach by Mohr et al. [38] (beyond those outlined in §4.1–4.2): while the latter simply synthesizes the most precise contract from an initial set of test cases, without any soundness guarantees, LeaSyn employs bounded verification to discover missed leaks, as we describe next, and provides soundness guarantees (as we show in §4.4).

Intuition: LeaSyn verifies whether the candidate contract LC_S captures all leaks visible by the attacker Atk on the target CPU Impl when considering all possible executions up to a given length k. That is, LeaSyn verifies a bounded variant of Definition 2. For this, LeaSyn encodes the contract verification task as a bounded model checking (BMC) problem. Since microarchitectural contract satisfaction is a property defined over pairs of Impl executions, our encoding relies on self-composition [15], which reduces reasoning about pairs of executions to reasoning about a single execution, by first constructing a product circuit consisting of two copies of Impl. The bounded contract satisfaction property ϕ_{ctrsat} is then encoded on top of this circuit. The BMC either (a) falsifies ϕ_{ctrsat} and provides a counterexample—a test case with the same contract traces but different attacker traces—which can be used to synthesize a better contract, or (b) proves that the contract is sound up to bound k.

Constructing the product circuit: We construct a product circuit consisting of two copies of Impl executing in parallel. To compare contract observations, which are produced only when instructions retire (*i.e.*, whenever the user-provided retirement predicate ψ holds), we need to synchronize the two copies of Impl on retirement. That is, whenever one copy of Impl retires an instruction but the other does not (*i.e.*, ψ holds only on one of the two executions), the copy of Impl where ψ holds is "paused" until the other one catches up. To do so, LeaSyn uses the stuttering product circuit construction introduced by Wang et al. [50], which constructs a product circuit synchronized on the retirement predicate ψ.

Property: We encode microarchitectural contract satisfaction on top of the product circuit into the $\phi^{C,\psi}_{ctrsat}$ property, where C is the contract and ψ is the retirement predicate, as follows. Our encoding is parametric in: (a) the total bound k, and (b) the attacker bound $1 \leq b \leq k$. Intuitively, the formula $\phi^{C,\psi}_{ctrsat}$ works as follows:

(1) it assumes that, at cycle 0, the two copies of Impl start from a valid initial state and with the same microarchitectural part,

(2) it assumes that, for cycles 0 to k, the two executions produce the same contract traces, *i.e.*, whenever the retirement predicate ψ holds on both executions, the contract observations are the same,

(3) it asserts that, for cycles 0 to b, the two executions have the same attacker observations, *i.e.*, they are attacker-indistinguishable.

Note that the attacker bound b is less than the total bound k, since contract observations are "slower" than attacker observations, as the former are produced only when instructions retire. This requires looking ahead to check whether future contract observations may "declassify" a difference in attacker observations. Since these differences are often caused by in-flight instructions, $k - b$ needs to be large enough to account for the retirement of these instructions to ensure that the associated observations are produced. However, sizing $k - b$ to the worst-case results in large bounds that (a) slow down checking contract satisfaction in simpler cases, and (b) may result in counterexamples with many instructions.

To account for this, LeaSyn uses a simple optimization, which is parametric in an instruction bound i. As soon as a difference in attacker observations is detected (in the first b cycles), we start

decrementing i whenever an instruction is retired by both executions. LeaSyn then terminates the BMC as soon as either i reaches 0 or the cycle bound k is reached, whichever comes first. This optimization allows us to dynamically adjust the number of cycles explored by the BMC. As we show in §6.2, this has two benefits: (a) it speeds up bounded verification since the BMC finds many simple counterexamples faster, without having to always explore all k cycles, and (b) it often results in shorter counterexamples with fewer instructions, which restrict the synthesizer's search space. The encoding of the property checked by LeaSyn in terms of `assume` and `assert` statements in Verilog is given in [51].

Proposition 4. *Let $\mathbb{K}$ be the maximum number of cycles that the CPU under verification takes to retire an instruction. If the BMC falsifies $\phi^{C,\psi}_{ctrsat}$, with retirement predicate ψ, BMC bound k, attacker bound $1 \leq b \leq k$, and instruction bound i, then there is an attacker-distinguishable test case T where the prefixes of length $min(\left\lfloor \frac{k}{\mathbb{K}} \right\rfloor, i)$ of the C-traces are identical.*

From a falsification of $\phi^{C,\psi}_{ctrsat}$ we extract a synthetic attacker-distinguishable test case, which is then used to synthesize the next contract candidate via the ILP introduced in §4.2. To this end, we determine the test case's set of strongly-distinguishing atoms and xor-distinguishing pairs in the prefix of length $min(\left\lfloor \frac{k}{\mathbb{K}} \right\rfloor, i)$. This process is repeated until the BMC does not discover any more unaccounted leaks. Each iteration of the loop in Phase 2 rules out at least one of the template's contracts, and thus eventually the BMC proves $\phi^{C,\psi}_{ctrsat}$ and the synthesis loop terminates.

Proposition 5. *Let $\mathbb{K}$ be the maximum number of cycles that the CPU under verification takes to retire an instruction. If the BMC proves $\phi^{C,\psi}_{ctrsat}$, with retirement predicate ψ, BMC bound k, attacker bound $1 \leq b \leq k$, and instruction bound i, then any C-indistinguishable test case T consisting of at most $min(\left\lfloor \frac{k}{\mathbb{K}} \right\rfloor, i)$ instructions is also attacker indistinguishable for the first b cycles.*

Proposition 5 formalizes what we refer to as a boundedly-sound contract in the overview in Figure 1.

4.4 Phase 3: Unbounded soundness verification

As soon as Phase 2b cannot discover further counterexamples, LeaSyn performs an unbounded verification step to ensure that the synthesized contract is *sound*, *i.e.*, all possible leaks in the target processor are captured by the contract.

For this, LeaSyn uses the LeaVe contract verifier [50] to prove that (unbounded) microarchitectural contract satisfaction holds for the current contract and the target CPU. In a nutshell, LeaVe verifies unbounded contract satisfaction by (a) learning the strongest possible inductive invariant over pairs of contract-indistinguishable executions, and (b) use this invariant to prove that contract-indistinguishable executions are also attacker-indistinguishable.

If LeaVe's verification passes, the synthesized contract is sound (from [50, Theorem 2]) and as precise as possible (from Proposition 3) for the target processor. If it fails, LeaSyn terminates and notifies the user. This failure can be due to two reasons:

- If the contract is unsound, the user can re-run LeaSyn with a larger bound for Phase 2b to identify the missed leaks.
- If the contract is sound but LeaVe fails to verify it, the user can provide additional information to LeaVe (e.g., additional relational invariants) to make the unbounded proof go through [50].

Proposition 6. *If Phase 3 passes, the contract synthesized by LeaSyn is sound and distinguishes as few attacker-distinguishable test cases as possible among the test cases explored during the synthesis process.*

5 Implementation

In this section, we present LeaSyn, a prototype that implements our synthesis approach from §4 for CPU designs in Verilog. Our prototype relies on the Yosys Open Synthesis Suite [10] for processing Verilog circuits, the Icarus Verilog simulator [5] for simulating test cases, the Google OR-Tools [2] for the ILP, the Yices SMT solver [9] for the bounded verification, and the LeaVe contract verification tool [50] for the unbounded verification. LeaSyn's implementation is open-source and available at [8].

Inputs: LeaSyn takes as input (1) the target processor Impl implemented in Verilog, (2) a Verilog expression e over Impl expressing the retirement predicate, (3) a Verilog expression Atk over Impl modeling the microarchitectural attacker, (4) the bounds for the bounded verification, and (5) the additional inputs for the unbounded verification in LeaVe (e.g., relational invariants). The user can also provide expressions for initializing parts of Impl's state.

Contract templates: LeaSyn implements multiple contract templates (described in §6.1) that capture different classes of instruction-level leaks. Even though these templates are defined in terms of the RISC-V ISA, they need to be instantiated for each new target processor since different processors often implement the same architectural elements (e.g., the register file) with different signals in Verilog. To address this, all our templates and atoms (across the entire LeaSyn's pipeline) are implemented in terms of the RISC-V formal interface [6] (RVFI), which provides a common interface to the architectural state of a RISC-V core for testing and verification. This interface, which is implemented by several open-source cores, significantly simplifies applying LeaSyn to a new CPU that already implements the RVFI.

Test generation: LeaSyn can accept user-provided test cases as input. Additionally, LeaSyn implements a test generation strategy for automatically synthesizing a given number of test cases. For each leakage function in the implemented templates, we also implement the associated modifier functions, which LeaSyn uses to generate test cases. LeaSyn's test generation strategy follows the process described in §4.1: (a) we first randomly generate an architectural state σ (consisting of a RISC-V program and initial values for registers), (b) we identify one of the instructions in the program, and (c) we use one of the applicable modifier functions for the selected instruction to generate the other initial state σ'.

Workflow: LeaSyn implements the workflow described in §4. First, it generates a set of test cases, simulates them, and compute their attacker and template-distinguishability w.r.t. the attacker and selected template. Then, it alternates between synthesis and bounded verification to come up with precise candidate contracts. Finally, it verifies whether the candidate contract is sound with the LeaVe (unbounded) contract verifier and terminates.

6 Evaluation

In this section, we report on the use of LeaSyn to synthesize sound and precise leakage contracts. We first introduce the methodology we followed for our evaluation (§6.1): the processors we analyze, the contract templates and attacker we consider, and the experimental setup. In our evaluation (§6.2), we address these research questions:

- **RQ1:** Can LeaSyn synthesize sound and precise leakage contracts from processor designs at RTL?
- **RQ2:** What is the impact of the number of initial test cases on the synthesized contract's precision and on the synthesis time?
- **RQ3:** What is the impact of different contract templates on the synthesized contract's precision?
- **RQ4:** Does LeaSyn synthesize contracts that are more precise than leakage contracts derived in prior work?
- **RQ5:** What is the impact of different property encodings on the verification time of Phase 2b (§4.3)?

We remark that all benchmarks and scripts for reproducing our results are available at [8].

6.1 Methodology

Benchmarks: We analyze the following CPUs:

- **DarkRISCV**: An in-order RISC-V core that implements the RV32I instruction sets [1]. We analyzed the 2-stage **DarkRISCV-2** and the 3-stage **DarkRISCV-3** implementations.
- **Sodor-2**: An educational 2-stage RISC-V core [7] that implements the RV32I instruction set.[2]
- **Ibex**: An open-source, production-quality 32-bit RISC-V CPU core [3].[3] We target the default "small" configuration, which has two stages and supports the RV32IM instruction set. We study three variants: (1) **Ibex-small** is the default "small" configuration with constant-time multiplication (three cycles) and without caches, (2) **Ibex-cache** is **Ibex-small** extended with a simple (single-line) cache, and (3) **Ibex-mult-div** employs a non-constant-time multiplication unit whose execution time depends on the operands [4]. For all variants, compressed instructions are disabled.

Contract templates: We consider the following basic templates:

- **Instruction Leaks (I)**: atoms exposing information about an instruction's encoding (op code, destination and source registers, immediate values).
- **Register Leaks (R)**: atoms exposing the value of an instruction's source and destination registers, and the program counter.
- **Memory Leaks (M)**: atoms exposing the accessed memory address and content read from/written to memory.
- **Alignment Leaks (A)**: two atoms that expose the alignment of a memory access: *IS_ALIGNED* exposes whether the last two bits of the memory address are 00 and *IS_HALF_ALIGNED* exposes whether the last two bits of the memory address are not 11.
- **Branch Leaks (BT)**: atoms that expose the outcome of branch instructions. For conditional jumps, the atoms expose if the branch is taken. For unconditional ones, the atoms return the constant 1.
- **Value Leaks (V)**: atoms exposing selected information about values of source and destination registers: (a) whether these values are equal to 0, and (b) the logarithm of the value. These atoms are inspired by known value-dependent leaks in the **Ibex** core [38, 50].

Additionally, we consider templates obtained by combining the basic ones listed above, and we denote such combinations with +. In particular, the **Base Leaks (B)** template consists of **I + R + M**, that is, the template containing all atoms from **I**, **R**, and **M**.

Attacker: We consider an attacker that observes when instructions retire, *i.e.*, when the retirement predicate holds.

Unbounded verification setup: For the unbounded verification using LeaVe, for all benchmarks, we re-use the initial-state invariants and the additional candidate inductive invariants manually constructed by Wang et al. [50] for verifying the same cores.

Experimental setup: All our experiments ran on an AMD Ryzen Threadripper PRO 5995WX with 64 cores and 512 GB of RAM. Each experiment ran in a dockerized Ubuntu 24.04 with OpenJDK 21. LeaSyn used Google OR-Tools version 9.10, Yosys version 0.48, Icarus Verilog version s20250103, and Yices version 2.6.5.

Evaluating precision: Leakage contracts classify pairs of executions as distinguishable or indistinguishable. To measure the precision of synthesized contracts, we use the standard notion of precision used for binary classifiers, following [38]. Given a target processor, a contract C, and a validation set V of test cases, the precision of C with respect to V is $\frac{TP}{TP+FP}$, where TP is the number of test cases that are contract and attacker distinguishable and FP is the number of test cases that are contract distinguishable but attacker indistinguishable. For each benchmark, we generate a validation set consisting of $2048K$ test cases.

6.2 Experimental results

RQ1 – Synthesis for open-source RISC-V cores: To evaluate whether LeaSyn can synthesize sound and precise leakage contracts from open-source cores, we apply LeaSyn to all benchmarks from §6.1. For each core, we use LeaSyn to synthesize a contract against the **B + A + BT + V** template with a set of $128K$ test cases.

Table 1 summarizes the results of our evaluation. The table reports (a) the number of iterations for the synthesis loop, (b) the static BMC bound k, the attacker bound b and the instruction bound i used, (c) the total synthesis time split between Phase 1, Phase 2a, Phase 2b, and Phase 3, (d) the number of atoms in the final synthesized contract, and (e) the precision of the synthesized contract.

We highlight the following findings:

- LeaSyn successfully synthesizes sound contracts for all target CPUs, and all contracts pass the unbounded verification phase.
- The synthesized contracts have precision higher than 0.99 for all targets except for **Ibex-cache**. The low precision in the **Ibex-cache** case is due to the core having a single-line cache, which leaks whether a given address has been accesses consecutively or not. This leak cannot be precisely captured in the **B+A+BT+V** template, and LeaSyn falls back to exposing the values of all memory accesses.
- For all cores, LeaSyn can synthesize a contract that passes the bounded verification (without Phase 3) in less than 5 hours. In several cases, the unbounded verification of the contract using LeaVe [50] dominates the overall execution time.
- For **DarkRISCV-2**, **DarkRISCV-3**, and **Sodor-2**, LeaSyn can synthesize a sound contract (each one with 8 atoms) directly from the initial test cases. For **Ibex**, instead, LeaSyn needs to iterate between Phases 2a and 2b, before converging on the final contract.

[2] We translate Sodor (written in Chisel) into Verilog for the analysis with LeaSyn.
[3] We translate Ibex (written in SystemVerilog) into Verilog for the analysis with LeaSyn.

Table 1: Results of LeaSyn's synthesis campaign.

Processor	*Iterations*	*BMC bound*	*ATK bound*	*Instr. bound*	*Synthesis time* (hh:mm:ss) *Phase 1*	*Phase 2a*	*Phase 2b*	*Phase 3*	*# atoms*	*Precision*
DarkRISCV-2	1	15	10	1	00:02:41	00:00:17	01:22:44	43:16:44	8	0.998
DarkRISCV-3	1	20	15	1	00:03:14	00:00:21	04:52:56	18:51:26	8	0.999
Sodor-2	1	6	5	1	00:04:38	00:00:09	00:13:31	03:01:59	8	0.998
Ibex-small	4	51	12	1	00:07:08	00:02:09	02:38:33	22:09:32	24	0.999
Ibex-cache	24	51	12	1	00:07:18	00:14:45	02:11:28	21:43:59	27	0.579
Ibex-mult-div	2	51	12	1	00:07:32	00:01:12	03:28:17	24:38:21	24	0.999

Furthermore, the synthesized contracts contain more atoms than those for **DarkRISCV** and **Sodor-2**, which is consistent with **Ibex** being more complex than the other benchmarks.

- While **Sodor-2**, **DarkRISCV-2**, and **DarkRISCV-3** only leak through control-flow operations, contracts for the **Ibex** cores capture additional leaks. The contracts for all **Ibex** cores expose: (a) if the second operand of `div`, `divu`, `rem`, `remu` instructions is 0, (b) if memory accesses are aligned or not (only for stores for **Ibex-cache**), (c) if the core is executing a memory read or write, and (d) if the core is executing a `mul`, `mulh`, `mulhu`, or `mulhsu`. Furthermore, the contract for **Ibex-cache** exposes the entire address of memory loads (capturing the cache leak), whereas the contract for **Ibex-mult-div** exposes the logarithm of the second operand for `mul` instructions (capturing the leak introduced by the slow-multiplication unit).

RQ2 – Impact of the number of test cases: LeaSyn uses test cases to generate the candidate contract. To evaluate the impact of the number of test cases on the synthesis process, we use LeaSyn to synthesize a contract for all our benchmarks using the **B+A+BT+V** template and different number of test cases (from 0 to 2048K).

Figure 2 reports the results of our experiments. In particular, Figure 2a reports the execution time (excluding the unbounded verification)[4] needed to synthesize a sound contract for the different configurations, Figure 2b reports the precision of the synthesized contract, and Figure 2c reports the fraction of time spent on synthesizing the first contract candidate, which includes all of Phase 1 and the first call to the ILP solver (Phase 2a), relative to the total execution time (excluding unbounded verification).

We highlight the following findings:

- A very small number of initial test cases often results in a high synthesis time. The reason for this is that small sets of initial test cases often exercise only a limited behavior of the target processor, which results in a greater number of iterations of the synthesis loop.
- Increasing the number of test cases is beneficial as long as the simulation time required to find a new leak is shorter than the time required to find a new counterexample via bounded verification. At around 1024k/2048k test cases the probability of finding new leaks via simulation is low and the simulation time starts to dominate.
- A very small number of test cases often results in low precision contracts. This is because Phase 2a (§4.2) relies on attacker-indistinguishable test cases to optimize the precision of the synthesized contract. Small sets of test cases do not provide enough information to guide synthesis and often leads to imprecise contracts.
- Increasing the number of test cases usually increases the precision of the synthesized contract, even though there are cases where the contract gets worse with more test cases. For all cases except **Ibex-cache**, the precision of the synthesized contract stabilizes at 1 quickly (at most after reaching 32K test cases). For **Ibex-cache**, the precision stabilizes at 0.579. This is due to the limitations of the template. All synthesized contracts for **Ibex-cache** expose the accessed memory address (to capture cache leaks), even though **Ibex-cache** implements a single-line cache so the actual leak only exposes whether a given address has been accessed consecutively or not, which cannot be captured in our template.

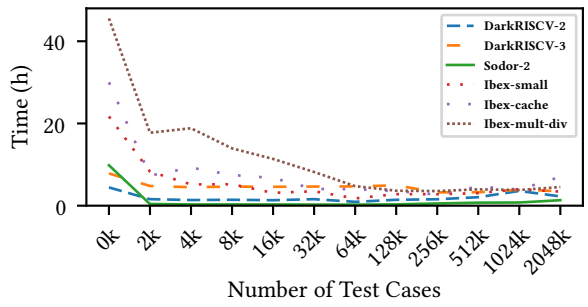
RQ3 – Impact of contract templates: The user-provided contract template defines the search space explored by LeaSyn. To study the impact of different templates, we use LeaSyn to synthesize a contract with increasingly more expressive templates and measure the contract's precision. For each core and template, we run LeaSyn with the same bounds as in Table 1 and with 64K test cases.

Figure 3 summarizes our results. We highlight these findings:

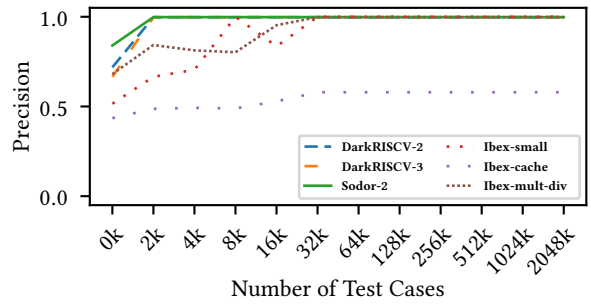
- Template **B** = **I**+**R**+**M** is the first template for which LeaSyn can synthesize sound contracts for all target CPUs. For templates **I**, **R**, **M** in isolation, LeaSyn cannot synthesize a sound contract, since no sound contract exists in these templates.
- **B** → **B** + **A** : For the **Ibex** cores, additionally considering template **A** results in more precise contracts. Executing aligned and non-aligned memory accesses take different times. In template **B**, this can only be captured by exposing the entire memory address. In **B** + **A**, LeaSyn synthesizes a more precise contract that exposes only whether memory accesses are aligned or not.
- **B+A** → **B+A+BT** : Additionally considering the **BT** template increases precision. The branch-taken and branch-non-taken cases have different execution times. Without **BT**, LeaSyn is forced to synthesize contracts that expose the values of the registers used to compute the branch. With **BT**, LeaSyn synthesizes more precise contracts that only expose whether the branch is taken.
- **B+A+BT** → **B+A+BT+V** : For the **Ibex** cores, additionally considering the template **V** further increases precision. These cores rely on a multiplication/division unit where (a) the execution time of multiplications is proportional to the logarithm of one of its operands, and (b) there is a fast path for division if the divisor equals to 0. With **V**, LeaSyn can synthesize contracts that characterize such leaks, rather than having to expose entire operands.

RQ4 – Comparison with contracts from prior work: Prior work have derived leakage contracts for some RISC-V cores. We focus on contracts from LeaVe [50], Mohr et al. [38], ConjunCT [21], VeloCT [22], and RTL2MμPATH [33] for the **Ibex-small** CPU and we compare them (in terms of precision) to the contract synthesized by LeaSyn using **B** + **A** + **BT** + **V** as template, an initial set of 64K test cases, and the bounds from Table 1. Like LeaSyn, [21, 22, 33, 38, 50]

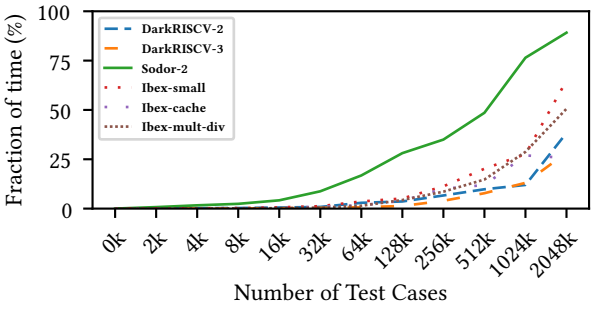
[4]We omit the unbounded verification time because (a) it is roughly independent on the number of test cases and (b) it often dominates the rest of the process.



(a) Synthesis time (excluding unbounded verification)



(b) Precision of the synthesized contract



(c) Fraction of time spent on synthesizing the first contract candidate (*i.e.*, Phase 1 and one iteration of Phase 2a) relative to the total synthesis time (excluding unbounded verification)

Figure 2: Impact of the number of test cases on contract synthesis

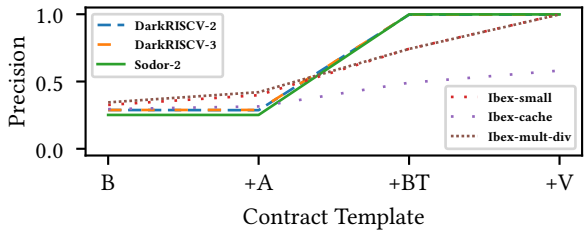


Figure 3: Impact of template expressiveness on precision

target an attacker that observes retirement time. Both LeaVe [50] and ConjunCT [21] have been applied to **Ibex-small**, and we consider the contracts reported in the corresponding papers. For Mohr et al. [38], we consider a contract synthesized with 64*K* initial test cases. VeloCT [22] (which is an improvement over ConjunCT) lacks an evaluation against **Ibex-small** (and it has only been applied to Boom in [22]); we contacted the paper's authors who confirmed that VeloCT produces the same contract as ConjunCT for Ibex, therefore for VeloCT we use the same contract as for ConjunCT. Finally, for RTL2MμPATH [33] (which also lacks an evaluation against **Ibex-small** and is built on top of the JasperGold commercial verification tool), we consider the most precise contract synthesized by LeaSyn for **Ibex-small** against a template capturing RTL2MμPATH's contracts, *i.e.*, a template that includes atoms that leak the operands of instructions and forces a leakage of the instruction word and the program counter.

We measure the precision of the four contracts against two different validation set: **different-programs** and **same-programs**. The **different-programs** validation set consists of 2048K test cases where the two programs in each test case might be different, which we generate with LeaSyn's default test-generation strategy. The **same-programs** validation set consists of 2048K test cases where the two programs in each test case are the same and only the initial inputs differ. We consider also **same-programs** since [21, 22, 33, 50] assume that the program itself is public and known to the attacker.

Table 2 summarizes the precision of all contracts. The contract generated by LeaSyn is significantly more precise than those from [21, 22, 33, 50] when considering the **different-programs** validation set. This is unsurprising as the contract templates underlying [21, 22, 33, 50] treat as contract distinguishable any two executions involving different programs/instruction streams. However, the contract produced by LeaSyn is also more precise than the ones from [21, 22, 33, 50] under the **same-programs** validation set. This is due to the fact that [21, 22, 33, 50] consider coarser templates than **B** + **A** + **BT** + **V**, which require them to expose more information than needed to capture the actual leaks in **Ibex-small**.

Finally, we compare the sound contracts from LeaSyn with the (potentially unsound) ones produced by the approach from Mohr et al. [38]. For this, we synthesize contracts using LeaSyn and the approach from [38] for **Ibex-small** with different numbers of initial test cases (from 1K to 2048K). For all synthesized contracts, we (a) measure precision against the **different-programs** validation set, and (b) for the contracts from [38] we also checked their soundness.

Table 3 reports the soundness/precision of all synthesized contracts. The contracts produced by LeaSyn are more precise (except for those produced with 1K and 2K initial test cases) than those produced by Mohr et al. [38]. In particular, with the maximum number of test cases (2048k), Mohr et al.'s approach reaches a precision of 0.69 whereas LeaSyn's contract attains a precision of 0.999. None of the contracts generated by the approach from Mohr et al. [38] is sound whereas, as expected, all contracts from LeaSyn are sound.

RQ5 – Impact of property encoding on bounded verification: We use LeaSyn to synthesize contracts for the **Ibex-small** core with **B** + **A** + **BT** + **V** as template. We implement and compare two encodings: the base one (denoted **b**) where the BMC always explores up to 51 cycles (same bound as in Table 1), and the optimized one (denoted **o** and described in §4.3) where the BMC stops the exploration as soon as: (a) one instruction retires after observing a difference on the attacker trace, or (b) it reaches 51 cycles.

Table 4 summarizes the results. We highlight these findings:

Table 2: Comparison of precision with contracts from previous works [21, 22, 33, 38, 50]

	LeaVe [50]	Mohr et al. [38]‡	ConjunCT [21]	VeloCT [22]	RTL2MμPATH [33]	Our work
same-programs	0.774	0.855	0.544	0.544	0.616	1.000
different-programs	0.054	0.784	0.056	0.056	0.064	0.999

‡ Unsound

Table 3: Comparison of precision and soundness with [38]

	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1024K	2048K
Mohr et al. [38]	0.706‡	0.738‡	0.697‡	0.733‡	0.750‡	0.781‡	0.784‡	0.784‡	0.784‡	0.784‡	0.699‡	0.693‡
Our work	0.515†	0.667†	0.704†	0.999†	0.835†	0.999†	0.999†	0.999†	0.999†	0.999†	0.999†	0.999†

† Sound ‡ Unsound

Table 4: Impact of property encoding on bounded verification

# test cases		2K	4K	8K	16K	32K	64K	128K	256K
Time (hh:mm)	**o**	8:18	5:10	5:12	3:07	3:30	1:46	2:48	2:47
	b	19:27	9:37	7:39	6:19	3:58	2:32	1:41	2:43
Iterations	**o**	61	40	27	13	10	7	3	3
	b	110	48	37	33	15	9	2	3
avg. # instr. in cex	**o**	1.1	1.3	1.1	1.2	1.2	1.3	1.7	1.7
	b	5.8	4.4	5.4	5.3	10.0	7.6	3.0	7.0

- The optimized encoding results in faster (bounded) synthesis. However, the speedup is more significant with a lower number of initial test cases. Part of this speedup is due to a reduced number of iterations along the synthesis loop.
- The optimized encoding results in shorter counterexamples, *i.e.*, programs with fewer instructions. Intuitively, shorter counterexamples are better since they reduce the synthesizer's search space.

7 Discussion

Verification bounds: The choice of bounds for the bounded verification step (Phase 2b) can affect the success of the synthesis process and its runtime. On the one hand, too short a bound can result in a candidate contract that passes the bounded verification but fails the final unbounded verification step (Phase 3), since the bounded verification failed in discovering some leaks. On the other hand, too large a bound can significantly slow down the entire synthesis process, in particular when the synthesis loops requires many iterations. In practice, in our experiments starting with a total bound of $i * k$ and an instruction bound of $i - 1$, where k is the maximum number of cycles for retiring an instruction and i is the maximum number of in-flight instructions, was sufficient to successfully synthesize the contracts.

Limitations on supported contracts: Our formal model (§3) is restricted to *sequential* leakage contracts that only capture leaks generated by architectural instructions. Using the terminology of [30, 40], LeaSyn only synthesizes "leakage clauses" for the sequential execution model. Furthermore, given that the leakage functions in our atoms map single architectural states to contract observations, LeaSyn is geared towards synthesizing contracts for instruction-level leaks, *i.e.*, those leaks that can be attributed directly to a specific instruction rather than to a sequence of instructions. We leave the synthesis of other classes of contracts, e.g., those capturing speculative leaks [30, 31], as future work.

Dependence on user-provided contract atoms: Our approach critically relies on user-provided contract atoms that are able to capture the leakage of the target processor soundly and precisely. For non-speculative leaks, it is fairly straightforward to define atoms able to soundly capture a processor's leakage, by exposing any information an instruction may depend on. It is less obvious how to define atoms able to capture leakage precisely, as this requires understanding the actual leakage of the processor. It is future work to investigate how to automatically derive such atoms.

Termination of the synthesis loop: The contract synthesis algorithm generates a sequence of candidate contracts in a non-monotonic fashion: each new candidate need not be a superset of the previous one. For instance, $\textsc{Ctr}_2$ from §2.1 is incomparable to $\textsc{Ctr}_1$. Nevertheless, the synthesis loop is guaranteed to terminate, since each counterexample rules out at least one incorrect candidate, and the number of possible contracts given a template is finite.

(Non-)determinism of synthesis: For a fixed set of test cases, there may be multiple incomparable sound contracts that have the same optimal precision. Beyond this inherent non-determinism, however, the synthesis process is deterministic: While the bounded verification step may generate different counterexamples in different runs, the algorithm will always converge to one of the sound contracts that are optimal with respect to the test cases generated in Phase 1.

Leakage contracts and secure programming: Leakage contracts provide the foundations for writing leak-free software. As shown by Guarnieri et al. [30], ensuring that program-level secrets do not influence contract traces is sufficient to ensure the absence of microarchitectural leaks for any CPU satisfying the contract.

In our framework, a contract LC_S is defined via a set S of atoms (π_A, ϕ_A), where ϕ_A indicates what is exposed on the contract trace whenever π_A holds. Different atoms may share the same leakage function, *i.e.*, it is possible that $\phi_A = \phi_B$ for $A \neq B$, but different leakage functions must have disjoint images (§3.2). To guarantee leak freedom at program level, following [30], one needs to ensure that the sequence of contract observations produced by the program is secret independent. For each leakage function ϕ, one thus needs to check that its applicability is secret independent, *i.e.*, that $\bigvee\{\pi_A \mid A \in S, \phi_A = \phi\}$ is secret independent, and whenever ϕ is applicable its value must be secret independent, too.

To check this non-interference property, one can adapt existing constant-time analysis tools [13, 19, 29], which are already able to account for various constant-time policies. A standard assumption, exploited by existing tools, is that the program path leaks and may thus not be secret dependent. This is the case for all contracts synthesized in this work. However, in principle, our framework allows for contracts that do not leak the outcome of branches, which has also been discussed in the context of efficient constant-time programming recently [54, 55] Adapting existing tools to handle such contracts in a precise manner may be more challenging.

8 Related work

Leakage verification: LeaVe [50] *verifies* a given leakage contract against a RTL processor by learning a set of inductive relational invariants. Contract Shadow Logic [47] follows a similar approach, but it uses an explicit state model checker instead of relying on invariants. While both approaches ensure soundness of contracts, these contracts are user-provided and their precision crucially relies on careful modelling by the user. In contrast, our work *synthesizes* sound and precise leakage contracts automatically, starting from a high-level contract template describing possible leakage sources.

Given a hardware design, ConjunCT [21] and its follow-up VeloCT [22] identify a set of *safe* instructions that can be invoked with secret arguments without timing leaks. While the focus on identifying safe instructions allows these tools to scale remarkably well, it also limits the guarantees they provide. In particular, they only guarantee that programs consisting exclusively of safe instructions will not result in leaks. However, since branching and memory instructions affect timing, they are not safe, which severely limits set of programs that can be run securely on the processor.

Ceesay-Seitz et al. [18] introduce a property called microarchitectural control-flow integrity (μ-CFI), which checks for undue influences from instruction operands to the (micro-architectural) program counter using a taint-tracking approach. While μ-CFI captures a number of security violations of processors (*e.g.*, certain speculative execution attacks), it can only check for violations of a fixed property and cannot express general leakage contracts.

Bloem et al. [16] verify leakage contracts for power side channels. Their technique can verify the soundness of a given contract, but it does not perform synthesis and offers no precision guarantees.

Leakage testing: Revizor [40] and Scam-V [39] check black-box CPUs against leakage contracts via testing. White box fuzzers like Phantom Trails [20], IntroSpectre [28], and SpecDoctor [34] can detect speculative execution bugs in CPUs by fuzzing an RTL design, but they cannot capture general contracts. While these approaches can find violations for certain classes of leaks, they cannot prove their absence and do not synthesize leakage descriptions.

Synthesis of leakage descriptors: RTL2muPath [32, 33] synthesizes *microarchitectural execution paths*, which characterize how instructions may traverse a pipelined processor. From these paths and so-called *leakage signatures*, which characterize how path variability may be induced, it is possible to derive leakage contracts in the sense of this paper.

Mohr et al. [38] synthesize leakage contracts that are precise, but cannot guarantee their soundness differently from LeaSyn. Moreover, they cannot handle contract templates where a leakage function is shared by multiple atoms, which LeaSyn uses when synthesizing contracts for all our benchmarks in §6.

9 Conclusion

To use leakage contracts to write software secure against microarchitectural attacks, programmers require sound and precise contracts for existing processors, which we currently lack. To address this gap, we presented an approach to automatically derive such contracts directly from an RTL processor design, with limited user intervention. We implemented our approach in the LeaSyn synthesis tool, which we used to synthesize sound and precise contracts for six open-source RISC-V cores. Our results indicate that contracts synthesized by LeaSyn are more precise than sound contracts constructed in prior works.

Acknowledgments

We would like to thank Sushant Dinesh for the help in applying VeloCT [22] to Ibex. This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreements No. 101020415, and 101115046), from the Spanish Ministry of Science and Innovation under the project TED2021-132464B-I00 PRODIGY, from the Spanish Ministry of Science and Innovation under the Ramón y Cajal grant RYC2021-032614-I, from the Spanish Ministry of Science and Innovation under the project PID2022-142290OB-I00 ESPADA, and from the Spanish Ministry of Science and Innovation under the project CEX2024-001471-M.

References

[1] [n. d.]. DarkRISCV processor. https://github.com/darklife/darkriscv
[2] [n. d.]. Google OR-Tools. https://github.com/google/or-tools
[3] [n. d.]. Ibex: An embedded 32 bit RISC-V CPU core. https://github.com/lowRISC/ibex
[4] [n. d.]. Ibex RISC-V Core – Mult./Divider Block. https://ibex-core.readthedocs.io/en/latest/03_reference/instruction_decode_execute.html#mult-div
[5] [n. d.]. Icarus Verilog. https://github.com/steveicarus/iverilog
[6] [n. d.]. RISC-V Formal Verification Framework. https://github.com/SymbioticEDA/riscv-formal
[7] [n. d.]. RISC-V Sodor processor. https://github.com/ucb-bar/riscv-sodor
[8] [n. d.]. LeaSyn implementation. https://github.com/zilongwang123/LeaSyn
[9] [n. d.]. Yices 2 SMT Solver. https://yices.csl.sri.com
[10] [n. d.]. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys
[11] Onur Aciiçmez. 2007. Yet another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW)*. ACM.
[12] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE.
[13] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. USENIX Association.
[14] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David Wu, and Yuval Yarom. 2024. Testing Side-channel Security of Cryptographic Implementations against Future Microarchitectures. In *Proceedings of the 31st ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
[15] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Math. Struct. Comput. Sci.* 21, 6 (2011), 1207–1252.
[16] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. 2022. Power Contracts: Provably Complete Power Leakage Models for Processors. In *Proceedings of the 29th ACM Conference on Computer and Communication Security (CCS)*. ACM.
[17] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant

CPUs. In *Proceedings of the 26th ACM Conference on Computer and Communication Security (CCS)*. ACM.

[18] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μ-CFI: Formal Verification of Microarchitectural Control-flow Integrity. In *Proceedings of the 31st ACM Conference on Computer and Communication Security (CCS)*. ACM.

[19] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[20] Alvise de Faveri Tron, Raphael Isemann, Hany Ragab, Cristiano Giuffrida, Klaus von Gleissenthall, and Herbert Bos. 2025. Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*. USENIX Association.

[21] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W. Fletcher. 2024. ConjunCT: Learning Inductive Invariants to Prove Unbounded Instruction Safety Against Microarchitectural Timing Attacks. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[22] Sushant Dinesh, Yongye Zhu, and Christopher W. Fletcher. 2025. H-Houdini: Scalable Invariant Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[23] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Trans. Inf. Syst. Secur.* 18, 1 (2015).

[24] Xaver Fabian, Marco Patrignani, and Marco Guarnieri. 2022. Automatic Detection of Speculative Execution Combinations. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[25] Xaver Fabian, Marco Patrignani, Marco Guarnieri, and Michael Backes. 2025. Do You Even Lift? Strengthening Compiler Security Guarantees against Spectre Attacks. *Proc. ACM Program. Lang.* 9, POPL (Jan. 2025).

[26] Bo Fu, Leo Tenenbaum, David Adler, Assaf Klein, Arpit Gogia, Alaa R. Alameldeen, Marco Guarnieri, Mark Silberstein, Oleksii Oleksenko, and Gururaj Saileshwar. 2025. AMuLeT: Automated Design-Time Testing of Secure Speculation Countermeasures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[27] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. 2023. A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[28] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. 2021. IntroSpectre: a pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[29] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[30] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[31] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. 2023. Speculation at fault: modeling and testing microarchitectural leakage of CPU exceptions. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. USENIX Association.

[32] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2021. Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM.

[33] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P. Mulligan, Gustavo Petri, Christopher W. Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[34] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[35] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[37] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[38] Gideon Mohr, Marco Guarnieri, and Jan Reineke. 2024. Synthesizing Hardware-Software Leakage Contracts for RISC-V Open-Source Processors. In *Proceedings of the 27th Design, Automation and Test in Europe Conference and Exhibition (DATE)*. ACM/IEEE.

[39] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. 2020. Validation of Abstract Side-Channel Models for Computer Architectures. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV)*. Springer.

[40] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[41] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[42] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*. ACM.

[43] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. 2023. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In *Proceedings of the 18th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. ACM.

[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 26th ACM Conference on Computer and Communication Security (CCS)*. ACM.

[46] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *CoRR* abs/1806.07480 (2018).

[47] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. 2025. RTL Verification for Secure Speculation Using Contract Shadow Logic. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[48] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES Implemented on Computers with Cache. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer.

[49] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[50] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*. ACM.

[51] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2025. Synthesis of Sound and Precise Leakage Contracts for Open-Source RISC-V Processors. *CoRR* abs/2509.06509 (2025). http://arxiv.org/abs/2509.06509

[52] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley.

[53] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. USENIX Association.

[54] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*. ACM.

[55] Hans Winderix, Marton Bognar, Job Noorman, Lesly-Ann Daniel, and Frank Piessens. 2024. Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[56] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. USENIX Association.

[57] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.* 7, 2 (2017).