Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts

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 recently been proposed as a new security abstraction at the Instruction Set Architecture (ISA) level. Leakage contracts aim to capture the information that processors leak through their microarchitectural implementations. However, so far, we lack a methodology to verify that a processor actually satisfies a given leakage contract.

In this paper, we address this challenge by developing LeaVe, the first tool for verifying register-transfer-level (RTL) processor designs against ISA-level leakage contracts. To this end, we show how to decouple security and functional correctness concerns. LeaVe leverages this decoupling to make verification of contract satisfaction practical. To scale to realistic processor designs, LeaVe further employs inductive reasoning on relational abstractions. Using LeaVe, we precisely characterize the side-channel security guarantees of three open-source RISC-V processors, thereby obtaining the first proofs of contract satisfaction for RTL processor designs.

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. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623192

*Also with Universidad Politécnica de Madrid.

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623192

1 INTRODUCTION

Microarchitectural attacks [15, 32, 34, 46, 51] compromise security by exploiting software-visible artifacts of microarchitectural optimizations like caches and speculative execution. To use modern hardware securely, programmers must be aware of how these optimizations impact the security of their code. Unfortunately, instruction set architectures (ISAs), the traditional abstraction layer between hardware and software, do not provide an adequate basis for secure programming: ISAs capture the functional behavior of processors but abstract away microarchitectural details and thus fail to capture their security implications.

To build secure software systems on top of modern hardware, we need a new abstraction at the ISA level that faithfully captures the information processors may leak through their microarchitectural implementations. We refer to this new abstraction as *leakage contracts*. For example, the leakage contract underlying constant-time programming [9], used for writing cryptographic code, states that processors can leak a program's control flow and memory accesses, which therefore must not depend on secret data.

Recent work has made significant strides towards using leakage contracts as a basis for building secure systems, through their formal specification [27, 37]; through automatic security analysis of software [18, 21, 25, 26, 47]; and through post-silicon processor fuzzing [14, 38–40]. However, leakage contracts can only unfold their full potential once hardware is available that *provably satisfies such contracts*. The proliferation of open-source processors around the RISC-V ecosystem presents an opportunity to fill this gap.

In this paper, we present the first approach for verifying register-transfer-level (RTL) processor designs against ISA-level leakage contracts. This requires overcoming the following challenges:

- Bridging the abstraction gap between sequential instruction-level leakage contracts and cycle-level processor designs that overlap the execution of multiple instructions.
- Leakage contracts capture a processor's information leakage on top of its functional specification. Verifying contract satisfaction, thus, requires reasoning about both functional and security aspects, which goes against the separation of these two concerns.
- Even simple open-source processor designs have large and complex state spaces, which prohibit explicit enumeration or bounded model checking.

Our verification approach and its implementation LeaVe overcome these challenges based on the following contributions:

(1) We adapt the leakage contract framework from [27] to RTL processor designs, capturing instruction-level contracts and realistic cycle-level attacker models in a single uniform framework.

(2) We introduce a decoupling theorem that separates security and functional correctness aspects for contract satisfaction.

(3) We develop a verification algorithm for checking the security aspects of contract satisfaction that employs inductive reasoning on relational abstractions to scale to realistic processor designs.

(4) We implement and experimentally evaluate our approach on three open-source RISC-V processors.

Next, we discuss these four contributions in more detail.

Leakage contracts for RTL processors: We adapt the leakage contract framework from [27] for RTL processor designs (§3). This requires significant changes since the framework in [27] builds on top of a simple sequential operational model of an out-of-order processor rather than on cycle-level RTL circuits. In a nutshell, we model both the instruction-level leakage contract and the microarchitectural attacker as *monitoring circuits*. These monitoring circuits generate contract traces, capturing the processor's *intended leakage* at instruction level, and attacker traces, capturing its *actual leakage* at microarchitectural level. In this setting, a microarchitecture *satisfies a contract* for a given attacker if the following holds: whenever two architectural states yield different attacker traces, then the two states also yield different contract traces.

Decoupling security and functional correctness: We introduce a decoupling theorem (§4.1) that separates security and functional correctness concerns for contract satisfaction. For this, we introduce the notion of *microarchitectural contract satisfaction* that refers only to the microarchitecture and ensures the absence of leaks. The decoupling theorem states that, for processors correctly implementing the instruction set architecture, contract satisfaction and microarchitectural contract satisfaction are equivalent. This allows us to focus *only* on the security challenges arising from leakage verification, while relying on existing approaches for functional correctness [16, 28, 30, 33, 41, 44, 52].

Verifying contract satisfaction: We develop a novel algorithm for checking microarchitectural contract satisfaction (§4.2), which we prove sound. That is, whenever our algorithm concludes that a contract is satisfied, then microarchitectural contract satisfaction indeed holds. Given a contract monitoring circuit and a microarchitecture, our approach inductively learns invariants associated with pairs of microarchitectural executions with the same contract traces using invariant learning techniques [23] and uses these invariants to establish contract satisfaction.

Implementation and evaluation: We implement our approach in LeaVe, a tool for verifying microarchitectural contract satisfaction for processor designs in Verilog (§5). We validate our approach by precisely characterizing the side-channel security guarantees of three open-source RISC-V processors in multiple configurations (§6). For this, we define a family of leakage contracts capturing leaks through control flow, memory accesses, and variable-time instructions, and use LeaVe to determine which contracts each processor satisfies against an attacker observing when instructions retire. Our evaluation confirms that LeaVe can be used to effectively verify side-channel security guarantees provided by open-source processors in less than 25 hours for our most complex targets. Our experiments also show that checking microarchitectural contract satisfaction (as enabled by our decoupling theorem) rather than on top of an architectural reference model significantly speeds up verification (less than 2 hours versus 33 hours for a simple 2-stage processor), allowing us to scale verification to realistic processors.

```
1   module ISA(input clk, output register);
2
3   reg [31:0]  imem [31:0], pc, register;
4   wire [31:0] instr = imem[pc];
5
6   assign op  = instr[7:0];
7   assign imm = instr[31:8];
8
9   always @(posedge clk) begin
10      pc <= pc + 1;
11  end
12
13  always @ (posedge clk) begin
14      case(op)
15      `ADD : register <= register + imm;
16      `MUL : register <= register * imm;
17      `CLR : register <= 0;
18  end
```

Figure 1: ISA reference model for our running example.

Bonus material: The LeaVe verification tool is available at [6]. An extended version of this paper containing the full formal model and proofs of technical results is available at [49].

2 OVERVIEW

Here, we illustrate the key points of our approach with an example. We start by presenting a simple instruction set and the processor implementing it (§2.1). Next, we show how microarchitectural leaks can be formalized using leakage contracts (§2.2). Finally, we illustrate how the LeaVe verification tool verifies that the contract is satisfied, thereby ensuring the absence of unwanted leaks (§2.3).

2.1 A simple processor

Next, we present the instruction set and processor implementation.

Instruction set. We consider an instruction set supporting addition and multiplication of immediates to a single register. Instructions consist of the instruction type (`ADD`, `MUL`, or `CLR`) and an immediate value `imm`. `ADD` adds the immediate to the register value, whereas `MUL` multiplies the register value by the immediate. Finally, `CLR` resets the register to zero.

Figure 1 depicts a Verilog reference model `ISA` for our instruction set that executes one instruction per cycle. Instructions are stored in the instruction memory `imem`. Lines 6 and 7 decode the instruction into operator (`ADD`, `MUL`, or `CLR`) and operand (immediate value). Lines 13 to 18 case-split on the type of operation and update the register with the new value. Finally, Line 10 advances the program counter.

Pipelined implementation. Figure 2 shows an implementation `IMPL` of the instruction set that processes instructions in a three-stage pipeline. If the pipeline is not stalled (flag `ready`), the processor starts by fetching a new instruction in line 13. As in Figure 1, the decode stage (lines 7 to 10) decodes a new instruction into operator and immediate. Next, the execute stage executes the decoded instruction (lines 24 to 34). The write-back stage updates the register

```
1   module IMPL(input clk, output ready, register);
2   reg [31:0] imem [31:0], pcF, register;
3
4   // Decode
5   wire [31:0] instr = imem[pcF];
6
7   always @(posedge clk) begin
8       ex_op  <= inst[7:0];
9       ex_imm <= inst[31:8];
10  end
11
12  always @(posedge clk) begin
13    if (ready) pcF <= pcF + 1;
14  end
15
16  assign pc = pcF-2;   // Architectural pc
17
18  // Execute
19  assign ready = (!mult);
20  assign rd = we ? wb_res : register;// Forwarding
21
22  log_time_mult(mult, m_imm, m_rd, m_res, done);
23
24  always @ (posedge clk) begin
25    if (ready)
26      case(ex_op)
27        `ADD : wb_res <= rd + ex_imm;
28               we <= 1; mult <= 0;
29        `MUL : mult <= 1; we <= 0;
30               m_rd <= rd; m_imm <= ex_imm;
31        `CLR : we <= 1; mult <= 0; wb_res <= 0;
32    if (done)
33      mult <= 0; wb_res <= m_res; we <= 1;
34  end
35
36  // Write back
37  always @ (posedge clk) begin
38    if (we) // write enabled
39      register <= wb_res; retired <= 1;
40    else
41      retired <= 0;
42  end
```

Figure 2: A simple processor that performs addition and multiplications. The multiplication module `log_time_mul` leaks part of both register value and immediate operand via timing.

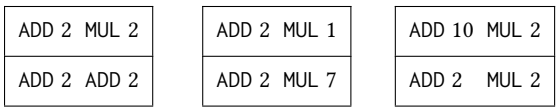

Figure 3: Traces that leak via timing.

with the result of the computation (lines 37 to 42). This step is controlled by the write-enabled flag `we`. Finally, the processor performs forwarding from the execute to the write-back stage (line 20).

Both `ADD` (line 27) and `CLR` (line 31) instructions are executed in a single cycle and their results are passed to the write-back stage.

In contrast, `MUL` instructions (line 29 to line 34) may take multiple cycles. Multiplication starts in line 29 by setting register `mult` to 1. This indicates that the processor cannot fetch new instructions (line 19) and must stall the pipeline (line 13). The processor then multiplies immediate and register value. This step

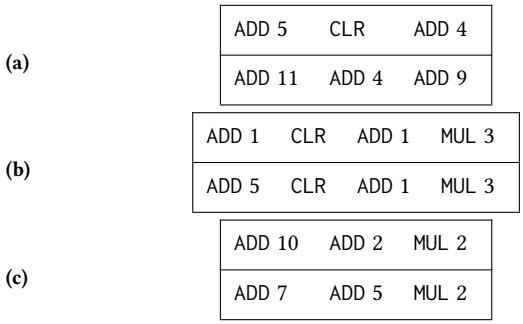

Figure 4: Traces that do not leak via timing.

is implemented in module `log_time_mult` (line 22), which we omit. The module takes time proportional to the logarithm of `m_rd`'s value, *i.e.*, the register value, to perform the multiplication.[1] It also has a fast path that completes the multiplication in a single cycle whenever operand or register are `0` or `1`. Once multiplication terminates, `mul_res` contains the multiplication result and the processor stops stalling the pipeline by setting `mult` to 0 (line 33) and passes the result to the write-back stage.

2.2 Specifying side-channel leakage

We now illustrate how to use leakage contracts to capture side-channel security guarantees for our example processor.

Leakage. To use the processor from Figure 2 securely, we need to know what the processor may leak to an attacker. In the following, we consider an attacker that observes the value of the output-ready flag `ready` at each cycle, *i.e.*, it observes the pipeline's timing.

Assume that initially the register has value `0`. Figure 3 shows pairs of instruction sequences that an attacker can distinguish. The sequences in Figure 3a are distinguishable since the upper trace performs a multiplication while the lower trace does not, resulting in a timing difference. Similarly, the attacker can distinguish the traces in Figure 3b, as the upper trace profits from the fast path in the multiplier, while the lower trace does not. Even though the immediate operands to `MUL` are the same in Figure 3c, the attacker can tell the sequences apart, as the register values are different.

In contrast, Figure 4 shows pairs of instruction sequences that are indistinguishable for our attacker. Figure 4a does not leak as it does not perform multiplication. Figure 4b initially performs additions with different values, but resets the register state via `CLR` before `MUL`. Finally, Figure 4c performs additions with different values that result in the same register state before `MUL`.

Next, we show how to capture leakage using monitors, which we use to formalize leakage contracts and attackers.

Capturing leakage via monitors. To use the processor securely, we need to distinguish program behaviors that leak from those that do not. For this, we compose the reference model `ISA` (Figure 1), which captures the *functional* behavior of the ISA, with a *leakage monitor* `LM` shown below. The leakage monitor captures which information may be leaked upon executing instructions. The monitor takes as input a module M representing the underlying circuit. We denote by `LM`$[M]$ the composition of `LM` and M such that

[1] This timing profile is similar to the Slow Multi-Cycle Multiplier from [3].

the composition hides M's outputs, and LM can refer to (but cannot modify) M's internal variables (§3.3).

In our example, the monitor leaks whether the operation that is performed is a multiplication or not (ismul). Whenever a MUL is executed, the monitor additionally leaks the register value (r) and whether the immediate is 0 or 1 (isFP), thereby capturing the leaks associated with the multiplier's fast path.

```
monitor LM(module M, output leak)
  assign inst = M.imem[M.pc];
  assign r = M.register;
  assign op = inst[7:0];
  assign imm = inst[31:8];
  assign isFP = (imm==0 || imm==1);
  assign ismul = (op==`MUL);

  always @( * ) begin
    if (ismul)
        leak = {r, isFP, ismul};
    else
        leak = {0, 0, ismul};
  end
```

Note that $\{a, b, c\}$ is Verilog notation for the concatenation of signals a, b, and c. Consider the leakage observations (*i.e.*, the values for leak) produced by LM[ISA], *i.e.*, the leakage monitor applied to the reference model. All pairs of sequences in Figure 4 produce the same observations, whereas all pairs in Figure 3 result in different observation traces. For example, in Figure 4a, LM[ISA] produces observations consisting of {0,0,0} for both instruction sequences. In contrast, for the second instruction of Figure 3a, the upper sequence produces observation {2,0,1} but the lower one produces {0,0,0}.

Attacker observations. Next, we define the observations an attacker can make about implementation IMPL. Since we consider an attacker that can observe the timing of the computation, we define another monitor ATK that simply exposes the ready bit.

```
monitor ATK(module M, output leak)
    always @ ( * ) begin
        leak = M.ready;
    end
```

The composition of attacker and implementation ATK[IMPL] defines the *actual information* an attacker may learn about the implementation.

Leakage contracts. The composition LM[ISA] of leakage monitor and reference model defines a *leakage contract* at the ISA level. The contract characterizes leaks at the granularity of the execution of instructions from the instruction set, and it expresses which parts of the computation may be leaked by the hardware. For programmers, the contract provides a guideline for writing side-channel free code: secrets should never influence leakage observations. In our example, any two program executions that differ only in their secrets (*e.g.*, the initial register value) must produce indistinguishable traces.

Contract satisfaction. The implementation IMPL *satisfies* the contract LM[ISA] under the attacker ATK whenever IMPL leaks no more than specified by the contract under ATK. That is, circuit ATK[IMPL] should leak no more than circuit LM[ISA], denoted LM[ISA] $\sqsupseteq$ ATK[IMPL]. That is, for any pair of initial architectural states for which LM[ISA] produces the same leakage observations, ATK[IMPL] must produce the same attacker observations. A formal definition of this relation is provided in §3.3. For example, for all pairs of instruction sequences shown in Figure 4, ATK[IMPL] must produce the same sequence of ready bits. In contrast, for the pairs of sequences in Figure 3, the sequence of ready bits may differ, but it does not have to. Next, we describe our methodology to check that an implementation satisfies a contract.

2.3 Verifying contract satisfaction

Formally verifying contract satisfaction amounts to proving that LM[ISA] $\sqsupseteq$ ATK[IMPL] holds. This requires reasoning about pairs of infinite traces from LM[ISA] and ATK[IMPL] for *all* possible initial memories (including both data and instructions) and *all* possible initial microarchitectural states. Beyond reasoning about security, this also implicitly requires to show that IMPL correctly implements the ISA. In our example, functional correctness bugs in IMPL would often also result in contract violations as leakage observations are a function of the architectural state. For instance, assume an incorrectly implemented CLR instructions that does not reset the register to 0. Then the traces in Figure 4b would likely be distinguishable via timing.

While functional correctness is thus crucial for security, it needs to be verified independently of security concerns. Indeed, there are many existing approaches [16, 28, 30, 33, 41, 44, 52] for checking ISA compliance. One of the contributions of this paper is to show how leakage and functional verification can be decoupled from each other, enabling a clean separation of functional and security concerns.

ISA compliance. So, what does it mean for the implementation to comply with the ISA? Intuitively, the implementation should go through the same sequence of architectural states as the reference model. However, the reference model processes one instruction in each cycle, while the implementation overlaps the execution of multiple instructions and may or may not retire an instruction in any given cycle. To bridge this gap, a *retirement predicate* captures when the processor retires instructions and thus commits changes to the architectural state. A retirement predicate ϕ over implementation circuit IMPL must satisfy the following constraint: whenever ϕ holds, IMPL's current architectural state corresponds to a valid architectural state of the reference model, and no changes to the architectural state may occur when ϕ does not hold. For our example, the architectural variables are pc, imem, and register and $\phi \triangleq (retired = 1)$ is a valid retirement predicate. In fact, ϕ acts as a witness to the fact that IMPL complies with the ISA defined by the reference model ISA: For any initial architectural state, ISA transitions through the same sequence of architectural states as IMPL does upon instruction retirement, *i.e.*, whenever ϕ holds. We denote this notion of ISA compliance by IMPL $\vdash_\phi$ ISA.

Decoupling leakage and functional correctness. Using the retirement predicate, we are able to decouple leakage from functional verification. To this end, we first define a *filtered semantics* (in §3.1) that only considers states in which ϕ holds. Since IMPL $\vdash_\phi$ ISA implies that IMPL's architectural state matches ISA's whenever ϕ holds, the sequence of architectural states produced by the filtered semantics of IMPL with respect to ϕ is equal to the sequence of states produced by ISA, assuming the processor is implemented correctly.

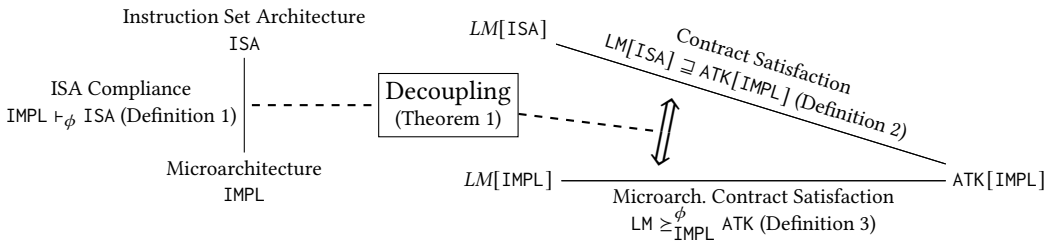

Figure 5: ISA compliance, contract satisfaction, and microarchitectural contract satisfaction.

Based on the filtered semantics, we can define the notion of *microarchitectural contract satisfaction*: To this end, we apply the leakage monitor LM directly to IMPL and then relate LM[IMPL] to ATK[IMPL], bypassing the reference model: For all pairs of traces of LM[IMPL], if contract observations (filtered using ϕ) are the same, then ATK[IMPL]'s observations must also be the same. We denote this relation by $\texttt{LM} \geq^{\phi}_{\texttt{IMPL}} \texttt{ATK}$.

Our main theorem, Theorem 1 (in §4.1), states that if IMPL correctly implements ISA with respect to the retirement predicate ϕ, then $\texttt{LM} \geq^{\phi}_{\texttt{IMPL}} \texttt{ATK}$ *if and only if* LM[ISA] ⊒ ATK[IMPL]. This means that we can analyze contract satisfaction purely based on the implementation IMPL. Figure 5 illustrates the main concepts and their relation.

Verification via inductive invariants. LeaVe, our verification approach, verifies $\texttt{LM} \geq^{\phi}_{\texttt{IMPL}} \texttt{ATK}$ by approximating it via the following safety property: Any two prefixes of traces that agree on their leakage observations also agree on attacker observations *and* determine each instruction's retirement time. A challenge in this formulation is that differences in attacker observations may surface before the corresponding differences in leakage observations due to pipelined execution and the fact that leakage observations corresponding to an instruction can only be evaluated upon instruction retirement. We address this challenge by applying a bounded lookahead to the leakage observations.

Checking this safety property requires appropriate inductive invariants, which would be tedious to come up with manually, in particular for complex designs. Thus, we synthesize appropriate invariants from a pool of candidate relational invariants following the classic Houdini algorithm [23].

3 FORMAL MODEL

In this section, we present the key components of our formal model. We start by introducing μVlog, a simple hardware description language (§3.1). Next, we show how to formalize instruction set architectures and microarchitectures in μVlog (§3.2). We conclude by formalizing leakage contracts (§3.3).

3.1 μVlog: A Hardware Description Language

μVlog is a language for specifying synchronous sequential circuits. It captures the key features of hardware description languages like Verilog and VHDL, and we use it as the core language for LeaVe.

Syntax. The syntax of μVlog is given in Figure 6. *Expressions* e are built from values $Vals = \mathbb{N} \cup \{\bot\}$, which are natural numbers or the designated value $\bot$, registers *Regs*, which store values, and variables *Vars*, which are shorthands for more complex expressions. Expressions can be combined using unary operators $\ominus e$, binary operators $e_1 \otimes e_2$, if-then-else operators **if** e_1 **th** e_2 **el** e_3, and bit-selection operators $e_1[e_2 : e_3]$. An *assignment* $r \leftarrow e$ sets the next value of register x to the value of expression e in the current cycle. A *wire* $v = e$ always has the value of expression e. Finally, a *circuit* C consists of a set of assignments A, a set of wires W, and a set of outputs $O \subseteq Regs \cup Vars$.

Given a circuit C, we refer to its assignments as $C.A$, to its wires as $C.W$, and to its outputs as $C.O$. The set $read(C)$ of read registers consists of all registers x that occur in at least one right-hand side of an assignment in $C.A$ or a wire in $C.W$. Similarly, the set $write(C)$ of write registers consists of all registers x occurring in left-hand sides of assignments in $C.A$. Finally, the set $wires(C)$ of wire variables consists of all variables v occurring in left-hand sides of wires in $C.W$. We assume that (1) $C.O \subseteq vars(C) \cup wires(C)$, where $vars(C) = read(C) \cup write(C)$, (2) each register and variable is on the left-hand side of at most one assignment or wire, and (3) wires in $C.W$ do not introduce cyclic dependencies.

Example 1. Consider the circuit *sISA* given below. The circuit implements a simple ISA, in which instructions consist solely of immediate values $m[pc]$ that are retrieved from memory m and added to the single internal register *reg*.[2]

$$sISA = \{pc \leftarrow pc + 1, reg \leftarrow reg + m[pc]\} : \{\} : \{reg\}$$

We have $vars(sISA) = read(sISA) = \{pc, reg, m\}$ and $write(sISA) = \{pc, reg\}$, and the single output *reg*; the circuit satisfies our assumptions.

Semantics. We formalize the semantics of μVlog circuits by specifying how their state is updated at each cycle. We model the state of a circuit as a *valuation* μ that maps registers in *Regs* to values in *Vals*, *i.e.*, $\mu : Regs \to Vals$. Given a circuit C, $states(C)$ denotes the set of all possible valuations over $vars(C)$. Given a valuation μ and a set of registers V, the projection $\mu\restriction_V$ restricts the scope of μ to the registers in V, *i.e.*, $\mu\restriction_V(x) = \mu(x)$ for all $x \in V$ and $\mu\restriction_V(x) = \bot$ otherwise. Finally, given two valuations μ, μ' and a set of registers V, $\mu \sim_V \mu'$ denotes that μ and μ' agree on the values of all registers in V, *i.e.*, $\mu \sim_V \mu'$ iff $\mu\restriction_V = \mu'\restriction_V$.

The *semantics* $[\![C]\!]$ of a circuit C takes as input a valuation μ and outputs the valuation μ' at the next cycle. An *execution* for C starting from valuation μ is the infinite sequence of valuations obtained by repeatedly applying $[\![C]\!]$. The *infinite trace semantics*

[2]For simplicity, in the examples we treat memories as addressable arrays. For instance, $m[pc]$ denotes the value in m at position pc. While this can be desugared in the syntax from Figure 6, we decided against this to simplify our encodings.

Basic Types			
(Registers)	r	$\in$	$Regs$
(Variables)	v	$\in$	$Vars$
(Identifiers)	i	$\in$	$Regs \cup Vars$
(Values)	n	$\in$	$Vals = \mathbb{N} \cup \{\bot\}$
Syntax			
(Expressions)	e	$:=$	$n \mid i \mid \ominus e \mid e_1 \otimes e_2$
			$\mid$ **if** e_1 **th** e_2 **el** $e_3 \mid e_1[e_2 : e_3]$
(Wires)	w	$:=$	$v = e$
	W	$:=$	$\{w_1, \ldots, w_k\}$
(Assignments)	a	$:=$	$r \leftarrow e$
	A	$:=$	$\{a_1, \ldots, a_n\}$
(Outputs)	O	$:=$	$\{i_1, \ldots, i_m\}$
(Circuits)	C	$:=$	$A : W : O$

Figure 6: μVLOG syntax

$[\![C]\!]^\infty$ of a circuit C maps each valuation μ to the infinite sequence of valuations for C's outputs, where the i-th valuation corresponds to the circuit's output after i cycles.[3] Additionally, the *filtered infinite trace semantics* $[\![C]\!]^\infty|\phi$ outputs only the valuations in $[\![C]\!]|\phi$ that satisfy a given predicate ϕ (other valuations are dropped). Finally, $[\![C]\!](\mu, i)$ denotes the valuation obtained by executing C for i cycles starting from valuation μ, whereas $C, \mu \models \phi$ denotes that ϕ is satisfied for circuit C and valuation μ. The full formalization of μVLOG is given in [49].

Example 2. Consider again circuit *sISA* from Example 1. Let us pick an initial valuation μ, such that $\mu(pc) = 0$, $\mu(reg) = 0$, and

$$\begin{array}{ll} \mu(m)(i) = i & \text{for } 0 \le i \le 10 \\ \mu(m)(i) = 0 & \text{otherwise .} \end{array}$$

Executing a single step gives us $\mu' = [\![sISA]\!](\mu)$, with $\mu'(pc) = 1$, and $\mu'(reg) = 0$. Since only the program counter changed, we get $\mu \sim_{\{reg,mem\}} \mu'$, but not $\mu \sim_{\{pc\}} \mu'$. The trace $[\![sISA]\!]^\infty(\mu)$ consists of the following sequence of register values (since the register value does not change after step 11), where $\cdot$ denotes concatenation:

$$[\![sISA]\!]^\infty(\mu) = 0 \cdot 0 \cdot 1 \cdot 3 \cdot 6 \cdot 10 \cdot 15 \cdot 21 \cdot 28 \cdot 36 \cdot 45 \cdot 55 \cdot 55 \cdot 55 \ldots$$

As an example of filtering, consider the predicate $\phi := pc \bmod 2 = 0$ indicating whether the program counter is even. The filtered semantics associated with ϕ yields the following sequence:

$$[\![sISA]\!]^\infty|\phi(\mu) = 0 \cdot 1 \cdot 6 \cdot 15 \cdot 28 \cdot 45 \cdot 55 \cdot \ldots$$

3.2 Modeling architectures and microarchitectures

We now show how instruction set architectures (short: architectures) and microarchitectures can be modeled in μVLOG. Then, we formalize what it means for a microarchitecture *IMPL* to correctly implement an architecture *ISA*.

Architectures. We view architectures as state machines that define how the execution progresses through a sequence of architectural states, where each transition corresponds to the execution of a single instruction. Given a set of *architectural registers ARCH*, we model an *architecture* as a circuit *ISA* over *ARCH*, *i.e.*, $vars(ISA) = ISA.O = ARCH$. We assume that a subset $init(ISA)$ of *ISA*'s states are identified as initial states.

Example 3. Consider again circuit *sISA* from Example 1. Its variables $vars(sISA) = \{pc, reg, m\}$ form the architectural state of the ISA. We identify as initial states all valuations μ such that $\mu(pc) = 0$ and $\mu(reg) = 0$. In the circuit from Figure 1 the architectural state is given by $vars(R) = \{imem, pc, register\}$ whereas $instr$, op, and imm are not listed as they are wires.

Microarchitectures. We model microarchitectures as circuits that capture the execution at the granularity of clock cycles. Thus, a *microarchitecture* is a circuit *IMPL* that refers to both architectural registers in *ARCH* and to additional microarchitectural registers $\mu ARCH$ such that $vars(IMPL) = IMPL.O = ARCH \cup \mu ARCH$ and $ARCH \cap \mu ARCH = \emptyset$. We assume that a subset $init(IMPL)$ of *IMPL*'s states is identified as the initial states and require that $\mu\restriction_{ARCH} \in init(ISA)$ for any state $\mu \in init(IMPL)$, *i.e.*, the architectural part of an initial microarchitectural state should be an initial architectural state.

Example 4. Let us look at a microarchitectural implementation *sIMPL* of the ISA in example 1. The implementation, shown below, can be in one of two states (indicated by the register st): execute state ($st = 0$) or write-back state ($st = 1$). In the execute state, *sIMPL* computes the result of adding the immediate to the current register value and assigns it to the variable res; it then moves to the write-back state (line 3). In the write-back state, *sIMPL* writes the result to the register reg, moves the state to the execute stage, and increments the program counter (line 4). If the immediate value is zero, the implementation triggers a fast path which keeps the circuit in the execute state, increments the program counter, and leaves the register unchanged (line 2). Finally, the circuit updates the variable ret which indicates whether the circuit retired in the current step. For readability, we write the example in an extended syntax that allows branches at the assignment level.[4]

```
1  if st = 0 then
2     if m[pc] = 0 th {st ← 0, pc ← pc + 1, ret ← 1}
3     el {st ← 1, res ← m[pc] + reg, ret ← 0}
4  el {st ← 0, reg ← res, pc ← pc + 1, ret ← 1} : {reg}
```

In addition to architectural variables $\{pc, reg, m\}$, the implementation contains microarchitectural variables $\{st, res, ret\}$. We pick as our initial valuations all μ such that $\mu(pc) = 0, \mu(reg) = 0, \mu(st) = 0$, and $\mu(ret) = 1$. As required, the initial state for architectural variables $\{pc, reg, m\}$ agrees with the state from Example 3.

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 with the help of a *retirement predicate* ϕ, a predicate indicating when *IMPL* retires instructions. Then, we say that a microarchitecture *IMPL* implements an architecture *ISA* (Definition 1) if one can map changes of the architectural state in *IMPL* to *ISA*'s executions using ϕ.

Definition 1. A microarchitecture *IMPL correctly implements* an architecture *ISA* given a retirement predicate ϕ over $vars(IMPL)$, written $IMPL \vdash_\phi ISA$, if for all valuations $\mu \in init(IMPL)$:

[3] With a slight abuse of notation, the trace semantics extend valuations to also record values of wires that are part of a circuit's outputs.

[4] This syntax can be easily expanded into the one in Figure 6 by pushing branches into expressions. For example, we can rewrite **if** e **th** { $x \leftarrow a$ } **el** { $x \leftarrow b$ } as $x \leftarrow$ **if** e **th** a **el** b .

(1) *(Witnessed architectural changes agree with ISA)* $[\![\textsc{Impl}]\!]^\infty|\phi(\mu) \sim_{\textit{ARCH}} [\![\textit{ISA}]\!]^\infty(\mu)$, and
(2) *(No architectural changes beyond those witnessed)* $[\![\textsc{Impl}]\!](\mu, i) \sim_{\textit{ARCH}} [\![\textsc{Impl}]\!](\mu, i-1)$ whenever $[\![\textsc{Impl}]\!](\mu, i) \not\models \phi$.

The predicate ϕ characterizes when instructions are retired, *i.e.*, when instructions modify the architectural state. Definition 1 uses ϕ to map architectural changes made by Impl to single steps in *ISA*'s executions. This is sufficient for single-issue processors, which retire at most one instruction per cycle. Multiple-issue processors, which may retire multiple instructions in a single cycle, require more complex ways of mapping architectural changes made by Impl to *ISA*'s steps. To simplify our model, we decided against more complex ISA compliance criteria since LeaVe's verification approach (§4) is decoupled from ISA compliance.

Example 5. Let's again consider implementation circuit *sImpl* from Example 4. We choose as retirement predicate $\phi \triangleq ret = 1$. Let's consider again valuation μ from Example 2, which maps $pc = 0$, and $\mu(m)(i) = i$, for $0 \le i \le 10$. Running *sImpl* on μ from produces the following sequence of register values, where we underline a register value whenever ϕ holds on the corresponding state.

$$[\![\textit{sImpl}]\!]^\infty(\mu) = \underline{0} \cdot \underline{0} \cdot 0 \cdot \underline{1} \cdot 1 \cdot \underline{3} \cdot 3 \cdot \underline{6} \cdot 6 \cdot \underline{10} \cdot 10 \cdot \underline{15} \cdot 15 \cdot \ldots$$

It's easy to check that $[\![\textit{sImpl}]\!]^\infty|\phi(\mu)$, *i.e.*, the sequence of underlined values, matches $[\![\textit{sISA}]\!]^\infty(\mu)$, and that the register value remains unchanged whenever ϕ doesn't hold. Since this is true, not only for μ but for all valid initial states, we can conclude that *sImpl* correctly implements *sISA*, *i.e.*, $\textit{sImpl} \vdash_\phi \textit{sISA}$.

3.3 Leakage contracts

In this section, we first introduce monitoring circuits, which we use to specify leakage contracts and attackers. Then, we formalize contract satisfaction [27] within our modeling framework.

Monitoring circuits. Monitoring circuits *monitor* the behavior of another circuit, and we will use them to formalize leakage contracts and attackers. We say that circuit M is a *monitoring circuit* for circuit C if (1) $\textit{write}(C) \cap \textit{write}(M) = \emptyset$, *i.e.*, the two circuits write to separate sets of registers, (2) $\textit{wires}(C) \cap \textit{wires}(M) = \emptyset$, *i.e.*, the two circuits write to separate wire variables, and (3) $\textit{vars}(C) \cap \textit{write}(M) = \emptyset$, *i.e.*, M does not influence C's behavior. Additionally, M is *combinatorial* whenever $\textit{read}(M) \subseteq \textit{vars}(C)$, *i.e.*, M only reads from C variables and thus does not have state of its own. Finally, the *composition* of the monitoring circuit M and the monitored circuit C, written $M[C]$, is the circuit defined as $C.A \cup M.A : C.W \cup M.W : M.O$, which computes over C's state without changing its behavior.

Leakage contracts. A *leakage contract* is the composition of a leakage monitor *LM*, *i.e.*, a combinatorial monitoring circuit *LM* for the architecture *ISA*, with the architecture *ISA* itself. That is, a leakage contract $\textit{LM}[\textit{ISA}]$ discloses parts of the architectural state during *ISA*'s execution at the granularity of instruction execution.

Hardware attackers. We formalize an *attacker* as a combinatorial monitoring circuit Atk for the microarchitecture Impl. That is, an attacker observes parts of the microarchitecture's state during the execution at the granularity of clock cycles.

(a)

1	0	2
5	1	3

(b)

1	0	2
5	0	3

Figure 7: Two pairs of instruction traces.

Example 6. Consider again circuit *sISA* from Example 1, the ISA specification of our running example. We define the leakage monitor *sLM*, which leaks whether the current instruction is zero. As *sLM* only reads *sISA*'s variables, it is combinatorial.

$$\textit{sLM} = \{\} : \{v = (m[pc] = 0)\} : \{v\}$$

Consider again the valuation μ from Example 2, which maps $\mu(m)(i) = i$, for $0 \le i \le 10$. Since for $i \le 10$, only the first instruction is zero, executing $\textit{sLM}[I]$ yields the following sequence.

$$[\![\textit{sLM}[\textit{sISA}]]\!]^\infty(\mu) = 1 \cdot 0 \cdot 0 \cdot 0 \cdot \ldots .$$

Example 7. Next, consider the implementation circuit *sImpl* from Example 4. We define the following attacker monitor, which leaks the program counter and thus the timing of the computation.

$$\textit{sAtk} = \{\} : \{\} : \{pc\}$$

Running $\textit{sAtk}[\textit{sImpl}]$ on μ yields the following sequence.

$$[\![\textit{sAtk}[\textit{sImpl}]]\!]^\infty(\mu) = 0 \cdot 1 \cdot 1 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot \ldots .$$

Contract satisfaction. Definition 2 formalizes the notion of contract satisfaction [27]. Intuitively, a microarchitecture Impl satisfies the contract $\textit{LM}[\textit{ISA}]$ 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 $\textit{LM}[\textit{ISA}]$. That is, for any two initial states that agree on their microarchitectural part[5], whenever $\textit{LM}[\textit{ISA}]$ results in identical traces, then $\textsc{Atk}[\textsc{Impl}]$ also results in identical traces (*i.e.*, Atk cannot distinguish the two initial architectural states).

Definition 2. Microarchitecture Impl *satisfies* contract $\textit{LM}[\textit{ISA}]$ for attacker Atk, written $\textit{LM}[\textit{ISA}] \sqsupseteq \textsc{Atk}[\textsc{Impl}]$, if for all valuations $\mu, \mu' \in \textit{init}(\textsc{Impl})$ such that $\mu \sim_{\mu\textit{ARCH}} \mu'$, if $[\![\textit{LM}[\textit{ISA}]]\!]^\infty(\mu) = [\![\textit{LM}[\textit{ISA}]]\!]^\infty(\mu')$, then $[\![\textsc{Atk}[\textsc{Impl}]]\!]^\infty(\mu) = [\![\textsc{Atk}[\textsc{Impl}]]\!]^\infty(\mu')$.

We remark that Definition 2 refers to 4 different traces: two contract traces from $\textit{LM}[\textit{ISA}]$ and two attacker traces from $\textsc{Atk}[\textsc{Impl}]$.

Example 8. Let's consider the two pairs of memories (a) and (b) shown in Figure 7. We will check contract satisfaction, *i.e.*, that $\textit{sLM}[\textit{sISA}] \sqsupseteq \textit{sAtk}[\textit{sImpl}]$ on these particular traces.

Let us start with the instructions from Figure 7a. Consider two states μ_a and μ'_a, such that $\mu_a(m)$ contains the upper instructions in Figure 7, and $\mu'_a(m)$ contains the lower ones. For $i \ge 3$, we let $\mu_a(m) = \mu'_a(m) = 0$. Running μ_a and μ'_a on the contract, we get:

$$[\![\textit{sLM}[\textit{sISA}]]\!]^\infty(\mu_a) = 0 \cdot \underline{1} \cdot 0 \cdot 1 \cdot 1 \cdot 1 \cdot \ldots$$
$$[\![\textit{sLM}[\textit{sISA}]]\!]^\infty(\mu'_a) = 0 \cdot \underline{0} \cdot 0 \cdot 1 \cdot 1 \cdot 1 \cdot \ldots$$

As the contract traces differ in the second position, contract satisfaction holds trivially. Next, consider the traces in Figure 7b. As

[5]Following [27], we assume that secrets initially reside only in the architectural state and that attackers can observe the initial values of registers in μArch, *i.e.*, $\mu \sim_{\mu\textit{ARCH}} \mu'$.

before, we construct valuations μ_b for the upper trace, and μ'_b for the lower trace. We get the traces below.

$$[\![sLM[sISA]]\!]^\infty(\mu_b) = [\![sLM[sISA]]\!]^\infty(\mu'_b) = 0 \cdot 1 \cdot 0 \cdot 1 \cdot 1 \cdot 1 \cdot \dots .$$

As both valuations produce the same trace, we need to check the attacker observations on the implementation. We get

$$[\![sAtk[sImpl]]\!]^\infty(\mu_b) = [\![sAtk[sImpl]]\!]^\infty(\mu'_b) = 0{\cdot}0{\cdot}1{\cdot}2{\cdot}2{\cdot}3{\cdot}4{\cdot}5{\cdot}\dots .$$

We can therefore conclude that contract satisfaction holds for these traces. To verify contract satisfaction, we need to not only check this property for μ_b and μ'_b, but for any pair of traces. We will discuss our approach for this in the next section.

4 VERIFYING CONTRACT SATISFACTION

Here, we present our verification approach for checking contract satisfaction. First, we introduce a decoupling theorem that allows us to separate security and functional correctness proofs (§4.1). Next, we present (and prove sound) an algorithm for verifying microarchitectural contract satisfaction (§4.2). All proofs are in [49].

4.1 Decoupling contract satisfaction from ISA

Since a leakage contract $LM[ISA]$ is defined on top of *ISA*, proving contract satisfaction according to Definition 2 requires reasoning about security *and* functional compliance with respect to *ISA* (since one needs to map contract traces from $LM[ISA]$ to implementation traces). We address this challenge by decoupling reasoning about security and about *ISA* compliance.

Leakage ordering. For this, we start by introducing a leakage ordering between combinatorial monitoring circuits for an underlying circuit C. Intuitively, a monitor M for C "leaks less" (*i.e.*, exposes less information) than another monitor M' for C if whenever $M'[C]$ produces equivalent traces on two initial states, then $M[C]$ also produces equivalent traces. Definition 3 formalizes this concept and extends it to support the filtered semantics.

Definition 3. Monitor M' *leaks at most as much information as* monitor M about circuit C, given registers $V \subseteq vars(C)$, and predicate ϕ (over C), written $M \geq_C^{V,\phi} M'$, if for all valuations $\mu, \mu' \in init(C)$ such that $\mu \sim_V \mu'$, if $[\![M[C]]\!]^\infty|\phi(\mu) = [\![M[C]]\!]^\infty|\phi(\mu')$, then $[\![M'[C]]\!]^\infty(\mu) = [\![M'[C]]\!]^\infty(\mu')$.

Differently from Definition 2 (which is defined in terms of four traces), Definition 3 is defined in terms of only two traces of C.

Example 9. We can use our new definition to express contract satisfaction over the implementation only, using predicate ϕ. Consider again the two pairs of traces in Figure 7 from Example 8. If we assume that the implementation is functionally correct, that is, it satisfies Definition 1, we can replace the specification *sISA* by its implementation *sImpl*. In particular, since Definition 1 ensures that *sISA*'s architectural values match *sImpl*'s whenever retirement predicate $\phi = (ret = 1)$ holds, we can check contract satisfaction by checking $sLM \geq_{sImpl}^{\{st,res,ret\},\phi} sAtk$. We call this condition *microarchitectural contract satisfaction*. Let us now check this property for the traces in Figure 7b. Running $sLM[sImpl]$, we get the following, where we underline outputs whenever ϕ holds.

$$[\![sLM[sImpl]]\!]^\infty(\mu_b) = [\![sLM[sImpl]]\!]^\infty(\mu'_b) = \underline{0} \cdot 0 \cdot \underline{1} \cdot \underline{0} \cdot 0 \cdot \underline{1} \cdot \underline{1} \cdot \dots .$$

This means the premise of the implication is satisfied, and we need to check the conclusion. As before, we get

$$[\![sAtk[sImpl]]\!]^\infty(\mu_b) = [\![sAtk[sImpl]]\!]^\infty(\mu'_b) = 0{\cdot}0{\cdot}1{\cdot}2{\cdot}2{\cdot}3{\cdot}4{\cdot}5{\cdot}\dots .$$

which establishes $sLM \geq_{sImpl}^{\{st,res,ret\},\phi} sAtk$ for μ_b and μ'_b. We formalize this idea in Theorem 1.

Decoupling theorem. Theorem 1 states that, for functionally correct processors, *microarchitectural contract satisfaction* (*i.e.*, $LM \geq_{Impl}^{\mu Arch,\phi} Atk$, which only refers to the microarchitecture *Impl*), is equivalent to contract satisfaction (Definition 2 which refers to architecture *ISA* and microarchitecture *Impl*). This allows us to cleanly separate reasoning about security and about functional correctness (without losing precision). In particular, we can split proving contract satisfaction into proving microarchitectural contract satisfaction (which ensures the absence of leaks with respect to *Impl*) and ISA compliance. LeaVe leverages Theorem 1 to only reason about security, whereas ISA compliance can be verified separately using techniques focusing on functional correctness [44].

Theorem 1 (Decoupling Theorem). *If* $Impl \vdash_\phi$ *ISA holds for retirement predicate* ϕ*, then*

$$LM \geq_{Impl}^{\mu Arch,\phi} Atk \Leftrightarrow LM[ISA] \sqsupseteq Atk[Impl].$$

4.2 Verifying microarchitectural contract satisfaction

In this section, we present an algorithm for checking microarchitectural contract satisfaction, *i.e.*, $LM \geq_{Impl}^{\mu Arch,\phi} Atk$. We first introduce notation for formalizing our verification queries in terms of temporal logic formulas. Next, we present the verification algorithm and conclude by proving its soundness.

Notation. To formalize our verification queries, we use a linear temporal logic over μVlog circuits. Formulas Φ in this logic are constructed by combining μVlog predicates ϕ with temporal operators $\circ$ (denoting "in the next cycle"), $\Box^B$ (denoting "for the next B cycles"), and $\Box$ (denoting "always in the future"), and the usual boolean operators. Given a temporal formula Φ over a circuit C, we write $C, \mu, i \models \Phi$ to denote that the formula is satisfied for initial state μ at cycle i. We write $C, \mu \models \Phi$ to mean $C, \mu, 0 \models \Phi$, and $C \models \Phi$ to mean that $C, \mu \models \Phi$ holds for all μ. Our temporal logic is standard; we provide its formalization in [49].

Example 10. Consider again circuit *sISA* from Example 1. Using initial valuation μ, where $\mu(pc) = 0$, the following holds.

$$sISA, \mu \models pc = 0 \qquad sISA, \mu \models \circ(pc = 1)$$
$$sISA, \mu \models \Box^3(pc \leq 3) \qquad sISA \models pc \geq 0 \rightarrow \Box(pc \geq 0)$$

Product circuit. Verifying microarchitectural contract satisfaction requires us to reason about *pairs of executions* of *Impl*, *i.e.*, it is a 2-hyperproperty [19]. We transform hyperproperties into properties over a single execution using a construction called *self-composition* [12]. For this, we construct a *product circuit* that executes two copies of a circuit C (C^1 and C^2) in parallel. Given circuit $C = \{x_1 \leftarrow e_1, \dots, x_n \leftarrow e_n\} : \{v_1 = e'_1, \dots, v_k = e'_k\} : o_1, \dots, o_m$, we define its *product circuit* $C \times C$ as $\{x_1{}^1 \leftarrow e_1{}^1, \dots, x_n{}^1 \leftarrow e_n{}^1, x_1{}^2 \leftarrow e_1{}^2, \dots, x_n{}^2 \leftarrow e_n{}^2\} : \{v_1{}^1 = e'_1{}^1, v_1{}^2 = e'_1{}^2, \dots v_k{}^1 =$

$e'_k{}^1, v_k{}^2 = e'_k{}^2, \} : \{o_1{}^1, \ldots, o_m{}^1, o_1{}^2, \ldots, o_m{}^2\}$ where e^i, for $i \in \{1, 2\}$, is obtained by replacing all registers x with x^i and all variables v with v^i in expression e.

Stuttering product circuit. While the product circuit allows us to reason about pairs of executions, we need another ingredient to check microarchitectural contract satisfaction, as it refers to the *filtered semantics* over a predicate ϕ. We cannot directly check the filtered semantics on the product circuit, as ϕ may be satisfied at different times. Instead, we modify the product circuit to synchronize the two executions based on ϕ. Given a circuit $C = \{x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n\} : \{v_1 = e'_1, \ldots, v_k = e'_k\} : o_1, \ldots, o_m$, we define its *stuttering product circuit* over predicate ϕ, denoted by $C \times_\phi C$, by replacing each assignment $x^1 \leftarrow e^1$ in the product circuit $C \times C$ with $x^1 \leftarrow \textbf{if } \phi^1 \wedge \neg\phi^2 \textbf{ th } x^1 \textbf{ el } e^1$ and, similarly, by replacing each $x^2 \leftarrow e^2$ in the product circuit $C \times C$ with $x^2 \leftarrow \textbf{if } \phi^2 \wedge \neg\phi^1 \textbf{ th } x^2 \textbf{ el } e^2$. This transformation ensures that whenever ϕ holds in one execution but not the other, the execution where ϕ holds "waits" for the other one to catch up.

Example 11. Consider the circuit $N = \{i \leftarrow i + 1\} : \{\} : \{i\}$. Forming the product yields $N \times N = \{i^1 \leftarrow i^1 + 1, i^2 \leftarrow i^2 + 1\} : \{\} : \{i^1, i^2\}$. Let us define filter predicate $\phi = (i \textbf{ mod } 2 = 0)$. We get $\phi^1 = (i^1 \textbf{ mod } 2 = 0)$, and $\phi^2 = (i^2 \textbf{ mod } 2 = 0)$, and

$$N \times_\phi N = \begin{Bmatrix} i^1 \leftarrow \textbf{if } \phi^1 \wedge \neg\phi^2 \textbf{ th } i^1 \textbf{ el } i^1 + 1, \\ i^2 \leftarrow \textbf{if } \phi^2 \wedge \neg\phi^1 \textbf{ th } i^2 \textbf{ el } i^2 + 1 \end{Bmatrix} : \{\} : \{i^1, i^2\} .$$

Let us fix $\mu_I(i^1) = 0$ and $\mu_I(i^2) = 1$. We only want to compare states where both ϕ^1 and ϕ^2 hold, *i.e.*, we want to compare the filtered semantics $[\![N]\!]^\infty|\phi(\mu^1)$ and $[\![N]\!]^\infty|\phi(\mu^2)$, where $\mu^1(i) = 0$ and $\mu^2(i) = 1$. In $N \times N$ the two executions are not synchronized and $N \times N, \mu_I \models \square(\phi^1 \leftrightarrow \neg\phi^2)$. In contrast, $N \times_\phi N$ synchronizes the two executions. As initially ϕ^1 holds but ϕ^2 does not, only i^2 gets incremented and, afterwards, the two copies run in lockstep. We can now check properties of the filtered semantics, *e.g.*, that $N \times_\phi N, \mu_I \models \square(\phi^1 \wedge \phi^2 \rightarrow i^2 = i^1 + 2)$ holds.

Algorithm idea. We now use the stuttering product circuit to verify that $LM \geq^{\mu\text{Arch},\phi}_{\text{Impl}}$ Atk holds. This requires us to show that all executions whose filtered semantics produce the same contract observations always produce the same attacker observations (see Definition 3). We start by adding an assumption to only consider executions of $\text{Impl} \times_\phi \text{Impl}$ that are contract equivalent. We encode this via the formula $\Phi_{ctr-equiv} := (\phi^1 \wedge \phi^2 \rightarrow \psi^{LM}_{equiv})$, where $\psi^M_{equiv} := \bigwedge_{o \in M.O} o^1 = o^2$ for a monitor M. We then only consider executions that satisfy $\square\Phi_{ctr-equiv}$. Next, our algorithm learns an inductive invariant LI over the stuttering product circuit under our assumption. This invariant holds on all reachable states of the circuit. Finally, our algorithm uses the invariant to prove that indeed all executions of the circuit are attacker equivalent. For this we show that $LI \rightarrow \psi^{\text{Atk}}_{equiv}$ holds. Note that we prove this property over $\text{Impl} \times_\phi \text{Impl}$, however the consequent of $LI \rightarrow \psi^{\text{Atk}}_{equiv}$ is stated over the unfiltered semantics. To ensure that the stuttering semantics is equivalent to the regular one, we also prove $LI \rightarrow (\phi^1 \leftrightarrow \phi^2)$, *i.e.*, no stuttering occurs on contract equivalent traces.

Algorithm description. We implement this approach in Algorithm 1. It relies on the procedure LearnInv, which we use to learn invariants over the stuttering product circuit. We first present Verify and later discuss LearnInv.

Function Verify is the entry point of our verification approach. It takes as input a μVlog microarchitecture Impl (the processor under verification), a leakage monitor LM (capturing the allowed leaks), an attacker monitor Atk (capturing what the attacker can observe), and a retirement predicate ϕ. To verify unbounded properties like $LM \geq^{\mu\text{Arch},\phi}_{\text{Impl}}$ Atk, the algorithm relies on inductive reasoning. For this reason, Verify additionally take as input (1) a set of candidate invariants CI over the stuttering circuit (which will be verified using LearnInv) as well as (2) a lookahead $b \in \mathbb{N}^+$. Concretely, LeaVe constructs the set of candidate invariants CI directly from Impl, Atk, and ϕ; see §5 for more details.

In line 2, we construct $\Phi_{initial}$ (over the stuttering circuit $\text{Impl} \times_\phi \text{Impl}$) capturing the initial conditions for pairs of executions relevant to our check. In $\Phi_{initial}$, $\psi^{\text{Impl}}_{init}{}^1$ and $\psi^{\text{Impl}}_{init}{}^2$ capture that the two executions start from valid initial states, whereas $\psi^{\mu\text{Arch}}_{equiv}$ ensures that the two executions initially agree on all registers in μArch, *i.e.*, $\psi^{\mu\text{Arch}}_{equiv} := \bigwedge_{x \in \mu\text{Arch}} x^1 = x^2$. In line 3, we construct $\Phi_{ctr-equiv} := (\phi^1 \wedge \phi^2 \rightarrow \psi^{LM}_{equiv})$ ensuring that contract observations are equivalent. In line 4, we call the LearnInv procedure to verify which of the candidate invariants in CI are, indeed, invariants. Hence, the learned invariants LI hold for any two contract-indistinguishable executions, *i.e.*, $C \models (\Phi_{initial} \wedge \square\Phi_{ctr-equiv}) \rightarrow \square \bigwedge LI$ holds where $\bigwedge LI$ stands for $\bigwedge_{\phi \in LI} \phi$. Finally, in line 5 we check whether the learned invariants are sufficient to ensure that (1) the attacker observations are the same and (2) the predicate ϕ is always synchronized between the two executions. If this is the case, Verify has successfully verified that $LM \geq^{\mu\text{Arch},\phi}_{\text{Impl}}$ Atk holds; see Theorem 2.

The LearnInv procedure learns, using inductive verification, which of the candidate invariants are true invariants using an approach similar to the Houdini tool [23]. LearnInv takes as input a circuit C, a formula capturing initial conditions $\Phi_{initial}$, a formula $\Phi_{assumption}$ that executions always need to satisfy, a bound b, and a set of candidate invariants CI. The procedure outputs the formulas in CI that can be proved to be invariants, *i.e.*, for which $C \models (\Phi_{initial} \wedge \square\Phi_{assumption}) \rightarrow \square \bigwedge LI$ holds. Concretely, LearnInv consists of a base case (lines 7–13) and an induction step (lines 14–20). Both parts follow a similar structure—they iteratively rule out invalid invariants based on counterexamples—and they differ only in the checked property: Ψ_{base} checks that for any state for which the initial conditions hold and for which the assumptions are satisfied for the next b cycles, the invariants must also hold. In contrast, $\Psi_{induction}$ checks that for any state for which the invariants hold and for which the assumptions are satisfied for the next b cycles, the invariants hold in the next cycle as well. Bound b controls for how many cycles to unroll the assumption $\square\Phi_{assumption}$. Unrolling the assumption is important for circuits where a difference in attacker observation occurs before a corresponding difference in contract observations. This may happen, *e.g.*, if a leak occurs early in the pipeline and is later justified by a difference in contract observations at retirement. It therefore often suffices to bound b by the processor's pipeline depth.

Soundness: Theorem 2 states that whenever Algorithm 1 returns $\top$, then microarchitectural contract satisfaction holds.

Algorithm 1 LeaVe verification approach

Input: Microarchitecture *Impl*, leakage monitor *LM*, attacker Atk, retirement predicate ϕ, lookahead b, candidate invariants *CI*

1: **procedure** Verify(*Impl*, *LM*, Atk, ϕ, b, *CI*)
2: $\quad\Phi_{initial} := \psi_{init}^{Impl\,1} \wedge \psi_{init}^{Impl\,2} \wedge \psi_{equiv}^{\mu Arch}$
3: $\quad\Phi_{ctr-equiv} := (\phi^1 \wedge \phi^2 \rightarrow \psi_{equiv}^{LM})$
4: $\quad LI :=$ LearnInv($Impl \times_\phi Impl, \Phi_{initial}, \Phi_{ctr-equiv}, b, CI$)
5: $\quad$**return** $Impl \times_\phi Impl \models \bigwedge LI \rightarrow \psi_{equiv}^{Atk} \wedge (\phi^1 \leftrightarrow \phi^2)$
6: **procedure** LearnInv($C, \Phi_{initial}, \Phi_{assumption}, b, CI$)
7: $\quad$**while** $\top$ **do** ▷ base case
8: $\quad\quad\Psi_{base} := (\Phi_{initial} \wedge \Box^b \Phi_{assumption}) \rightarrow \bigwedge CI$
9: $\quad\quad$**if** $C \models \Psi_{base}$ **then**
10: $\quad\quad\quad$**break**
11: $\quad\quad$**else**
12: $\quad\quad\quad$Let μ be the counterexample
13: $\quad\quad\quad CI := \{\phi \in CI \mid C, \mu \models \phi\}$
14: $\quad$**while** $\top$ **do** ▷ ind. step
15: $\quad\quad\Psi_{inductive} := (\bigwedge CI \wedge \Box^b \Phi_{assumption}) \rightarrow \circ \bigwedge CI$
16: $\quad\quad$**if** $C \models \Psi_{inductive}$ **then**
17: $\quad\quad\quad$**return** *CI*
18: $\quad\quad$**else**
19: $\quad\quad\quad$Let μ be the counterexample
20: $\quad\quad\quad CI := \{\phi \in CI \mid C, \mu \models \phi\}$

Theorem 2. $Verify(Impl, LM, Atk, \phi, b, RI) \Rightarrow LM \geq_{Impl}^{\mu Arch, \phi} Atk$.

Example 12. Consider again the implementation *sIMPL* from Example 4. We want to verify that $sLM \geq_{sIMPL}^{\{st,res,ret\},\phi} sAtk$ holds. We start by building the stuttering product circuit $sIMPL \times_\phi sIMPL$ with respect to retirement predicate $\phi = (ret = 1)$. We can assume that the two executions produce the same contract observations, whenever both executions retire. We capture this assumption in formula $\Phi_{ctr-equiv} := (ret^1 = 1 \wedge ret^2 = 1 \rightarrow m^1[pc^1] = m^2[pc^2])$, which we assume to hold throughout the execution. Next, we want to learn an inductive invariant over $sIMPL \times_\phi sIMPL$ under assumption $\Box\, \Phi_{ctr-equiv}$. We pick the following set of candidate invariants.

$$CI = \begin{Bmatrix} pc^1 = pc^2, st^1 = st^2, res^1 = res^2, ret^1 = ret^2 \\ st^1 = 0 \rightarrow ret^1 = 1, st^1 = 1 \rightarrow ret^1 = 1 \end{Bmatrix}$$

Procedure LearnInv starts by checking the invariant candidates on the initial state. We set bound b to 1. Since in all valid initial states μ, we have $\mu(pc) = 1$, $\mu(st) = 0$, and all microarchitectural variables are assumed to be equal via $\Phi_{initial}$ we retain all candidate invariants. Next, LearnInv checks whether the candidate invariants are preserved under transitions. That is, if we assume the invariant holds and take a transition step, the invariant must still hold. Since our invariant does not require memory m to be equal in both executions, taking the else branch in line 3 of *sIMPL* (see Example 4) produces a counterexample where $res^1 \neq res^2$ and we remove the corresponding invariant. Similarly, taking the else branch in line 3 produces a state where $st^1 = 1$ and $ret^1 = 0$ and LearnInv removes the invariant as well. The remaining candidate invariants are preserved under transitions and the procedure returns. This leaves us with the following set of learned invariants.

$$LI = \begin{Bmatrix} pc^1 = pc^2, st^1 = st^2, ret^1 = ret^2 \\ st^1 = 0 \rightarrow ret^1 = 1 \end{Bmatrix}$$

Finally, procedure Verify checks whether the conjunction of the learned invariants implies that attacker observations and retirement are the same in both executions. For our example, this means checking that the following implication holds.

$$\begin{pmatrix} pc^1 = pc^2 \wedge st^1 = st^2 \wedge \\ ret^1 = ret^2 \wedge st^1 = 0 \rightarrow ret^1 = 1 \end{pmatrix} \rightarrow \begin{pmatrix} pc^1 = pc^2 \wedge \\ (ret^1 = 1) \leftrightarrow (ret^2 = 1) \end{pmatrix}$$

As the implication is valid, we have proved microarchitectural contract satisfaction.

5 IMPLEMENTATION

In this section, we present the LeaVe verification tool, which implements the verification approach from §4.2 for Verilog. LeaVe uses the Yosys Open Synthesis Suite [8] for processing Verilog circuits, the Icarus Verilog simulator [4] for simulating counterexamples, and the Yices SMT solver [7] for verification. LeaVe is open source and available at [6] together with the benchmarks and scripts for reproducing the experiments from §6.

Inputs: LeaVe takes as input (1) the processor under verification (PUV) *Impl* implemented in Verilog, (2) a leakage monitor formalized as Verilog expressions over *Impl*'s architectural state, (3) an attacker expressed as Verilog expressions over *Impl*, (4) a retirement predicate ϕ expressed as a Boolean condition over *Impl*, and (5) a lookahead $b \in \mathbb{N}^+$.[6] Users can provide candidate relational invariants as expressions e over *Impl* and LeaVe will construct the candidate invariant $e^1 = e^2$. Users can also provide additional invariants over individual executions of *Impl* to help ruling out spurious counterexamples.[7]

Workflow: LeaVe works in two steps that follows Algorithm 1.

First, LeaVe determines the greatest subset of the provided candidate relational invariants that is inductive. For this, LeaVe implements the LearnInv function from Algorithm 1 (described below). In addition to the user provided candidate invariants, the set of candidate invariants for LearnInv contains: (1) *all* relational formulas of the form $x^1 = x^2$ where x is a register or wire in *Impl*, (2) formulas of the form $e_{Atk}{}^1 = e_{Atk}{}^2$ for all expressions e_{Atk} in the provided attacker, and (3) the invariant $\phi^1 \leftrightarrow \phi^2$ indicating that the retirement predicate is always synchronized between the two executions.

Next, LeaVe analyzes the learned invariants to determine if they are sufficient to prove security with respect to the given attacker. For this, LeaVe checks if the invariants associated with the attacker and with the retirement predicate are part of the set of learned invariants, which is sufficient to ensure the satisfaction of the check at line 5 in Algorithm 1.

Implementation of LearnInv: LeaVe's implementation of LearnInv follows Algorithm 1: (1) It constructs the stuttering product circuit by combining two copies of the PUV and using the

[6] As a rule of thumb, a sufficient choice for b is the maximum number of cycles needed for an instruction to traverse the pipeline (from fetch to retire).

[7] LeaVe only verifies the relational invariants, which concern security. Invariants over *Impl*, which concern functional correctness, are assumed and not checked by the tool.

provided retired predicate ϕ to synchronize the two executions (as described in §4.2). (2) Then, it inlines the property to be verified (*i.e.*, Ψ_{base} and $\Psi_{inductive}$ from Algorithm 1) as `assume` and `assert` Verilog statements in the product circuit. (3) Next, it checks whether the property holds. (4) Whenever a property is not satisfied, LeaVe analyzes the counterexample to determine which candidate relational invariants are violated (lines 12-13 and 19-20 in Algorithm 1)

For (1) and (2), we implemented dedicated Yosys passes that construct the stuttering product circuit and inline candidate relational invariants. For (3), LeaVe uses Yosys to encode the product circuit and the verification queries into SMT logical formulas and the Yosys-BMC [8] backend to verify the property with the Yices SMT solver (using the lookahead b as verification bound). For (4), when verification fails, Yosys-BMC translates the SMT counterexample into a Verilog testbench. LeaVe instruments the testbench to monitor the value of all candidate invariants, simulates the testbench using Icarus Verilog, and discards the violated invariants.

6 EVALUATION

This section reports on our use of LeaVe to verify the security of three open-source RISC-V processors. We start by introducing our methodology (§6.1): the processors we analyze, the leakage contracts and attacker we consider, and the experimental setup. In our experimental evaluation (§6.2), we address the following three research questions: **Q1**: Can LeaVe be used to reason about the security of open-source RISC-V processors? **Q2**: What is the impact of varying the lookahead b on verification time? **Q3**: What is the impact of decoupling security and functional correctness on verification?

6.1 Methodology

Benchmarks: We consider the following benchmarks.

- **RE**: The simple processor from §2. The `log_time_mul` module is implemented using shift operations (logarithmic in the number of set bits of the multiplier), inspired by one of Ibex's multipliers [3].
- **DarkRISCV**: A RISC-V processor implementing most of the RISC-V RV32E and RV32I instruction set [1]. The processor is in-order and single-issue, and we analyzed its 2-stage (**DarkRISCV-2**) and 3-stage (**DarkRISCV-3**) versions.
- **Sodor**: An educational RISC-V processor [5]. We analyzed the 2-stage version of Sodor implementing the RV32I instruction set.
- **Ibex**: An open-source, production-quality 32-bit RISC-V CPU core [2].[8] We target Ibex in its default configuration (called "small" [2]), which underwent functional correctness verification. The processor has two stages and supports the RV32IMC instruction set. In our experiments, we consider three variants of Ibex: (1) **Ibex-small** is the default "small" configuration with constant-time multiplication (three cycles) and without caches, (2) **Ibex-cache** is the **Ibex-small** version 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 [3].

For the RISC-V processors in our experiments (*i.e.*, all variants of **DarkRISCV**, **Sodor**, **Ibex**), we make the following assumptions during verification: (1) debug mode is disabled, (2) all fetched instructions are legal and not compressed, (3) no exceptions or interrupts are raised during execution, and (4) only unprivileged instructions are executed. Additionally, for **Ibex-cache**, we assume that memory operations are aligned at word boundaries due to limitations of our simple cache implementation. Finally, for all processors we manually specify a retirement predicate indicating when instructions retire.

Leakage contracts: We consider leakage contracts constructed by composing the following building blocks:

- **I**: This contract exposes the architectural program counter and the corresponding instruction retrieved from memory.
- **B**: This contract exposes the architectural outcome of (direct and indirect) branch instructions. That is, for conditional branches, the contract exposes the architectural value of the condition.
- **M**: This contract exposes the addresses accessed by load and store memory instructions.
- **A**: This contract exposes whether load and store memory instructions are aligned.
- $\mathbf{O_m}$: This contract exposes the operands of `mul` and `imul` multiplication instructions.
- $\mathbf{O_d}$: This contract exposes whether the divisor in `div` (division) and `rem` (remainder) instructions is 0.

In the following, we write **A**+**B** to denote the composition of contracts **A** and **B**. For instance, **I**+**B**+**M** is the contract that exposes everything exposed by **I**, **B**, and **M**. This contract corresponds to the standard constant-time model [9]. We order contracts by the amount of information they leak, where stronger contracts leak less. For example, **I** is stronger than **I**+**B** as it exposes less information. For each processor from §6.1, we implemented all the above mentioned contracts and their combinations as leakage monitors over the processor's architectural state.

Attacker: For all processors from §6.1, we implemented an attacker monitor that observes when instructions retire by exposing the value of the retirement predicate at each cycle.

Additional candidate invariants: For **DarkRISCV**, **Sodor**, and **Ibex**, we manually specified candidate relational invariants capturing that "if instructions enter a pipeline stage in both executions, then the instructions are the same in both executions". Moreover, for **Ibex-cache**, we also added a candidate invariant capturing that "if both executions are executing a load instruction, then the signals detecting a cache hit are the same." All these invariants can be formalized as formulas of the form $e^1 = e^2 \rightarrow e'^1 = e'^2$. These are not part of the invariants automatically generated by LeaVe, which are of the simpler form $e^1 = e^2$.

Experimental setup: All our experiments are run on a Ubuntu 20.04 virtual machine with 8 CPU cores and 32 GB of RAM running on Linux KVM on a server with 4 Xeon Gold 6154 CPUs and 512 GB of DDR4 RAM. We configured LeaVe to run with Yosys version 0.24 + 10, Icarus Verilog version 12.0, and Yices version 2.6.4.

6.2 Experimental results

Q1: Reasoning about open-source processors: To evaluate whether LeaVe can verify the security guarantees of open-source processors, we use it to prove microarchitectural contract satisfaction against an attacker Atk that observes when instructions are

[8] Ibex is written in SystemVerilog. To analyze it with LeaVe, we first translate it into plain Verilog using scripts from Ibex's developers.

retired. For each processor and leakage monitor from §6.1, we use LeaVe to check whether the attacker monitor leaks less than the leakage monitor with respect to the processor and its retirement predicate (indicating whenever instructions retire).

Table 1 reports (1) the strongest contract that could be verified against Atk, (2) the time needed for the verification of the satisfaction of the strongest contract, (3) the total number of iterations taken by LearnInv for the base and induction steps (i.e., the number of issued SMT queries), and (4) the minimum lookahead b for which verification succeeded. We highlight the following findings:

- For the **RE** processor from §2, LeaVe successfully verified contract satisfaction against the contract $\mathbf{O_m}$ exposing the multiplication's operand in 1.5 minutes with a lookahead of 33. Such a lookahead is needed to ensure that in-flight multiplications are retired and the corresponding contract observation is produced.
- For **DarkRISCV**, LeaVe proves contract satisfaction against the **I** contract, which exposes the current program counter and the loaded instruction, in 7 minutes for the two-stage version **DarkRISCV-2** and in around 11 minutes for the (more complex) three-stage version **DarkRISCV-3**.
- Differently from **DarkRISCV**, **Sodor-2** only satisfies the weaker **I**+**B** contract, which additionally exposes the outcome of branch instructions. This arises from the processor employing a simple form of branch prediction, which predicts that the branch is always not taken. This results in a timing leak because mispredictions trigger a pipeline flush. Consider the following instruction (returned by LeaVe as a counterexample when trying to prove satisfaction against **I**) $i \triangleq \texttt{beq}\ t_1\ t_2\ pc+4$ at address pc, which conditionally jumps to $pc+4$ if registers t_1 and t_2 have the same value. The next instruction will *always* be the one at address pc+4 (so, executions will be equivalent under contract **I**). However, executing i on **Sodor-2** takes a different number of cycles depending on whether t_1 and t_2 are equal.
- For **Ibex-small**, LeaVe can only prove security against the **I**+**B**+$\mathbf{O_d}$+**A** contract, which additionally exposes (a) whether the divisor in division and remainder instructions is 0 and (b) whether memory accesses are aligned. The $\mathbf{O_d}$ is needed to capture that division and remainder operations take 1 cycle when the divisor is 0 or 37 cycles otherwise. Moreover, the **A** contract is needed to capture that **Ibex** handles memory accesses that are not aligned on word boundaries by performing two separate word-aligned memory accesses. Note that the difference in complexity between **Sodor-2** (a simple educational processor) and **Ibex-small** (a production-quality processor) is reflected in the difference in the time taken by a single LearnInv iteration (1.1 versus 16 minutes on average) and by the larger lookahead (1 vs 38).
- For **Ibex-cache**, LeaVe can only prove security against the **I**+**B**+$\mathbf{O_d}$+**M** contract. Differently from **Ibex-small**, which is secure against the **I**+**B**+$\mathbf{O_d}$+**A** contract, **Ibex-cache** needs the **M** contract that exposes the accessed memory addresses (rather than the alignment bit). This reflects the effects of our single-line cache which requires 3 cycles for hits and 4 cycles for misses.
- For **Ibex-mult-div**, LeaVe can only prove security against the **I**+**B**+$\mathbf{O_d}$+$\mathbf{O_m}$+**A** contract, which also exposes the operands of multiplication instructions ($\mathbf{O_m}$). This captures the effects of the non-constant-time multiplier used in **Ibex-mult-div**, whose execution time is proportional to the logarithm of the multiplication operands.

Table 1: Verification results for our benchmarks. For each processor, the table indicates the strongest satisfied contract (i.e., the one exposing the least amount of information) against an attacker observing when instructions retire.

Processor	*Strongest contract*	*Verification time (in minutes)*	LearnInv *iterations*	b
RE	$\mathbf{O_m}$	1.5	10	33
DarkRISCV-2	**I**	7.2	52	2
DarkRISCV-3	**I**	11.1	83	2
Sodor-2	**I**+**B**	97.8	85	1
Ibex-small	**I**+**B**+$\mathbf{O_d}$+**A**	1479.4	90	38
Ibex-cache	**I**+**B**+$\mathbf{O_d}$+**M**	1396.7	67	38
Ibex-mult-div	**I**+**B**+$\mathbf{O_d}$+$\mathbf{O_m}$+**A**	1291.9	75	38

Q2: Impact of lookahead: LeaVe's verification queries are parametric in the lookahead b. A larger lookahead corresponds to stronger assumptions and may thus enable learning stronger invariants. This, however, comes at the cost of more complex queries to the SMT solver, which increases solving time. To understand the impact of increasing b, we use LeaVe to analyze the **Sodor** processor against the **I**+**B** contract for different values of $b \in \{1, 2, 3, 5, 10, 20\}$.

Table 2 reports the total verification time, the total number of iterations taken by the LearnInv sub-procedure for the base and induction steps (i.e., the number of issued SMT queries), the time per iteration, and the number of invariants learned. Our results indicate that increasing the lookahead b results in slower iterations of LearnInv and in more invariants. For instance, increasing the bound from 1 to 20 results in increasing the iteration time from 1.15 to 8.29 minutes. The total number of LearnInv iterations (and, thus, the total verification time), however, varies depending on which counterexamples the SMT solver returns.

Q3: Impact of decoupling: To understand the impact of checking microarchitectural contract satisfaction using our decoupling theorem versus checking contract satisfaction directly using an architectural model *ISA* (c.f. Definition 2), we modified LeaVe to directly prove contract satisfaction according to Definition 2. For this, we (1) replace the construction of the stuttering circuit $\mathit{Impl} \times_\phi \mathit{Impl}$ with the product circuit $\mathit{ISA} \times \mathit{ISA} \times \mathit{Impl} \times \mathit{Impl}$ and (2) modify the construction of the $\Psi_{initial}$ and $\Psi_{contract}$ formulae, whereas the rest (e.g., the LearnInv procedure) is the same. We refer to this modified version of LeaVe as 4way-LeaVe (see [49, Appendix D]). Note that 4way-LeaVe and LeaVe prove different properties which, as stated in Theorem 2, are equivalent only for ISA-compliant designs.

We analyzed the **Sodor** processor against the **I**+**B** contract using both LeaVe and 4way-LeaVe. We focused our analysis on **Sodor** because it comes with a Verilog ISA model (i.e., 1-stage **Sodor**).

In our experiments, when using a lookahead of $b = 2$, LeaVe successfully proved that the **I**+**B** contract is satisfied in 97.8 minutes. In contrast, 4way-LeaVe tool proved contract satisfaction in 33.5 hours. This illustrates that, even for the simple 2-stage **Sodor** processor, directly proving Definition 2 is impractical. It also confirms that our decoupling theorem is instrumental in enabling practical automated proofs of contract satisfaction for realistic hardware.

Table 2: Verification time for Sodor against the I+B contract for different lookaheads b.

b	*Verification time (in minutes)*	*LearnInv iterations*	*Time per iteration (in minutes)*	*Number of invariants*
1	97.8	85	1.15	1732
2	81.9	63	1.30	1770
3	123.5	81	1.52	1776
5	102.9	59	1.74	1776
10	174.4	63	2.77	1776
20	497.6	60	8.29	1776

7 DISCUSSION

Limitations: Our formalization of leakage contracts and our notion of ISA compliance impact both the decoupling theorem (Theorem 1) as well as the microarchitectures and contracts supported by LeaVe. In terms of microarchitectures, our notion of ISA compliance (Definition 1) only applies to single-issue processors. Supporting multi-issue processors requires an ISA compliance notion that accounts for retiring multiple instructions per cycle. In terms of leakage contracts, LeaVe targets *sequential* leakage contracts that only refer to "architectural" instructions. We leave the support for leakage contracts that refer to transient instructions, like the speculative contracts from [27], as future work.

We also remark that (1) LeaVe currently lacks support for inputs, and (2) our formalization of attackers as combinatorial monitoring circuits limits LeaVe to reason about *passive attackers* that can only observe (part of) a processor's microarchitecture during execution. We leave corresponding extensions to future work.

Lookahead: LeaVe's verification approach is parametric in a lookahead $b \in \mathbb{N}^+$ which determines for how many cycles the contract-equivalence assumption needs to be unrolled in the verification queries issued by the LearnInv function in Algorithm 1. The lookahead b is used to expose contract observations (produced at retirement) for instructions that are in-flight. In particular, it allows accounting for microarchitectural differences at cycle i that are later declassified by a contract observation produced at cycle (at most) $i + b$. We remark that the choice of b does not affect the soundness of LeaVe (cf. Theorem 2), but it may affect the success of verification. For instance, verifying the satisfaction of the contract `LM` from §2 for the processor from Figure 2 requires $b = 33$ (see Table 1), *i.e.*, verification fails for smaller bounds. In our experiments, setting b to the maximum number of cycles needed for instructions to traverse the pipeline (from fetch to retire) was always sufficient whenever contract satisfaction holds.

Leakage contracts and secure programming: Leakage contracts may serve as a foundation for secure programming. As shown by Guarnieri et al. [27], ensuring at program level that secret data do not influence leakage contract traces is sufficient to ensure the absence of leaks at microarchitectural level for processors that satisfy the contract. Thus, LeaVe's verification results have direct implications for programmers. As an example, for each processor in our evaluation (§6), the strongest contract verified by LeaVe, reported in Table 1, indicates which parts of a computation should not involve secrets to ensure leakage freedom. For instance, secure programming for the contract **I**+**B**+$\mathbf{O_d}$+$\mathbf{O_m}$+**A**, satisfied by **Ibex-mult-div**, requires ensuring that secrets do not influence (i) the program's control-flow (**I**+**B**), (ii) whether the divisor in `div` and `rem` instructions is 0 ($\mathbf{O_d}$), (iii) the operands of `mul` and `imul` instructions ($\mathbf{O_m}$), and (iv) the alignment of memory accesses (**A**).

8 RELATED WORK

Hardware verification for security: UPEC [22] is an approach for detecting confidentiality violations in RTL circuits. Similarly to Definition 3, the UPEC property is defined as a non-interference-style property over pairs of microarchitectural executions. However, the security property verified in [22] is fixed; it specifically focuses on microarchitectural leaks due to transitive execution; and it is not directly based on an ISA-level specification, *i.e.*, it does not correspond to a leakage contract in a straightforward manner. In contrast, our approach directly supports leakage contracts defined at ISA-level.

Bloem et al. [13] propose an approach for verifying power leakage models (formalized on top of the Sail domain specific language [10]) for RTL circuits, which differs from LeaVe in two key ways: (1) They target power side channels, whereas LeaVe focuses on software-visible microarchitectural leaks. This is reflected in different notions of contract satisfaction: the one from [13] is probabilistic and related to threshold non-interference, whereas ours is related to standard non-interference. (2) Their verification approach needs a user-provided simulation mapping that "specifies for all registers in the hardware [..] a location in the contract modeling the hardware location" [13, §3.4] where a location is a register or an input. Defining such a mapping can be challenging for complex processors, *e.g.*, registers of stateful microarchitectural components (like caches or predictors) may depend on multiple instructions. LeaVe does not need such a mapping for checking contract satisfaction; it only needs (automatically synthesized or manually provided) candidate relational invariants over the microarchitecture.

Iodine [48] and Xenon [45] check if the execution time of an RTL circuit is *input independent* given a partitioning of the circuit's inputs into secret and public. This partitioning is too coarse to support leakage contracts, where the notion of what is "secret" depends on the executed instructions. Finally, secure Hardware Description Languages [20, 53] aim at building secure processors by construction. They require partitioning RTL registers and inputs into secret and public, which is too coarse-grained for leakage contracts.

Knox [11] is a verification approach for hardware security modules (HSMs) that targets an HSM's hardware and software components. While leakage contracts capture a processor's security guarantees at ISA level, Knox focuses on ensuring that all components of an HSM are *both* functionally correct and leakage free. Differently from LeaVe, Knox relies on a combination of annotations and interactive proofs.

Hardware verification for functional correctness: A multitude of approaches for verifying functional correctness of processors have been proposed [16, 28, 30, 33, 41, 44, 52]. Some of these approaches adopt a notion of ISA compliance similar to Definition 1. For instance, Reid et al. [44] illustrate a verification approach (used internally at ARM) for checking compliance between a microarchitecture and a reference architectural model, where the notion

of ISA compliance requires that all changes to the architectural state are reflected by a "step" of the reference model (similarly to Definition 1).

The Instruction-Level Abstraction (ILA) project [28, 52, 54] aims to specify and verify instruction-level models of processors and accelerators. They present techniques for (1) checking whether an RTL implementation correctly implements an ILA model, (2) determining which parts of a processor's state are architectural [52], and (3) deriving processor invariants [54]. Some of these techniques can help in LeaVe's verification. For instance, [52] can help in identifying the *Arch* and *μArch* sets, whereas [54] can complement LeaVe's invariant learning approach.

Finally, fuzzing approaches [17, 31] can detect violations of ISA compliance, but they cannot prove functional correctness.

Detecting leaks through testing: Revizor [39, 40] and ScamV [14, 38] search for contract violations (*i.e.*, they find counterexamples to Definition 2) for black-box CPUs. However, they require physical access to a CPU and can be applied only post-silicon. Other approaches [24, 35, 50] instead detect leaks by analyzing hardware measurements without the help of a formal leakage model but, again, apply only post-silicon. Finally, SpecDoctor [29] and SigFuzz [43] can test for leaks on RTL designs and they are applicable in the pre-silicon phase. Differently from LeaVe, all these approaches *cannot* prove the absence of leaks.

Formal leakage models: Researchers have proposed many formal models for studying microarchitectural security at program level, ranging from simple models associated with "constant-time programming" [9, 36] to more complex ones capturing leaks associated with speculatively executed instructions [18, 21, 25, 26, 42, 47]. Most of these models focus at the software level and have no formal connection with leaks in hardware implementations. In contrast, [27, 37] propose frameworks for formalizing security contracts between hardware and software. Our notion of contract satisfaction (Definition 2) is inspired by the framework from [27], which we instantiate and adapt for reasoning about RTL processors.

9 CONCLUSION

We presented an approach for verifying RTL processor designs against ISA-level leakage contracts. We implemented our approach in the LeaVe verification tool, which we use to characterize the side-channel security guarantees of three open-source RISC-V processors. This demonstrates that leakage contracts can be successfully applied to RTL processor designs. It also paves the way for linking recent advances on specification [27, 37] and software analysis [18, 21, 25, 26, 47] for leakage contracts to RTL processor designs.

ACKNOWLEDGMENTS

We would like to thank Alastair Reid and Piotr Sapiecha for feedback and discussions. This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415), 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 a gift by Intel Corporation.

REFERENCES

[1] [n. d.]. DarkRISCV processor. https://github.com/darklife/darkriscv
[2] [n. d.]. Ibex: An embedded 32 bit RISC-V CPU core. https://github.com/lowRISC/ibex
[3] [n. d.]. Ibex RISC-V Core – Multiplier/Divider Block. https://ibex-core.readthedocs.io/en/latest/03_reference/instruction_decode_execute.html#mult-div
[4] [n. d.]. Icarus Verilog. https://github.com/steveicarus/iverilog
[5] [n. d.]. RISC-V Sodor processor. https://github.com/ucb-bar/riscv-sodor
[6] [n. d.]. LeaVe implementation and benchmarks. https://github.com/zilongwang123/LeaVe
[7] [n. d.]. Yices 2 SMT Solver. https://yices.csl.sri.com
[8] [n. d.]. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys
[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*.
[10] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (2019).
[11] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *OSDI*.
[12] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011).
[13] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. 2022. Power Contracts: Provably Complete Power Leakage Models for Processors. In *CCS*. https://doi.org/10.1145/3548606.3560600
[14] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. 2021. Validation of Side-Channel Models via Observation Refinement. In *MICRO-54*. ACM. https://doi.org/10.1145/3466752.3480130
[15] Jo Van Bulck et al. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.
[16] Jerry R. Burch and David L. Dill. 1994. Automatic verification of pipelined microprocessor control. In *CAV*.
[17] Sadullah Canakci, Chathura Rajapaksha, Anoop Mysore Nataraja, Leila Delshadtehrani, Michael Taylor, Manuel Egele, and Ajay Joshi. 2022. ProcessorFuzz: Guiding Processor Fuzzing using Control and Status Registers. *arXiv preprint arXiv:2209.01789* (2022).
[18] Sunjay Cauligi, Craig Disselkoen, Klaus V. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *PLDI*. https://doi.org/10.1145/3385412.3385970
[19] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010).
[20] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. 2019. SecChisel framework for security verification of secure processor architectures. In *HASP@ISCA*.
[21] Xaver Fabian, Marco Patrignani, and Marco Guarnieri. 2022. Automatic Detection of Speculative Execution Combinations. In *CCS*.
[22] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2022. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Trans. Comput.* 72, 1 (2022).
[23] Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *FME*.
[24] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*.
[25] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *CCS*.
[26] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *IEEE S&P*.
[27] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *IEEE S&P*.
[28] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2019. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM TODAES* 24, 1 (2019). https://doi.org/10.1145/3282444
[29] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities. In *CCS*.

[30] Ranjit Jhala and Kenneth L McMillan. 2001. Microarchitecture verification by compositional model checking. In *CAV*. Springer.
[31] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *USENIX Security*.
[32] 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 *IEEE S&P*. https://doi.org/10.1109/SP.2019.00002
[33] Ulrich Kühne, Sven Beyer, Jorg Bormann, and John Barstow. 2010. Automated formal verification of processors based on architectural models. In *FMCAD*.
[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.
[35] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis Background Superscalar Memory Architecture. In *USENIX Security*.
[36] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC*. 156–168.
[37] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic hardware-software contracts for security. In *ISCA*.
[38] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. 2020. Validation of Abstract Side-Channel Models for Computer Architectures. In *CAV*.
[39] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *ASPLOS*.
[40] 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 *IEEE S&P*.
[41] V.A. Patankar, A. Jain, and R.E. Bryant. 1999. Formal verification of an ARM processor. In *Proceedings Twelfth International Conference on VLSI Design*. https://doi.org/10.1109/ICVD.1999.745161
[42] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *CCS*.
[43] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. 2023. SIGFuzz: A Framework for Discovering Microarchitectural Timing Side Channels. In *DATE*. https://doi.org/10.23919/DATE56975.2023.10136966
[44] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-end verification of processors with ISA-Formal. In *CAV*.
[45] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2021. Solver-Aided Constant-Time Hardware Verification. In *CCS*.
[46] Stephan van Schaik et al. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.
[47] Marco Vassena et al. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. In *POPL*.
[48] Klaus von Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *USENIX Security*.
[49] 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. *CoRR* abs/2305.06979 (2023).
[50] Daniel Weber et al. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security*.
[51] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*.
[52] Yu Zeng, Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. 2021. Generating Architecture-Level Abstractions from RTL Designs for Processors and Accelerators Part I: Determining Architectural State Variables. In *IEEE/ACM ICCAD*. https://doi.org/10.1109/ICCAD51958.2021.9643584
[53] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*. https://doi.org/10.1145/2694344.2694372
[54] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. 2020. Synthesizing environment invariants for modular hardware verification. In *VMCAI*.