

VerIFI: Formal Verification of Microarchitectural Information-Flow Integrity

Katharina Ceesay-Seitz
ETH Zurich
Switzerland
kceesay@ethz.ch

Flavien Solt
UC Berkeley
USA
flavien.solt@berkeley.edu

Mengyuan Yin
ETH Zurich
Switzerland
menyin@student.ethz.ch

Kaveh Razavi
ETH Zurich
Switzerland
kaveh@ethz.ch

Abstract—Fundamental security properties such as confidentiality and integrity are violated in CPUs due to unintended or missing information flows. We introduce Information-Flow Integrity (IFI), a CPU property that formalizes the legal information flows between architectural hardware elements, as derivable from the ISA. Ensuring IFI enables the systematic detection of both microarchitectural vulnerabilities and functional bugs that compromise confidentiality or integrity. To concretely verify IFI on a CPU implementation, we introduce VerIFI, a verification method that captures IFI by proving four key properties for the existence or absence of certain information flows. Specifically, VerIFI ensures that data is appropriately *fresh*, *masked* by non-tainted data, and that information flows are *contained* within the originating architectural elements, and *confined* to the correct architecturally-visible state elements across space and time. VerIFI introduces a precise spatio-temporal path gating technique that supports controlled introduction and declassification of information flows, accommodating microarchitectural optimizations and known exceptions. Applying VerIFI to two verified RISC-V CPUs, Kronos and Ibex, we confirm known vulnerabilities and discover two new ones.

Index Terms—information flow tracking, timing side channels, hardware security

1. Introduction

The growing number of information leakage and integrity vulnerabilities plague modern Central Processing Units (CPUs). References [1]–[27] demonstrate the need for the correct handling of information flows in hardware. Existing verification methods capture information flows only in part, either at the microarchitectural [25], [28]–[31] or functional level [32]–[34]. As such, we currently lack a method that fully specifies and verifies correct information flows in a CPU for a given Instruction Set Architecture (ISA). This paper introduces Information-Flow Integrity (IFI), a novel CPU property that enables the augmentation of ISAs with information flow specifications and a formal verification method that captures this property.

Information-Flow Violations. Both commercial errata [35] and open-source bug reports [24], [25], [36]–[38] reveal frequent Information Flow (IF) violations. Such vulnerabilities compromise fundamental confidentiality or integrity properties in the CPU itself, violating how information should propagate as implicitly defined by the ISA. Vulnerabilities that violate confidentiality enable an attacker to leverage timing side channels or architectural bugs to exfiltrate information. Vulnerabilities that compromise

integrity typically lead to a full compromise of the system, e.g., by hijacking the architectural control flow [24], [25]. Formal verification has the unique advantage of being able to prove the absence of bugs, hence covering all possible execution paths and corner cases, in which such vulnerabilities often lurk. Our aim is to define a formal property such that its verification can prove confidentiality and integrity of the information flows as expected by software.

Information-Flow Integrity. We specify Information-Flow Integrity (IFI), a novel CPU security property that formalizes how information is allowed to flow within a CPU. IFI augments existing ISAs with information flow policies by mapping bit-precise taint propagation rules from hardware Information Flow Tracking (IFT) methods [39]–[42] to RISC-V instructions. IFI unifies the aspects of functional verification and constant-time verification that are relevant for IFT. A program executed on a CPU that is proven to comply with IFI is guaranteed not to expose more information than is allowed by the ISA. Similarly, such a program is also guaranteed not to violate the integrity of data beyond the manipulations that are allowed by the ISA. In other words, an IFI-compliant CPU is free from timing side channels and architectural data leaks or integrity violations. The IFI specification allows customizations for concrete microarchitectures, by introducing information propagation rules for microarchitectural optimizations like caches or prediction units. In contrast with existing verification methods, these customizations allow further verification after excluding selected paths. IFI turns implicit software assumptions into verifiable hardware guarantees.

VerIFI. VerIFI verifies that a concrete CPU implementation propagates information exactly as defined by Specification-derivable Information Flow (SIF) rules. We verify IFI in concrete CPU implementations by defining four novel hardware IFT properties that capture all the specified information flows: *freshness*, *masking*, *containment*, and *confinement*. These properties are expressed as IFT properties that dynamically track information flows between architecturally visible elements, such as General Purpose Registers (GPRs), Control and Status Registers (CSRs), memory, or the Program Counter (PC). *Freshness* ensures that data is overwritten whenever new data is supposed to be transported to a state element, preventing the use or leakage of stale data, e.g., as in microarchitectural data sampling attacks [8], [11], [43]. *Masking* conversely verifies that information does not propagate to an instruction's destination when it is supposed to be masked by non-

tainted data. *Containment* checks that information does not propagate between state elements, when no information movement is specified. Finally, *confinement* enforces that information does not propagate to any architecturally visible state elements, except for the ones specified by the SIFs, and only at times where the rules allow so. This captures architectural leakage attacks, like Zenbleed [44] or AEPICLeak [45], as well as timing side channels [1], [2], [46]–[48]. To avoid excessive blocking of information flows or verification scenarios, as often done by the state of the art [25], [28]–[30], [49], we introduce *path gating*, a new spatio-temporal precise taint injection and declassification method. This method gates the taint propagation at specified locations, under specified conditions, while restoring the original path otherwise.

We verify these four properties by augmenting the hardware design with IFT logic [41] and using an off-the-shelf model checker [50]. Our evaluation on Ibex and Kronos shows that even though both CPUs were previously verified for timing side channels and functional bugs, they still contain two IFI violations, which we confirmed are actual vulnerabilities. Most IFT properties were proven in roughly two hours per instruction.

Contributions. We make the following contributions:

- We adopt hardware IFT to augment ISAs with information flow semantics that specify bit-precise expected information propagation caused by instructions. We then specify IFI for different RISC-V instructions using these information flow semantics.
- We introduce VerIFI, a generic method for verifying IFI based on four IFT properties: *freshness*, *masking*, *confinement* and *containment*. We verify these properties by implementing a generic and precise spatio-temporal *path gating* method that enables custom declassification of information flows.
- We prove IFI or find counter examples by verifying two open-source in-order RISC-V CPUs. In particular, we find two new vulnerabilities: one in Kronos that has previously undergone formal verification and fuzz testing [24], [25] and one in Ibex that has already been extensively verified [25], [28], [29], [51], [52].

We **open sourced** VerIFI at <https://github.com/comsec-group/verifi>.

2. Background

We provide some background on hardware verification and Information Flow Tracking (IFT).

2.1. Verification of hardware

Functional verification. Functional hardware verification typically compares a clock-cycle accurate hardware design at Register Transfer Level (RTL) against a higher-level reference model. This can be done in RTL simulation [24], [38], [53], or by formal verification [32]–[34], [54]. The latter verifies properties exhaustively for all unbounded design input sequences [55], [56]. Complete functional formal verification is expensive. It often requires human effort for 1) writing invariants tied to one specific design version, 2) building custom design-specific tooling and 3) solving verification scalability problems in creative design-specific ways, like instantiating multiple copies of the load-store unit to model different arrival times of data [34]. Moreover, a full functional reference model

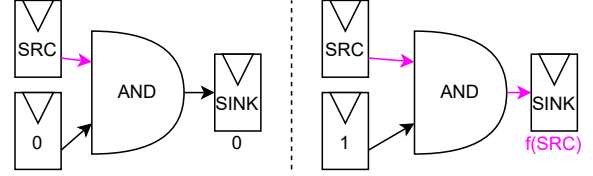


Figure 1: Information flows may exist depending on values of the rest of the circuit state. Left: no information flow exists between the source (SRC) and the sink of an and gate. Right: an information flow exists between the source and the sink in the next cycle. Sink becomes a function of SRC.

is rarely available. This is especially true when CPUs implement custom ISA extensions [34], which is one main advantages of using the RISC-V ISA. Verifying the interaction of instructions, i.e., IFs between them, is particularly challenging [34], and thus often not fully verified [29], [30], [32], [33].

2.2. Information flow tracking

Incorrect or unintended information propagation can cause both functional bugs and security vulnerabilities. If information propagates from a secret-storing component to an adversary-observable component, this threatens confidentiality. If information propagates from an adversary-controlled component to a component whose value should be independent of adversary influence, this threatens integrity. If insufficient information propagates during a computation, this might be the sign of a correctness issue. Further, it can cause stale data to be used by later operations. IFT monitors information propagation.

Hardware information flows. In this paper, we focus on information flows between RTL state signals, e.g., registers or memory words, and the time is measured at clock-cycle granularity. We say that information propagates between a source and a sink element if the value at the sink (sampled at clk_{sink}) is a non-constant function of the value at the source (sampled at clk_{source}). Note that the existence of an IF depends on the rest of the circuit’s input and states, as illustrated in Figure 1. Later in this paper, we will refer to these IFs as hardware IFs.

Hardware IFT. Hardware IFT methods track such IFs. Typical implementations operate at the level of logic gates [39] or higher-level hardware components [41], [42]. The taint propagation formula corresponding to an and gate is defined as in Equation 1 [39], where a and b are the inputs, and y is the output. The notations $a^t/b^t/y^t$ denote, if valued to 1, the taintedness of the respective signals.

$$y^t = [a^t \wedge (b \vee b^t)] \vee [(a \vee a^t) \wedge b^t] \quad (1)$$

Hardware IFT implementations typically instrument the design under test with shadow logic that does not interfere with the regular operation of the design, but allows to track the influence of a data source on a data sink [39]–[42]. Taint represents secret data in confidentiality scenarios, or adversary-controlled data in integrity scenarios [57], [58].

3. Motivation

A large fraction of real-world CPU bugs can be understood as violations of expected information flows, either excessive or missing. This results in information leakage [1]–[18] or integrity violations [19]–[21], [25], [59]. Figure 2 summarizes instances of known bugs in terms of expected and actual IFs.



Figure 2: Architectural IFs from top (before instruction execution) to bottom (after instruction execution). The boxes represent architecturally visible state elements: Control and Status Registers (CSRs), General Purpose Registers (GPRs), memory (Mem) and the Program Counter (PC). Solid black lines are expected architectural flows. Dashed red lines indicate *missing* IFs. (a) Expected flow for an add instruction. (b) Intel’s erratum BV84 for a REP MOVSB instruction [35]. (c) A multiplication instruction in Ibex writes to the wrong destination [25]. (d) AMD’s erratum 699 for an VMLoad instruction [35]. (e) Kronos CVE 2023-51973 for an add instruction [25]. (f) Intel’s erratum AAT106 for a VMPTRLD instruction [35]. (g) AMD’s erratum 1321 for a WRMSR instruction [35]. (h) AMD’s erratum 721 [35]. (i) ProcessorFuzz bug 7 [27]. (j) BOOM issue 503. (k) VexRiscv bug CVE 2023-34896 [24]. (l) AMD’s erratum 719 [35].

Figure 2(a) represents the expected architectural IFs for an addition: information should flow from some source GPRs to a specified destination GPR. No other architectural state (e.g., configurations such as privilege level or physical memory protection) should affect the architectural information flow generated by this instruction. If architectural data accessed by an instruction propagates further into the microarchitecture than expected and eventually influences a future architectural state, then confidentiality is breached. Such an information leakage could happen via an explicit flow, where some architectural state is directly written to [25], [53], or implicitly, e.g., by changing the update time of a microarchitectural element such as the cache [1], [2], [8], [11], [12], [17], [43] or the PC [25], [28]–[31], [49], [51], [60]. Figure 2 further highlights different IF-violating CPU bugs found in commercial as well as open-source CPUs. Black arrows represent IFs expected by the ISA, while red arrows represent the unexpected or missing IFs that occurred due to a bug in a specific implementation.

Excessive IFs. Figure 2(b) depicts a bug in Intel CPUs that inadvertently pollutes the contents of three GPRs under specific circumstances when an interrupt occurs during a REP MOVSB instruction [35]. Case (c) depicts the flow caused by Ibex bug CVE-2024-28365 where unexpected bus behavior caused multiplication instructions to write to a wrong GPR [25]. (d) shows a case where under certain conditions virtualization related instructions write to CSRs unexpectedly. Bug (e) appeared in Kronos [61]. An

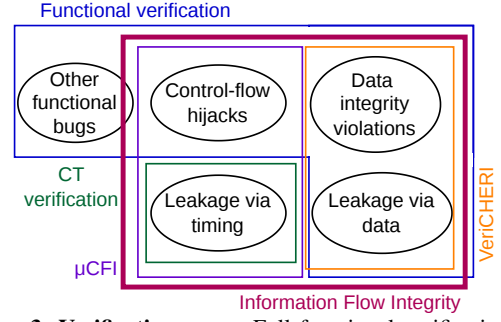


Figure 3: Verification scope. Full functional verification misses leakage via timing. Constant Time (CT) verification focuses on timing side channels exclusively. μ CFI [25] captures CT and control-flow hijacks. VeriCHERI [63] focuses on confidentiality and integrity for CPUs with capabilities.

addi instruction was able to pass its operand unexpectedly to a CSR instruction which mistakenly writes it into a CSR, enabling control-flow hijacks and timing side channels [25]. (f) describes a bug in Intel CPUs where a virtualization instruction corrupts memory in certain cases [35]. (g) depicts a bug in AMD CPUs where a write to a machine status register caused an unexpected exception, happening only when changing a specific bit from 0 to 1 [35]. This bug affects the update time of the PC and thus leaks the data via a timing side channel. (h) CVE 2024-44927 in Kronos causes jump instructions to operate on wrong source GPRs and thus either leak these GPRs via an architectural IF through the PC or enable a control-flow hijack attack [25]. Of all the bugs we analyzed, we observe that the unexpected or missing information flow systematically becomes visible in architectural elements.

Observation 1. Real-world bugs related to excessive IFs are plentiful and diverse.

Absent IFs. Conversely, bugs may arise from missing IFs. These often leave stale data in state elements, which then propagate excessively in later computations. Figure 2, bug type (i) was found in the open-source RISC-V BOOM CPU, where it fails to propagate information from floating point registers into a CSR [62]. (j) depicts Intel’s CML100 erratum, which describes a bug present in the generations 6 to 10 of Intel Core CPUs due to a missing information flow from performance counters to a model-specific register. Bugs in category (k) like CVE-2023-34896 in VexRiscv prevent correct instruction executions and thus do not propagate information between GPRs [24]. Bugs like erratum 719 of AMD (l) or CVE 2023-34914 in Picorv32 miss to transport CSR information into GPRs. In these cases, older instruction’s results, i.e., stale data, would not be overwritten as expected and would thus leak. We observe the following:

Observation 2. Information leakages or use of stale data is often caused by *missing* information flows.

3.1. Extending the verification scope

Figure 3 compares verification methods grouped by types of bugs that they cover. Functional verification implicitly catches IF violations if they cause an architecturally-observable effect, e.g., by propagating unexpected data to a software observable state element like memories or registers. Many vulnerabilities, however, leak information

via timing side channels. For example, they may manifest as a different update time of the Program Counter (PC) [25], [28], which is not considered by functional verification. Dedicated Constant Time (CT) verification methods focus only on information leakage via timing [25], [28]–[31], [49], [51], [60]. Neither functional, nor CT verification methods capture all IF violations. Many arise from subtle instruction interactions, where one instruction prepares microarchitectural state [8], [11], [12], and another instruction later exploits it. Functional verification often explicitly excludes the verification of such interactions between instructions for scalability reasons [32], [33]. However, to capture IF violations over unbounded sequences, verifying these interactions is essential [25], [34].

Observation 3. We need a verification method that captures confidentiality and integrity violations, and thus also guarantees the assumptions made by software IFT policies.

VerIFI specifically focuses on such interactions by verifying data leakages across instructions in unbounded, arbitrarily long, sequences. VerIFI combines the IFT relevant aspects of functional verification with timing side channel detection in a generic manner. Hence, VerIFI can be easily adapted to different ISAs, custom ISA extensions, CPU implementations and versions. In particular, VerIFI provides a mean to verify the functional aspects of IFI for ISA extensions where no formal reference model is available, such as data-independent timing [64], [65].

3.2. Threat model

We assume a RISC-V CPU which is executing the victim software, which can be processing sensitive information (e.g., cryptographic keys) or be exposed to malicious input (e.g., through system calls or the network). The aim of the attacker is exploiting a vulnerability in the CPU to compromise the confidentiality or integrity of information in the victim software. Compromising the confidentiality of the information can happen through leakage either architecturally, or microarchitecturally via a timing channel. Compromising the integrity of the information (either data or instructions) can lead to data- or control-flow corruption in the victim software. We assume that the attacker does not have architectural access to the information: they cannot read or modify the architectural state that, according to the ISA, contains this information. IFI aims to detect these CPU vulnerabilities that ultimately result in the violation of confidentiality and integrity guarantees as depicted in Figure 2. We further assume that attackers may cause interrupts, but power side channels and other attacks that assume physical access are out of scope.

4. Information-Flow Integrity

We define *Information-Flow Integrity (IFI)*, a property that compares Information Flows (IFs) within a CPU implementation with those expected from the ISA specification. This section defines two types of flows: Effective Information Flows (EIFs) and Specification-derivable Information Flows (SIFs). It then introduces formal semantics describing SIFs, and finally states the IFI property.

4.1. Two sorts of IFs

IFI enforces that all information flows happening in a CPU are expected from the ISA or CPU specification, and thus do not constitute a functional bug or a timing side channel.

Architectural model. We define the architectural model of a CPU as a finite-state machine, with its states corresponding to the architecturally visible storage elements defined by the ISA. We denote as $\mathbb{A}_{ISA}$ this set of architectural states. For example, in RISC-V, $\mathbb{A}_{ISA}$ is the union of the following components [66]:

- PC: The program counter, for example, readable through the `auipc` instruction or writable by `jalc`.
- R: The General Purpose Registers (GPRs), for example readable and writable through arithmetic instructions like `add`.
- CSR: The Control and Status Registers (CSRs), e.g., readable and writable through CSR instructions like `csrcw`.
- M: Memory, for example readable and writable through load and store instructions like `lw` and `sw`.

Note, for example, that caches, predictors and prefetchers are not part of $\mathbb{A}_{ISA}$, but of the microarchitectural state.

Executions. For a given CPU implementation, a given initial architectural state and a given instruction sequence $(i_n)_{n \geq 0}$, we define the *execution* $(A_n)_{n \geq 1}$ as the sequence of architectural states resulting from executing the instruction sequence on the CPU, where A_n is the architectural state after executing instruction i_n . For a state element $s \in \mathbb{A}_{ISA}$, $A_n(s)$ denotes the value stored in s in state A_n in the CPU’s implementation.

Effective Information Flows (EIFs). We define Effective Information Flows (EIFs) as hardware IFs (defined in Section 2) filtered to the granularity of architectural elements, e.g., GPRs, CSRs (spatial granularity) and instructions (temporal granularity). Hardware IFs operate at the finer granularity of clock cycles (temporal granularity) and registers (spatial granularity).

Specification-derivable information flows (SIFs). While the ISA generally does not explicitly specify IFs, it defines the effect of instructions on the values of the architectural elements (e.g., GPRs, CSRs or memory). An instruction induces an architectural IF whenever its outputs are a non-constant function of its inputs. Architectural states can further influence the behavior of instructions as specified by the ISA. We denote the set of all these architectural information flows as *specification-derivable* IFs (SIFs). In the following we will introduce IF semantics that allows the explicit specification of IFs per instruction.

4.2. SIF semantics

We introduce an abstract model that enables the definition of specification-derived IFs (SIFs) with SIF semantics.

Abstract SIF model. We formalize SIFs over an abstract ISA model, which augments the ISA with taint states that track how information should propagate under each instruction. We call this model the SIF model. We define SIF semantics over configurations, expressed as $\mathcal{M} = \langle R_t, CSR_t, M_t, PC_t, \mu_t, A, \mu \rangle$. $A = \langle R, CSR, M, PC \rangle$ models the architectural state. μ represents microarchitectural signals. $\mathcal{M}$ augments the model with one corresponding taint bit per state bit. For example, R_t refers to the taint bits for GPRs (R). This taint models explicit IFs caused by data movements and operations, as well as implicit flows via control paths or timing [25], [28], [30], [41], [42], [49], [51]. The model defines SIFs by augmenting instructions with SIF semantics. These specify how taint should propagate as a result of the execution

$$\text{ALU} \quad \frac{I(i_n) = \text{instr} \quad \text{rd}, \text{rs1}, \text{rs2} \quad t = C_{instr}^t(R[\text{rs1}], R_t[\text{rs1}], R[\text{rs2}], R_t[\text{rs2}])}{\langle R_t, \text{CSR}_t, M_t, \text{PC}_t, \mu_t, A, \mu \rangle \xrightarrow{\text{instr}} \langle R_t[\text{rd} \mapsto t], \text{CSR}_t, M_t, \text{PC}_t, \mu_t, A', \mu[c_{id} \mapsto i_n] \rangle}$$

Figure 4: An instruction has an IF to its destination in dependence of its inputs rs1 and rs2, as specified by taint tracking rule C_{instr}^t , detailed in [Table 1](#).

of an instruction starting with a specific combination of taint and design states. In the model, information only propagates from one architecturally visible state to another by an execution step with a matching SIF rule.

SIF semantics of instructions. We define SIF semantics that operate at instruction granularity. [Figure 4](#) shows the generic SIF rule for RISC-V ALU instructions. Let $I = (i_n)_{n \geq 1}$ be an infinite instruction stream, and $I(i_n)$ the instruction that is currently being decoded. The register that holds the current instruction word IW is part of the microarchitectural state μ . The arrow “ $\rightarrow$ ” denotes an instruction execution step. An instruction commits when the identifier of the currently committing instruction, c_{id} , matches the instruction’s identifier i_n . In its premise (top part), the SIF rules define the instruction’s source state (A, μ) and taint state (A_t, μ_t) that are specified to be read by the instruction. For example, in [Figure 4](#), the sources are the registers $R[\text{rs1}], R[\text{rs2}] \in A$. The SIF rule’s premise matches before instruction execution when the instruction being decoded matches the rule’s instruction type. State elements in A and μ and taint state elements in A_t and μ_t that are not defined in the premise can be of arbitrary value. A rule’s conclusion (bottom part) matches at instruction commit time and defines how taint propagates in dependence of the specified sources and design states seen by the instruction before execution. A' refers to the updated architectural state according to the instruction’s functional semantics [67]. We refer to specific elements within GPRs (R), CSRs (CSR) and memory (M), by using an address as index. For example $R[\text{rs1}]$ refers to the data stored in GPR rs1 , and $R_t[\text{rs1}]$ refers to the corresponding taint bit vector. In [Figure 4](#), taint propagates to the destination register (rd), according to the rule $C_{instr}^t(\text{rs1}, R_t[\text{rs1}], \text{rs2}, R_t[\text{rs2}])$, latest when i_n commits. C_{instr}^t is the bit precise taint propagation rule that matches the logic function of the instruction, based on IFT mechanisms [39], [41], [68], listed in [Table 1](#) and discussed in [Section 4.2.1](#). The notation t represents a taint value of same bit width as the taint register that it is assigned to, e.g., 32 bits for a particular R_t of a 32-bit RISC-V CPU. All other not explicitly mentioned architectural taint states remain *unaltered* after the execution step on the SIF model. In particular, taint is not systematically cleared. To define a SIF model, one such rule must be defined for each instruction that is specified either in the ISA or in a CPU-specific specification.

SIF semantics by example. Consider [Listing 1](#), with register values initialized to some arbitrary values. The listing performs some arithmetic operations and a CSR read and write operation (i4). There exists a SIF, e.g., from registers x1 and x2 to register x3 (i0), and from x3 to x7 (i1). Thus, there exists a SIF from x1 and x2 to x7 (via x3). Data stored in registers can persist over multiple execution steps. For example, i2 operates on x3 , which was computed by i0. This causes a SIF from x1 and x2 to x9 .

Listing 1: RISC-V toy example to explain the IFI definition. The **orange** registers depend on x1 read by i0. The **red** registers depend on x4 . Initial register values are independent.

```
i0: add x3, x1, x2      # x3 = x1 + x2
i1: sub x7, x6, x3      # x7 = x6 - x3
i2: or x9, x10, x3      # x9 = x10 or x3
i3: add x7, x8, x4      # x7 = x8 + x4
i4: csrrw x4, csr1, x7  # x4 = csr1, csr1 = x7
```

Listing 2: Definition of mul_rs1_t used in [Table 1](#).

```
not_tainted_lsb = R_t[rs1] - 1;
not_tainted_lsb_reversed =
    reverse_bit_order(not_tainted_lsb);
mul_rs1_t =
    (~not_tainted_lsb_reversed & R[rs2]);
```

Violation 1: If, in an incorrectly implemented CPU, i0 writes its (potentially sensitive) input data in x1 into a CSR csr1 and i4 reads from csr1 as specified, then there exists an IF from x1 to x4 , that does not belong to the SIFs. This violation resembles the vulnerability CVE-2023-51973 found in the Kronos CPU [25].

Violation 2: If this snippet is executed on a CPU and there exists an IF from x6 (read by i1), through x7 , to CSR csr1 , IFI is violated, because i3 was supposed to overwrite x7 with data independent of x6 .

4.2.1. Bit-accurate taint propagation rules

In [Table 1](#) we define bit-accurate rules that specify how taint (representing SIFs) propagates in the SIF model based on an instruction’s operation. The rules assume that only one bit of one instruction source is tainted at a time, while for untainted bits all possible combinations of values are verified. Thus, the rules need to be checked for each bit. Our implementation combines this check in proofs per instruction and source. The rules are based on known taint tracking rules by GLIFT [39], RTLIFT [70] and CellIFT [41]. New rules are marked with VerIFI in the Origin column. We define mul_rs1_t as shown in [Listing 2](#). mul_rs2_t is defined analogously by swapping rs1 and rs2 .

Influence of untainted bits. Taint propagation not only depends on the operations applied to the tainted states, but also on the values of the non-tainted signals that interact with them [39], [41]. For example, C_{or}^t in [Table 1](#) expresses that if the 5 lower bits of x3 in the or instruction (i2) in [Listing 1](#) are tainted, and x10 is not tainted, taint propagates to rd (x9) only if some of the 5 lower bits of the untainted operand x10 have value zero. It does not propagate if all 5 lower bits of x10 have value one.

Customizing SIF rules. [Figure 5](#) shows the SIF for the bne instruction in RISC-V without the extension for compressed instructions. Branches and jumps can cause a misaligned exception, which redirects the control flow to an exception handler. Hence, they can propagate

TABLE 1: Instruction-specific IFI taint propagation rules.

Instruction	Source register	Taint propagation rule	Origin
$C_{add,sub,xor}^t$	$R_t[rs1]/R_t[rs2]$	$R_t[rs1] \vee R_t[rs2]$	[41]
C_{and}^t	$R_t[rs1]$	$(R_t[rs1] \& R[rs2])$	[39], [41]
C_{and}^t	$R_t[rs2]$	$(R_t[rs2] \& R[rs1])$	[39], [41]
C_{or}^t	$R_t[rs1]$	$(R_t[rs1] \& \sim R[rs2])$	[39], [41]
C_{or}^t	$R_t[rs2]$	$(R_t[rs2] \& \sim R[rs1])$	[39], [41]
C_{eq}^t	$R_t[rs1]$	$!(\sim R_t[rs1] \& (R[rs1] \oplus R[rs2]))$	VerIFI
C_{eq}^t	$R_t[rs2]$	$!(\sim R_t[rs2] \& (R[rs1] \oplus R[rs2]))$	VerIFI
C_{mul}^t	$R_t[rs1]$	mul_rs1_t	VerIFI
C_{mul}^t	$R_t[rs2]$	mul_rs2_t	VerIFI
C_{sra}^t	$R_t[rs1]$	$(R_t[rs1] >> R[rs2][4 : 0]) \vee ((R_t[rs1][31] \& \& R[rs2])$	VerIFI

$$\begin{aligned}
 \text{BNE-NO-COMPRESSED} \quad & I(i_n) = \text{bne } rs1, rs2, label \quad t = C_{eq}^t(R[rs1], R_t[rs1], R[rs2], R_t[rs2]) \quad t_{pc} = t \vee PC_t \\
 & \langle R_t, CSR_t, M_t, PC_t, \mu_t, A, \mu \rangle \xrightarrow{\text{bne}} \langle R_t, CSR_t[\text{mtval} \mapsto t], CSR_t[\text{mepc} \mapsto t_{pc}], M_t, PC_t = t_{pc}, \mu_t[\text{misaligned_exception} \mapsto t], A', \mu[cid \mapsto i_n] \rangle \\
 \text{BEQ-W-COMPRESSED} \quad & I(i_n) = \text{beq } rs1, rs2, label \quad t = C_{eq}^t(rs1, R_t[rs1], rs2, R_t[rs2]) \vee PC_t \\
 & \langle R_t, CSR_t, M_t, PC_t, \mu_t, A, \mu \rangle \xrightarrow{\text{beq}} \langle R_t, CSR_t, M_t, PC_t = t, \mu_t, A', \mu[cid \mapsto i_n] \rangle
 \end{aligned}$$

Figure 5: Top: A ‘branch not equal’ (bne) instruction has a SIF to the PC, determined by C_{eq}^t , the taint propagation rule for the equality comparison (Table 1). For RISC-V CPUs that do not support the compressed instruction set extension, it also has an implicit SIF to a signal deciding about a misaligned exception and by that a timing flow to the PC, as well as to the CSRs. Bottom: Taint propagation rule for a ‘branch if equal’ (beq) instruction specified for a CPU that implements the compressed instruction set extension. No flow to the exception handling logic and CSRs is specified.

$$\begin{aligned}
 \text{MUL} \quad & I(i_n) = \text{mul } rd, rs1, rs2 \quad t = C_{mul}^t(R[rs1], R_t[rs1], R[rs2], R_t[rs2]) \quad t_{time} = \begin{cases} 0 & \text{if } CSR[DIT] = 1 \\ 1 & \text{otherwise} \end{cases} \\
 & \langle R_t, CSR_t, M_t, PC_t, \mu_t, A, \mu \rangle \xrightarrow{\text{mul}} \langle R_t[\text{rd} \mapsto t], CSR_t, M_t, PC_t = t_{time} \vee PC_t, \mu_t, A', \mu[cid \mapsto i_n] \rangle
 \end{aligned}$$

Figure 6: A SIF rule customized for the Ibex CPU [69]. A mul instruction taints the timing of the execution (PC_t) depending on the value of the Data-Independent Timing (DIT) mode CSR.

information from their operands to CSRs, the PC, and microarchitectural signals that indicate the existence of the exception, and thus affect the timing of the CPU pipeline. It further shows the rule for the beq instruction in RISC-V with the compressed instruction extension, where no exceptions are specified for branches.

Relaxing SIF rules. It could be desirable to relax the SIF rules, for example, to allow data-dependent execution timing in chosen configurations. Figure 6 shows how CPU-specific instruction behavior can be specified as SIF semantics. The Ibex CPU specifies a Data-Independent Timing (DIT) mode, in which the execution time of multiplication instructions (mul) must not depend on the mul’s operands. However, it also implements an optimized version where this rule is relaxed. The SIF rule correspondingly specifies a different taint value t_{time} that must be assigned to PC_t , depending on the DIT bit in the CSRs. We provide further examples in Section A.

4.3. IFI definition

We denote A_t^{EIF} as the taint states corresponding to architecturally defined state elements within the CPU implementation, corresponding to the architectural states A^{EIF} . We denote A_t^{SIF} as the architectural taint states in the SIF model, corresponding to its architectural states A^{SIF} . We define A_{t,i_n}^{EIF} as the taint state values after execution of instruction i_n on the CPU and A_{t,i_n}^{SIF} as the

taint state values in the SIF model after the execution step of i_n . We define microarchitectural (taint) states analogously, replacing the A with μ . The execution of an instruction i_n in the SIF model updates the taint states A_{t,i_n}^{SIF} and μ_{t,i_n}^{SIF} as a function of $A_{t,i_{n-1}}^{SIF}$, $A_{i_{n-1}}^{SIF}$, $\mu_{t,i_{n-1}}^{SIF}$ and $\mu_{i_{n-1}}^{SIF}$, according to the SIF semantics defined for $I(i_n)$. We define all initial taint values in the CPU $A_{t,-}^{EIF}$ and in the SIF model $A_{t,-}^{SIF}$ to be zero and let both models start in the same initial state (A_0, μ_0) .

IFI property. Definition 1 defines the IFI property. We present both the abstract SIF model and the CPU model with the same instruction sequence I . We allow the artificial injection of taint into some starting states A_{t,i_s}^{SIF} and A_{t,i_s}^{EIF} , where i_s is the instruction that completed last on the model, or into the initial state. Note before the taint injection, all taint states are zero in both models, due to the initial taints being zero. A CPU satisfies IFI, if and only if both CPU and abstract model start in the same initial configuration and for any arbitrarily introduced taint into some states A_{t,i_s}^{EIF} and A_{t,i_s}^{SIF} within any arbitrary sequence I executed on both models, the taint flow between architectural states in the CPU matches the one in the abstract model at all instruction commit times. $c_{id,clk}$ refers to a specific instruction identifier that indicates which instruction commits at CPU clock cycle clk .

Definition 1. Information-Flow Integrity (IFI):

$$\forall A_0, \forall \mu_0, \forall (i_n)_{n \geq 1}, \forall clk | c_{id,clk} = i_n : A_{t,i_n}^{EIF} = A_{t,i_n}^{SIF}$$

Some microarchitectural states may be altered by the instruction. If these IFs can be derived from the ISA, we state them in the instruction's IF semantics. Otherwise, they are unspecified. If these modifications influence the specified destination elements of any other instruction, an IFI violation will be detected as violation of that other instruction's SIF rule violation. Thus, this definition captures unspecified IFs that occur due to unexpected interactions of instructions.

5. Information-Flow Integrity Verification

We now describe our methodology for verifying IFI compliance. We first provide a high-level overview of the generic RTL properties leveraged to verify IFI in [Section 5.1](#). Then, we detail the concrete implementation in [Section 5.2](#).

5.1. Proving IFI compliance of a CPU

We first discuss taint injection and then outline the hardware properties that when all hold, prove IFI compliance of a CPU. These properties implement the SIF model by specifying the SIF rules and proving that the taint states corresponding to architecturally visible states of a CPU instrumented with taint logic [39], [41], [42], [70] (A_{t,i_s}^{CPU}) match the taint states of the architecturally visible states in the abstract model at all instruction commit times (A_{t,i_s}^{SIF}).

Initial architectural state. We start both the abstract SIF model as defined in [Section 4.2](#) and the taint-logic instrumented CPU RTL model in an arbitrary architectural state A_0 and all states matching between the two models. In the concrete implementation we start the CPU in the reset state and abstract memories and the GPRs. We set all taint states in both models to be zero initially.

Taint injection. As defined in [Section 4.2](#), our model allows the artificial introduction of taint into any architectural state A_{t,i_s}^{SIF} , where i_s is the instruction that completed last on both CPU and SIF model, or into the initial state. Similarly, we artificially inject taint into a CPU taint state A_{t,i_s}^{CPU} , at some point where all architectural states A_0 match between the CPU and the SIF model. IFI holds trivially before taint injection.

5.1.1. Hardware properties for IFI verification

Next, we discuss the properties that prove the equality of the CPU's taint states with the SIF model defined taints. A_{t,i_n}^{EIF} are a subset of the clock-cycle accurate hardware IFs in a CPU. We denote the latter as A_{t,i_s}^{CPU} and prove the properties expressed over these state. [Figure 7](#) provides intuition by showing example violations of each of these properties.

Containment. In the SIF model, information only propagates from one architectural state to another if an instruction is specified to cause that propagation. Thus, if in a CPU, an IF originates in a state in which *no* instruction is specified to read from that state, and that IF propagates to some architecturally visible state at any clock cycle, a difference between A_{t,i_s}^{CPU} and A_{t,i_s}^{SIF} exists. The *containment* property checks for such flows, exemplified in [Figure 7](#) (a). Information might propagate from states that are not supposed to be read by the decoded instruction ([b.2.2](#)), or when no specified instruction is decoded at all ([b.2.4](#)). For

example, b.2.4 matches CVE-2024-42999 found by VerIFI, which we will discuss in [Section 6](#).

Confinement. The SIF rules define the states whose taint changes with the execution of an instruction. In the SIF model, taint propagates from one architectural state to another via a (sequence of) instructions. For example, taint may propagate from some state A_{t,i_n}^{SIF} , to $A_{t,i_{n+1}}^{SIF}$ via the execution of i_{n+1} , and to some later state $A_{t,i_{n+m}}^{SIF}$ via the execution of i_{n+m} . However, in the SIF model, taint never propagates between instructions via other paths than through a chain of instruction destinations. Thus, if in a CPU, an IF originates in an instruction's source, and propagates to architecturally visible states at any clock cycle without passing through the instruction's destination at the instruction's commit time, a difference between A_{t,i_s}^{CPU} and A_{t,i_s}^{SIF} exists. The property catches all unspecified IFs to architecturally visible states to an instruction's destination at unspecified times ([b.2.1](#)), or to states not specified as destination ([b.2](#), [b.2.3](#)), both at any clock cycle over infinite sequences. For example, b.2 matches bug 3 found by ProcessorFuzz [27].

Freshness and Masking. The SIF rules bit-accurately define for any given combination of architectural states A_{i_s} , microarchitectural states μ_{i_s} , and corresponding taint states A_{t,i_s} and μ_{t,i_s} , how taint propagates from an instruction's sources to its destinations latest at the instruction's commit time. Thus, if an instruction is executed in a state that matches a rules premise and information propagates from sources to destinations differently than specified by the SIF rule, IFI is violated. We verify the equality of taint propagation at instruction's destinations, at commit time with the *freshness* and *masking* properties. [Figure 7](#) (c) illustrates *freshness* violations as dashed red arrows (missing information flows) between specified sources and destinations. The missing update of the instruction's destination leaks information from an older instruction, similar to Zenbleed [44] or Cascade's V12 [24]. [Figure 7](#) (d) illustrates a *masking* violation as red arrow between specified sources and specified sinks, showing an unspecified bitwise update.

Once information reached an architectural state as specified, it may either propagate further when another SIF rule matches, or it must remain contained. We verify the properties for any state in any possible sequence of states. We demonstrate how these properties verify IFI case by case in [Section B.1.1](#).

5.2. VerIFI implementation

We now discuss how VerIFI implements the properties described in [Section 5.1](#). We introduce two general mechanisms (taint path gating and Instruction Under Verification (IUV) tracking), shared by the properties. Then, we explain taint injection, and finally show how the properties are checked using SystemVerilog Assertions (SVAs) [71].

VerIFI takes as input a CPU RTL design augmented with a taint logic instrumentation, e.g. obtained by GLIFT [39], RTLIFT [70], CellIFT [41], or HybridIFT [42]. VerIFI is implemented in SystemVerilog using concurrent SVAs specified over CPU and taint signals. We prove them with an industrial hardware model checker, Cadence Jasper [50]. The properties require introduction and declassification of taint at chosen times and locations, for which we describe a novel path gating method. Further,

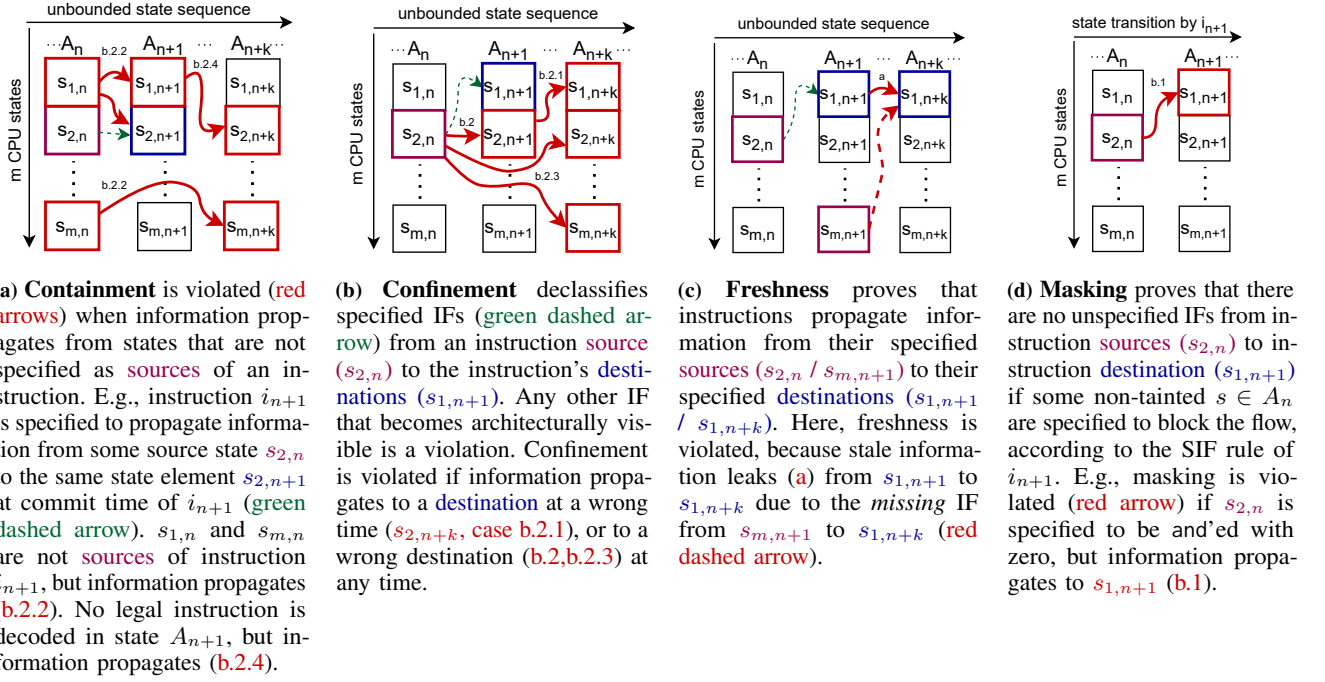


Figure 7: Hardware properties and their violation cases (a-b.2.4) discussed in detail in Section B.1.1.

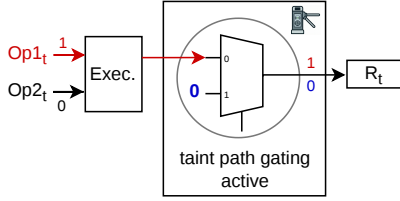


Figure 8: Taint path gating. $Op1_t$ is tainted, tainting the execution (Exec.) output. Taint path gating intercepts the taint logic driving a signal (e.g. the register taint signal R_t), and either sets it to a chosen value (here 0) if taint path gating is active, or connects it to its original driver.

they require the tracking of an IUUV through the pipeline, for which we implement a CPU-external IUUV tracker. The properties check containment and confinement in every clock cycle. For masking and freshness we specify CPU-specific check conditions, e.g., depending on the writeback condition or IUUV tracker stage.

5.2.1. General concepts

First, we discuss general concepts that we use to construct the properties.

Taint path gating. Figure 8 illustrates VerIFI's taint path gating method. We use logic abstraction to disconnect a taint signal (R_t) from its driver (output of Exec.) and make it an input. We then use SVA assumptions to either constrain the gated signal to a chosen value (zero in the figure) when path gating should be active, or to match its original driving logic otherwise. Because the disconnected (free) signal behaves like an input, the assumption does not affect the value of the original driving logic. This is in contrast to other methods in the literature [28], where the design's inputs and states are constrained to scenarios that force the declassified signal to a chosen value. Thus, taint path gating allows the verification of all possible

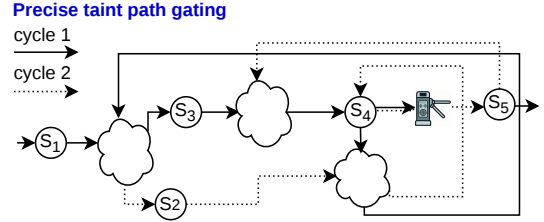


Figure 9: Path gating for declassification. Illustration of taint propagation over two clock cycles, where the taint path between states S_4 and S_5 is gated in clock cycle 1. For declassification, this represents a case where information is allowed to flow, and thus taint is blocked from propagating. In cycle 2 we allow taint to propagate in order to detect unexpected IFs to S_5 at other cycles than cycle 1.

input scenarios, even the ones that would originally taint the gated signal, while being able to ignore specified IFs via declassification. Figure 9 depicts the effect of path gating. Taint propagation from RTL signal S_4 is prevented precisely in clock cycle 1, due to some chosen path gating condition being true. In other cycles, taint propagates from that signal as the CPU's taint instrumentation defines.

IUV tracker. The CPU-specific Instruction Under Verification (IUUV) tracker assigns a one-bit token per pipeline stage, set when the IUUV occupies that stage. The tracker is implemented external to the CPU and uses only the handshake logic of the CPU to determine when instructions move between stages. Tracking starts with the first stage that receives tainted data, often the decode stage. When taint injection is active, the tracker sets the token of that stage, and clears it otherwise. If the pipeline stalls, it keeps the token. When the handshake signals pass an instruction to the next stage (e.g., the execution stage), the tracker passes on the token. If at the same time

no new instruction enters from the previous stage (e.g., from decode to execution), the tracker inserts a bubble by clearing the token in the stage that had an IUV visiting and has passed it on. For example, it clears the execution stage’s token if the execution completes (and passes on the token to the writeback stage) and at the same time no new (not tracked) instruction enters from decode. On a pipeline flush, the token is cleared automatically since no stage receives the IUV.

5.2.2. Taint injection

Freshness/Masking/Confinement. Freshness, masking and confinement share the same taint injection mechanism and can be verified together. We case split [25], [33], [34] the verification by proving each property per Instruction Under Verification (IUV) and source element, such as $R_t[rs1]$, $R_t[rs2]$. We prove the assertions in an isolated proof environment per instruction and taint source (implemented as ‘task’ in Cadence Jasper terms). We disconnect the chosen source (e.g. the taint signal of the signal reading from a GPR or CSR) from its driving logic and control its value with SVA assumptions similar to μ CFI [25]. However, in contrast to literature [25], [28], [31], which avoids all taint flows to GPRs, we aim to discover more leakage scenarios. Thus, we need to consider taint flows reaching the taint injection point. Taint path gating on the taint source can be used to constrain the taint source to be *tainted* when path gating is active, and to inject the taint value of the original driving logic otherwise.

Bit-wise case split for taint injection. When injecting taint, the question arises whether we should taint multiple bits at the same time. When checking for unspecified taints (confinement, masking), in case of a violation, it is irrelevant which bit caused the violation. This, however, differs when checking for the existence of taint (freshness). Let us consider an instruction with two sources. We could taint both sources at the same time, and prove that there always exists a taint flow from the sources to the destination. However, such a property would be satisfied even if only one of the specified sources is ever used. If, e.g., the instruction is used in software to clear a secret (e.g., and-ing it with zero), but the zero operand is not used by the hardware, the secret leaks. We constrain the taint source so that only one bit is tainted at a time, proving its individual effect. The model checker then considers all combinations of sequences in which exactly one source bit of one IUV is tainted, where any instruction in the sequence can be an IUV, and any source bit can be tainted. We prove in Section A.1 that the bit-wise case split is sufficient to verify the existence of taint (freshness), and rely on the proof in [25], Appendix A.1, which proves that a bit-wise case split is sufficient to detect absence of taint (confinement, masking). Leaving all top level design inputs unconstrained, we verify all possible combinations of instruction sequences, including illegal ones, and arbitrary external inputs.

Forwarding paths. We use taint path gating to declassify data passed on to younger instructions via forwarding paths. To ensure that we also verify scenarios where instructions receive forwarded data we extend the taint condition to include a potential forwarding condition, if the signal into which we inject taint corresponds to a signal that can read from both register data and forwarded data. To consider

forwarding to later stages, we replace the taint logic driving the signal that reads the instruction source data with an ‘or’ of the taint signals of the register data and forwarding data. Thus, if any of the two is tainted, its taint will propagate. While this method does not introduce false negatives, it may introduce false positives. However, we have not observed any in our experiments.

Containment. To verify containment, we case split taint injection on architectural state signals and inject taint when *no* instruction is supposed to read from them. We define the containment taint injection condition for a taint source as the negation of the disjunction of all taint injection conditions of instructions that are specified to access that taint source. This means taint injection is active whenever *no* instruction is specified to read that state element. We apply taint path gating for taint injection from the reset onwards, *tainting* the source as long as no instruction is specified to read from it and open the gate, i.e. restore the original taint path, whenever the containment taint injection condition is false. We reactivate the gating and taint injection as soon as there is again no instruction specified to read from that source. For example, we taint the memory output from the reset onwards and in all clock cycles in which there is no load instruction allowed to read memory data. If any architectural element becomes tainted, the information must have propagated from a point in time when no load was supposed to read from memory.

5.2.3. Property checking

Confinement and containment. *Confinement* checks that taint only reaches architectural states via the IUV’s specified destination at commit time. To determine the specified destination element, we save the destination GPR, CSR or memory address from the instruction word for which we injected taint. Then, we declassify the specified flows to destinations using path gating whenever the IUV tracker indicates that an instruction is in the stage in which the destination state element is written, and the saved address matches. Any further taint flows to architectural states at any clock cycle over infinite instruction sequences are excessive and violate IFI. *Containment* is violated if any taint injected reaches an architecturally visible state. Thus, we use the same assertion as for confinement, and no declassification.

Detecting architectural taint at the GPRs. The confinement property is shown in Listing 3.

Listing 3: Confinement/containment SVA.

```
assert property (!all_gpr_t0);
```

Listing 4: Freshness assertion in SystemVerilog.

```
freshness: assert property (
(
    freshness_masking_check_time &&
    expected_taint_at_declass_point_q
)
|> driver_of_declassified_sig != 0
);
```

`all_gpr_t0` is the concatenation of all GPR taint signals. A key observation can be made that an information leakage to any architecturally visible data-storing element can

eventually reach a GPR through one or more RISC-V assembly instructions. By the RISC-V specification there exists at least one instruction for each architectural state that transports its value to the GPRs. For example, an information leakage to the PC can be redirected to any GPR using an `auipc` instruction. Similarly, an information leakage to the CSR can be redirected to any GPR using a `csrrw` instruction. An information leakage to memory can be redirected to any GPR using an `lw` instruction [66]. Based on this observation, we can use a single property specified over the taint signals of the GPRs as taint sink for catching all invalid information flows to any architecturally visible state element. Since we are proving IFI for arbitrary and unbounded instruction sequences, any IFI violation manifesting as a taint flow to another architectural state, e.g., a CSR, can be detected by extending the instruction sequence by one more instruction that transports the taint to a GPR. Since we verify unbounded and arbitrary instruction sequences, the model checker is guaranteed to find such a sequence, if it exists.

Freshness and masking. We use the IUV tracker to sample the instruction’s sources before or when the IUV enters the execution stage. The untainted instruction sources determine whether taint must propagate (freshness) or must not propagate (masking), as specified in the SIF rules of the instruction. The `freshness_masking_check_time` signal is true when the declassification condition for confinement is active. The *Freshness* property is shown in Listing 4. The signal `driver_of_declassified_sig` is the signal that drives the declassified (and thus disconnected, i.e., logic abstracted) signal in the taint logic instrumented design. E.g., when we declassify the taint signal corresponding to the signal that writes back data to GPRs, `driver_of_declassified_sig` corresponds to the taint logic driving the write back data’s taint signal. `expected_taint_at_declass_point_q` is calculated based on the sampled instruction source signals, and the taint rules in Table 1. *Masking* is implemented analogously, but with `expected_taint_at_declass_point_q` negated and the inequality being an equality.

Replacing taint rules with a miter circuit. For instructions that are not mentioned in Table 1, we calculate specification-derived taint propagation with a miter circuit [51], [60], [68], [72] over logic that models the instruction’s operation, using the instruction sources as inputs. Thus, we duplicate this logic and track information as difference between the two copies rather than using taint rules, which is at least as precise as taint logic, and often more precise. The miter’s inputs are free signals [73], which we constrain with assumptions to match the instruction’s source register taint and value as follows: If a taint bit of a source is set, we constrain the two corresponding miter input bits to have arbitrary, but different values. If a source taint bit is zero, we constrain the two corresponding miter input bits to match the CPU’s corresponding instruction source value bit, as their concrete values affect the freshness and masking conditions. The miter outputs differ whenever taint rules would indicate a taint flow and are equal otherwise. We then use this result instead of taint rules to determine whether freshness or masking should hold. While easier to implement, and spared from potential false positives of taint logic [39], [41], [68], this method

TABLE 2: Lines of code of the evaluated CPUs (CellIFT-instrumented) and verification code (common and CPU-specific).

Design	CellIFT-ed design	Verification code	
		SystemVerilog	Tcl
Common	—	4853	575
Ibex	36 006	1890	117
Kronos	14 040	1572	162

has higher runtime compared to using taint rules, as we show in Section 6.

Replacing path gating for taint injection. Above, we suggested taint path gating for taint injection for freshness, masking and confinement to account for unexpected IFs to the taint source. In this case the taint source’s taint value depends on the whole CPU pipeline state when taint injection is inactive, complicating proofs. Due to our precise spatio-temporal declassification, any IF reaching the taint source constitutes an IFI violation. This insight allows us to replace path gating for taint injection with an additional property, which checks that taint never flows back to the taint injection source. Then, we can constrain the taint source to zero when taint injection is inactive (instead of reconnecting it to its original taint logic), which leads to fewer verification scenarios per proof and thus reduces the proof time per instruction.

6. Evaluation

We evaluate VerIFI on two open-source, microcontroller-class RISC-V CPUs (Kronos and Ibex). After describing setup and tools, we present runtime performance (Section 6.2), IFI property checking results (Section 6.3), and the discovered vulnerabilities (Section 6.4).

Evaluation setting. We chose CellIFT [41] for instrumenting the CPUs with taint logic, due to its state-of-the-art scalability and its open-source availability. As a model checker we chose Cadence Jasper Gold, version 2025.03. Our experiments were executed on a server equipped with an AMD EPYC 7H12 CPU running at 2.6 GHz and 480 GB of RAM, using 70 CPU threads. For debugging we used `rvcoded.js` [74] and `Pathfinder` [68].

6.1. Complexity

Next, we discuss the code complexity and manual effort.

Design size. Table 2 shows the size of the design and verification code. While the CellIFT-instrumented Ibex RTL code is $2.5\times$ larger than Kronos’s, the Ibex-specific verification code is only $1.2\times$ larger.

Manual effort. We estimate manual effort to be comparable to [28] and [60], while capturing more leakage. We defined ISA-specific rules once in 1 day. Many are reusable across ISAs. The following are CPU-specific efforts: Implementing the IUV tracker (~ 2 days) requires design knowledge, but no DUT modifications. Signals are usually easy to identify (~ 1 hour). We check committed architectural states of ISA-defined elements. While the IUV tracker eases the definition of declassification conditions, consideration of special cases (like error handling) can require debugging. Typical for verification [75], most time is spent debugging. Implementation and debugging (as non-designers) took several weeks per CPU. Property verification is automated [50] and requires no manual invariant search like [28], [34].

TABLE 3: Average runtime per CPU and proven property. M = multiplication, D = division. ○ means time out after 24 hours. ‘-’ means not applicable.

CPU	Freshness	Masking	Confinement	Containment
Kronos	< 30s	<15m	<10m	<10m
Ibex	<30s	~2.75h	~2h	~2h
Ibex (M)	~11h	~11h	~5h	-
Ibex (D)	○	○	~5h	-

TABLE 4: Proof results of Freshness (F), Masking (M) and Confinement (C) for **Ibex**, small. ✓ means proven, ✗ means violated, ○ means time out. If different, we show results **without** / **with** data-independent timing flag.

Instruction	Source	F	M	C
ADD, ADDI	RS1/2	✓	✓	✓
AND, ANDI	RS1/2	✓	✓	✓
OR, ORI	RS1/2	✓	✓	✓
XOR, XORI	RS1/2	✓	✓	✓
SRA, SLL	RS1	✓	✓	✓
SUB	RS1/2	✓	✓	✓
SLT, SLTI, SLTU	RS1/2	✓	✗	✓
DIV	RS1	○	✗	✓
DIV	RS2	○	○	✗/✓
MUL	RS1	✓	✓	✓
MUL	RS2	✗/✓	✗	✗/✓
BEQ, BNE	RS1/2	✓	✗	✗/✓
CSR	RS1	✓	✓	✓

6.2. Performance

Table 3 compares the runtime of the different properties. Freshness is the only property that checks for the *existence* of taint, and proofs complete in seconds on both CPUs for most instructions. The other properties check for taint *absence*, which takes longer: about 10 minutes for Kronos, and about 2 to 5 hours for Ibex. Freshness and masking for division instructions time out after 24 hours. These properties depend on the taint values, the division miter and the division logic, which is a common source of complexity in model checking. However, we are able to verify confinement, which covers most of the potential unspecified IFs between instructions, in about 5 hours. Checking C_{instr}^t of **Table 1** rules can be implemented with a miter, which increases the runtime for masking properties to more than 5 hours for some instructions (e.g., for shifts).

Initially, the proof time of confinement and masking for Ibex took about 75 minutes longer. We then added a new case split. Instead of path gating for taint injection, we added a new property that proves that no taint flows back to the taint injection source, leading to the proof times that are shown in **Table 3**. This additional property takes about 2 hours to be proven. Thus, only when sufficient hardware resources are available for parallel verification, the case split is beneficial to the overall runtime.

In general, proof times heavily depend on the order of the proofs, the combinations of properties per prove command and the available hardware resources.

6.3. Property verification results

VerIFI proves *freshness*, *masking* and *confinement* per instruction, and *containment* per architectural state. We first present the verification results for the two CPUs, and then discuss the discovered vulnerabilities in Section 6.4.

Ibex. **Table 4** shows verification results for *freshness*, *masking* and *confinement* on Ibex in the small configuration

TABLE 5: Proof results of Freshness (F), Masking (M) and Confinement (C) for **Kronos** with constrained bus interface. ✓ means proven, ✗ means violated.

Instruction	Source	F	M	C
ADD, ADDI	RS1/2	✓	✓	✓
AND, ANDI	RS1/2	✓	✓	✓
OR, ORI	RS1/2	✓	✓	✓
XOR, XORI	RS1/2	✓	✓	✓
SRA, SLL	RS1/2	✓	✓	✓
SUB	RS1/2	✓	✓	✓
SLT, SLTI, SLTU	RS1/2	✓	✗	✓
BEQ, BNE	RS1/2	✓	✓	✓
BLTU	RS1/2	✓	✓	✓

TABLE 6: VerIFI results for *containment*. Kronos behaves differently with bus interface constrained to be error free vs. when allowing errors. Ibex small contains the bug fix of CVE-2024-28365, while Ibex custom (incl. writeback stage) does not.

Source	Kronos	Kronos bus errors	Ibex small	Ibex custom
mem_data	✓	✗	✗	✗
reg_data_op1	✓	✗	✓	✗
reg_data_op2	✓	✗	✓	✗

with unconstrained bus interface. On Ibex, IFI holds for most arithmetic instructions. Several masking properties fail, but our analysis indicates these are false positives from CellIIFT’s taint tracking. For example, the reason for the failure for branches is that Ibex, like many CPUs, implements equality checks for branches as a subtraction followed by a zero check of the result. This construct produces false positives in CellIIFT, as well as in the other existing information flow tracking mechanisms [40]–[42], because addition instrumentation propagates more taint than the instrumentation of the comparison operator would. To demonstrate that access pattern leakage is in scope, we modelled cached memory accesses. Taint flows from load addresses, via the cache access logic, to the PC/GPRs, violating *confinement*.

SIF rule customization. Multiplication and division instructions exhibit timing side channels in the default runtime configuration as previously shown [25], [28], [29], [51]. The SIF rules can be customized as we have shown in **Figure 6**. When the DIT flag in an Ibex-specific CSR is set before the execution of the instruction, no timing side channel is expected by the specification. To implement such a customization, we include the DIT flag into the taint injection condition and encounter no further timing side-channel counterexamples.

Kronos. **Table 5** shows verification results for *freshness*, *masking* and *confinement* on Kronos. We first constrain the bus interface such that every (data or instruction) request of the CPU is followed by a response exactly one clock cycle later, which limits the verified scenarios. In this configuration, most properties under verification pass. Masking of ‘set less than’ instructions (SLT(I/U)) fails, like for Ibex, due to taint tracking false positives. When leaving the CPU interfaces unconstrained, we find the vulnerabilities discussed in **Section 6.4**.

Containment. **Table 6** presents the results for the *containment* property, which we prove per architectural state. Again, all properties fail for Kronos with unconstrained bus interface. On Ibex (μCFI [25] custom configuration), we

rediscover a vulnerability. On Ibex (small configuration), we find a new vulnerability discussed below.

6.4. Discovered vulnerabilities

Validating the verification. μ CFI reports a register leakage (CVE-2024-28365) that was found when manually analyzing a violation. VerIFI should find such leakage automatically. Thus, to validate our verification method, we further verified the Ibex configuration used in [25] where the leakage was found. This required us to update the IUV tracker to consider the inclusion of a write-back stage. We successfully re-discovered the leakage both with confinement and containment. The latter fails because the intermediate result of multiplications persists after Ibex' improper response to unexpected memory errors. Thus, the result can be exfiltrated later by instructions that were not expected to read the multiplication's source GPR. When including the bug fix to the CVE, the properties pass.

6.5. New vulnerabilities

We now discuss the new vulnerabilities discovered.

Kronos: Leakage of memory data - new CVE-2024-42999. When supplied with a memory acknowledgement signal that was not preceded with a load or store request, Kronos wrongly decodes an invalid instruction as a load. The register with an address corresponding to the invalid instruction's `rs1` field will be treated as the operand data and used to calculate the load address. Consequently, an unexpected load will occur, causing a value from memory to be falsely loaded into the GPR whose address is equal to the invalid instruction's `rd` bits. An attacker capable of injecting spurious acknowledgements (e.g., via fault injection, realistic for microcontrollers [76]) can force arbitrary memory values into GPRs, enabling memory leaks. Similarly, the attack vector could be used to intentionally corrupt a GPR. We reported this vulnerability at [77], [78].

Kronos: Information leakage of GPRs. An unexpected data acknowledgement signal on the bus causes the load store unit's 'ready' signal to rise without corresponding load or store request. This causes the CSR write-enable signal to be spuriously activated during any arithmetic or logic instruction following a CSR instruction. The data in the [31:20] field of the arithmetic or logic instruction is misinterpreted as the CSR address, and the data of the source register of one of the arithmetic or logic instructions will be mistreated as the CSR write data. As a result, the GPR[rs1] can flow to any other GPR by a subsequent CSR read instruction. This vulnerability might have a root cause related to CVE-2023-51973 [25], but our results show that the vulnerability has broader consequences than previously known.

Furthermore, with unconstrained bus interface, we obtain property violations where instructions read from the source registers of a younger instruction. This is similar to CVE-2024-44927, which enabled a control-flow hijack in response to unsolicited bus requests. We discover that the underlying vulnerability also leads to register data leakage.

Ibex: Leakage of memory data. In Ibex, VerIFI flagged a violation of the containment property when memory data changes, and the corresponding error correcting code (ECC) bits sent with the data are not updated accordingly. Ibex triggers an unexpected exception. According to the specification, the ECC bits shall only be checked by Ibex

when a load or store request is happening. Thus, it is valid to integrate Ibex into a system-on-chip, connecting it to a shared data bus on which other IP cores could communicate while Ibex is not interacting with the memory. Due to Ibex' vulnerability flagged by VerIFI, it is possible to deduce information about the data on the bus by observing the existence of the exception. Alternatively, a malicious IP could intentionally trigger the exception by changing the data on the bus, e.g., attacking Ibex' availability. The vulnerability has been acknowledged by the Ibex maintainers. No CVE was issued, due to being only an availability concern [79].

7. Discussion

Imprecision. Hardware taint tracking mechanisms (e.g., CellIFT [41]) can have false positives (but no false negatives), which we note in the Section 6.

Declassification according to the SIF rules is a manual process. Missing to declassify an IF can lead to false positives. Declassifying too many IFs can cause false negatives. The risk of choosing a wrong signal or condition can be reduced when design engineers provide the declassification conditions as part of the specification [49]. An incorrect tracker implementation could cause false negatives. However, since freshness checking and confinement declassification are triggered by the same signal, human mistakes in the tracker implementation are easily found by unexpected violations of one or the other.

Reliance on design signals. We rely on the correct functionality of the design's handshake signals when implementing the IUV tracker. These often implement a simple ready/valid protocol, which can be verified by targeted functional verification properties. Note that detailed conditions like, e.g., flush conditions, are not required to implement the tracker.

Extensions. The SIF model specification method as well as VerIFI can be adapted for other ISAs analogously to our definition for RISC-V. Our SIF semantics define information propagation in program order, but an out-of-order (OoO) CPU implementation must comply with these rules as well. Specifying declassification conditions for OoO CPUs requires additional considerations regarding the commit condition of instructions. Contrary, we expect the IUV tracker to be simpler, because OoO CPUs already track instructions. Obtaining unbounded IFT proofs on large CPUs remains an open challenge [80].

Other functional bugs. IFI does not detect functional bugs that do not affect the information flow. For example, inverting bit-wise all the bits of the output of the addition, whatever the input, would not be detected by IFI verification, because bitwise inversion does not affect taint propagation. When a subsequent instruction uses the wrongly calculated result, some security check might be bypassed and information may propagate unexpectedly in software. For example, VerIFI cannot detect an integrity violation where an instruction always overwrites a GPR different from its destination with constant 0, as this is an operand data-independent functional bug.

8. Related Work

8.1. Overview

Table 7 compares IFI with related work. Most cover either functional or CT verification, while IFI captures both.

TABLE 7: We compare coverage of Functional Verification (FuV) and constant time (CT), as well as whether interactions between instructions are considered, and how much of the ISA is covered by the implementation.

	FuV	CT	IFI	Instruction interaction	CF instrs.
VerIFI (this paper)	●	●	●	●	●
Verifying x86 [33]	●	○	●	○	●
ARM formal [32]	●	○	●	○	●
CHERIOT Ibex [34]	●	○	●	●	●
μCFI [25]	○	○	●	●	●
H-Houdini [30], UPEC [60]	○	●	○	○	○
LeaVe [28], Shadow [49]	○	●	●	●	○
RTL2MμPATH [31]	○	●	●	●	○

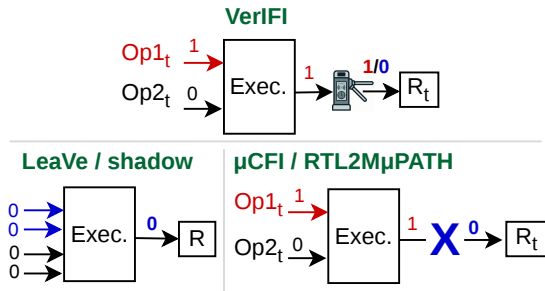


Figure 10: Handling architectural taint flows. VerIFI is able to precisely gate IFs under chosen conditions. μCFI [25] and RTL2MμPATH [31] instead block flows to the GPRs at all times. LeaVe [28] and contract shadow logic [49] constrain chosen destinations to be equal in both instances, which back-propagates this constraint to instruction inputs.

CT verification schemes often assume functional correctness [28], [49], which is hard to achieve in practice [32], [34]. IFI verification does not rely on such assumptions. LeaVe [28] and contract shadow logic [49] exclude all IFs originating in control flow (CF) instructions with their input constraints. Since they assume functional correctness, they cannot detect unspecified data flows, e.g., from a branch source to some GPR. Further, they cannot detect timing variances caused by CF instructions, similar as μCFI [25]. Due to imprecise declassification, μCFI cannot detect unspecified IFs to GPRs that do not affect the PC before reaching a GPR [25]. H-Houdini [30] and UPEC [60] exclude CF instructions, covering less interactions between instructions, and do not verify branches at all. The x86 verification in [33] does not consider interactions between instructions. Recently, several security contract extensions for ISAs have been proposed [81]–[83]. IFI does not focus on verifying a single contract with a chosen leakage channel. Instead, it provides a generic verification method for proving any customized information leakage or integrity contract for ISAs. MileSan [23] detects IFI violations through fuzzing, even on larger CPUs. However, it cannot prove their absence.

8.2. Taint injection and declassification

Declassification. Figure 10 compares various methods for handling of architectural IFs. LeaVe [28] and contract shadow logic [49] use self composition, where taint is modelled as difference between two instances of a design [68]. To avoid certain taint flows, LeaVe [28] proposes to use

assumptions, which ensure that instruction sources that may affect a declassified destination are never different between the two copies (i.e., never tainted). As a result, only input sequences that do not cause an IF to the declassified signal can be verified. Using this method, LeaVe [28] and contract shadow logic [49] avoid tainting inputs to branch instructions, which is not of interest for CT verification. μCFI [25] and RTL2MμPATH [31] declassify all IFs through architectural paths between instructions at all times, because they focus on taint flows to the PC (or commit signal in case of RTL2MμPATH) via microarchitectural paths only. Both methods are too restrictive for verifying IFI. To capture more leakage, we need to be able to taint all instruction sources, while declassifying only specific, chosen IFs originating in them. Path gating solves this problem.

Taint flowing back to the taint source. CT verification methods [25], [28], [31] typically declassify all taint flows to GPRs, and can thus ignore taint that flows back to the taint source during verification. They further ignore implicit taint flows back to the source via other paths than the GPRs, likely assuming that these would also taint control signals within the CPU and eventually surface as CT property violation. IFI considers these paths as it aims to detect unspecified functional flows as well.

Hardware verification. Ibex with CHERIOT capabilities has been recently functionally verified against the corresponding Sail model [34]. The full verification required the manual identification of a large set of helper invariants for proof strengthening. The mainline Ibex could be proven against Sail with the Cadence Jasper Gold model checker for all instructions, except for the M-type ones (multiplications and division), which did not conclude [84]. VerIFI was able to prove confinement for M-type instructions. VeriCHERI uses the general security property of non-interference between memory areas belonging to different tasks as a base for verifying the security of CHERI capabilities [63]. Sun et al. [85] extend taint tracking with a second instrumentation logic that extracts structural information flow paths, e.g., tracking whether an IF propagated through a specific module. This method is an interesting alternative to declassification if a fine-grained microarchitectural specification is available. It requires knowing through which state elements information must propagate. PROSPECT is a formalization for a secure speculation mechanism that protects constant-time programs against speculatively created timing side channels. PROSPECT is implemented for a RISC-V CPU, but the implementation is not formally verified. To verify PROSPECT against IFI, customized SIFs could be written that are less precise (as assumed by PROSPECT), and account for accepted leakage during declassification, which PROSPECT implements as an explicit store of secret data to public memory [86]. SecVerilog [87] defines IFT semantics for the Verilog language, based on static labeling of Verilog variables. Hardware implemented in SecVerilog [88], [89] or with dedicated hardware monitors [90] or static partitioning [89], [91] provide formal security guarantees for target software and usually come with area or performance overheads. IFI instead defines generic per-instruction IFT semantics and taint propagation rules, enabling verification of any CPUs written in standard languages, without a need for security

labels. However, it needs to be combined with verified software to extend guarantees beyond the hardware level.

IF ISA extensions. Several hardware/software co-verification methods specify, often conservative, information propagation rules for, mostly, data flow instructions and use them together with hardware verification based on, e.g., GLIFT [39] to verify security properties of software executed on a specific hardware [92]–[94]. Several software IFT schemes define IF rules for ISAs. Most overapproximate taint propagation, since they do not consider bit-precise taint propagation like VerIFI. For example, label propagation for logic instructions is often chosen as ‘or’ of both operands [95]–[97]. [93] defines conservative IFT rules for data flow operations, as well as precise gate level taint tracking, similar to GLIFT [39] for a few gates. FineDIFT [97] implements a co-processor that propagates a taint label between memory and registers dynamically, assuming that the main CPU propagates information in that way, as expected by the ISA. All these methods assume that a CPU follows their defined IF propagation rules. VerIFI is the first verification method for proving these assumptions on an actual CPU design. Information flow tracking ISAs exist, which explicitly add label propagation to the ISA [95], [98]. These methods can control IFs at runtime. Compliance verification of a CPU with such an ISA remains an open topic. Sailor [99] determines an ISA footprint to ensure secure context switching in software. Their toolchain could potentially be used to automatically generate SIF rules.

Software security. Many information flow policies that restrict in which way information may be modified have been defined for software security [100], [101]. They are usually tailored to software-level concepts and take secure information propagation within the hardware for granted.

9. Conclusion

We presented Information-Flow Integrity (IFI), a generic CPU property that augments an existing ISA with information flow semantics, and enables the detection of vulnerabilities that compromise the confidentiality and integrity of architecturally-observable information inside the CPU. IFI is defined as operational semantics for instructions with respect to taint propagation. IFI semantics are customizable, allowing the explicit specification of taint paths that are derivable from particular microarchitectural features (e.g., data-independent timing) or known bugs. We further introduce VerIFI, a verification method for verifying IFI compliance via four RTL-level Information Flow Tracking (IFT) properties: *freshness*, *masking*, *confinement* and *containment*. Making use of our novel spatio-temporal taint path gating method, we are able to precisely declassify specified information flows, enabling the discovery of further unknown paths that lead to vulnerabilities. We have applied VerIFI to two open-source RISC-V CPUs, Kronos and Ibex. We could formally prove compliance for multiple instructions, while also finding two new vulnerabilities despite the fact that both these CPUs had already undergone formal verification.

Acknowledgments

We thank the reviewers for their valuable feedback. This work has been supported by a Qualcomm Innovation Fellowship and the Swiss State Secretariat for Education, Research and Innovation under contract number

MB22.00057 (ERC-StG PROMISE). The turnstile icon used in Figures 8-10 was generated with Gemini Pro.

References

- [1] 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, “Spectre attacks: Exploiting speculative execution,” in *IEEE S&P ’19*. IEEE, 2019.
- [2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *USENIX Security ’18*.
- [3] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, “SmotherSpectre: exploiting speculative execution through port contention,” in *ACM SIGSAC*, 2019.
- [4] J. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Fore-shadow: Extracting the keys to the Intel sgx kingdom with transient out-of-order execution,” in *USENIX SEC*, 2018.
- [5] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, “Fallout: Leaking data on meltdown-resistant cpus,” in *ACM SIGSAC*, 2019.
- [6] “CVE-2018-3639.” Available from MITRE, CVE-ID CVE-2018-3639, 2018.
- [7] G. Maisuradze and C. Rossow, “ret2spec: Speculative execution using return stack buffers,” in *ACM SIGSAC*, 2018.
- [8] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “ZombieLoad: Cross-privilege-boundary data sampling,” in *ACM SIGSAC ’19*, ser. Ccs ’19. Association for Computing Machinery, pp. 753–768.
- [9] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, “Netspectre: Read arbitrary memory over network,” in *ESORICS*, 2019.
- [10] J. Stecklina and T. Prescher, “Lazyfp: Leaking fpu register state using microarchitectural side-channels,” *arXiv preprint arXiv:1806.07480*, 2018.
- [11] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “Ridl: Rogue in-flight data load,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 88–105.
- [12] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, “Crosstalk: Speculative data leaks across cores are real,” in *IEEE S&P ’21*. IEEE, 2021.
- [13] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, “Lvi: Hijacking transient execution through microarchitectural load value injection,” in *IEEE SP*, 2020.
- [14] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, “Cacheout: Leaking data on Intel cpus via cache evictions,” in *IEEE SP*, 2021.
- [15] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, “Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks,” in *USENIX SEC*, 2021.
- [16] J. Wikner and K. Razavi, “RETBLEED: Arbitrary Speculative Code Execution with Return Instructions,” in *USENIX Security ’22*.
- [17] J. Wikner, D. Trujillo, and K. Razavi, “Phantom: Exploiting decoder-detectable mispredictions,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 49–61. [Online]. Available: <https://doi.org/10.1145/3613424.3614275>
- [18] J. Wikner and K. Razavi, “Breaking the barrier: Post-barrier spectre attacks,” in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 3516–3533.

- [19] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against {Cross-Privilege} spectre-v2 attacks," in *31st USENIX Security Symposium (USENIX Security '22)*, 2022, pp. 971–988.
- [20] D. Trujillo, J. Wikner, and K. Razavi, "INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution," in *USENIX Security '23*.
- [21] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector: {High-Precision} branch target injection attacks exploiting the indirect branch predictor," in *33rd USENIX Security Symposium (USENIX Security '24)*, 2024, pp. 2137–2154.
- [22] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based fault injection using selective state reset," in *33rd USENIX Security Symposium (USENIX Security '24)*, 2024.
- [23] T. Kovats, F. Solt, K. Ceesay-Seitz, and K. Razavi, "Milesan: Detecting exploitable microarchitectural leakage via differential hardware-software taint tracking," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025.
- [24] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU Fuzzing via Intricate Program Generation," in *USENIX Security '24*.
- [25] K. Ceesay-Seitz, F. Solt, and K. Razavi, " μ cfi: Formal verification of microarchitectural control-flow integrity," in *ACM CCS*, 2024. [Online]. Available: https://comsec-files.ethz.ch/papers/mucfi_ccs24.pdf
- [26] S. Rügge, J. Wikner, and K. Razavi, "Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions," in *USENIX Security*, 2025.
- [27] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "ProcessorFuzz: Processor fuzzing with control and status registers guidance," in *HOST '23*.
- [28] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2128–2142. [Online]. Available: <https://doi.org/10.1145/3576915.3623192>
- [29] S. Dinesh, M. Parthasarathy, and C. Fletcher, "CONJUNCT: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks," in *IEEE S&P '24*. IEEE Computer Society, pp. 176–176. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00180>
- [30] S. Dinesh, Y. Yhu, and C. Fletcher, "H-HOUDINI: Scalable invariant learning," in *ASPLOS 2025*.
- [31] Y. Hsiao, N. Nikoleris, A. Khyzha, D. P. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "Rtl2m μ path: Multi- μ path synthesis with applications to hardware security verification," in *MICRO*.
- [32] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with ISA-formal," in *CAV '16*, S. Chaudhuri and A. Farzan, Eds. Springer International Publishing, pp. 42–58.
- [33] S. Goel, A. Slobodova, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 47–60. [Online]. Available: <https://doi.org/10.1145/3372885.3373811>
- [34] L.-E. Ploix, A. Armstrong, T. Melham, R. Lin, H. Wang, and A. Courtney, "Comprehensive formal verification of observational correctness for the cheriot-ibex processor," 2025. [Online]. Available: <https://arxiv.org/abs/2502.04738>
- [35] F. Solt, P. Jattke, and K. Razavi, "RemembERR: Leveraging micro-processor errata for design testing and validation," in *IEEE/ACM MICRO '22*.
- [36] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *IEEE S&P '21*.
- [37] J. Hur, S. Song, S. Kim, and B. Lee, "Specdoctor: Differential fuzz testing to find transient execution vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1473–1487.
- [38] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "{MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *32nd USENIX Security Symposium (USENIX Security '23)*, 2023, pp. 1307–1324.
- [39] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, p. 109–120, Mar. 2009. [Online]. Available: <https://doi.org/10.1145/2528521.1508258>
- [40] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *IEEE DATE '17*. IEEE.
- [41] F. Solt, B. Gras, and K. Razavi, "CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL," in *USENIX Security '22*.
- [42] F. Solt and K. Razavi, "Hybridift: Scalable memory-aware dynamic information flow tracking for hardware," in *ICCAD (International Conference on Computer-Aided Design)*, 2024.
- [43] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *USENIX Security '23*.
- [44] T. Ormandi. Zenbleed. Accessed: 2025-10-24. [Online]. Available: <https://lock.cmpxchg8b.com/zenbleed.html>
- [45] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "{ÆPIC} leak: Architecturally leaking uninitialized data from the microarchitecture," in *31st USENIX Security Symposium (USENIX Security '22)*, 2022, pp. 3917–3934.
- [46] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *USENIX Security '18*.
- [47] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, vol. 17.
- [48] L. Gerlach, D. Weber, R. Zhang, and M. Schwarz, "A security RISC: Microarchitectural attacks on hardware RISC-v cpus," in *IEEE S&P '23*.
- [49] Q. Tan, Y. Yang, T. Bourgeat, S. Malik, and M. Yan, "RTL verification for secure speculation using contract shadow logic," in *ASPLOS 2025*.
- [50] Cadence jasper formal property verification (fpv) app. Accessed: 2025-10-24. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html
- [51] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "Towards a formally verified hardware root-of-trust for data-oblivious computing," in *ACM/IEEE DAC '22*, ser. Dac '22. Association for Computing Machinery, pp. 727–732. [Online]. Available: <https://doi.org/10.1145/3489517.3530981>
- [52] Ibex documentation - verification stages. Accessed: 2025-10-24. [Online]. Available: https://ibex-core.readthedocs.io/en/latest/03_reference/verification_stages.html
- [53] R. Kande, A. Crump, G. Persyn, P. Jauernig, A. R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities," in *USENIX Security '22*.
- [54] S. Goel, A. Slobodova, R. Sumners, and S. Swords, "Balancing automation and control for formal verification of microprocessors," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 26–45.
- [55] K. L. McMillan, "Interpolation and sat-based model checking," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2003, pp. 1–13.
- [56] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.

- [57] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner, "Gate-level information flow tracking for security lattices," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 1, pp. 1–25, 2014.
- [58] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware information flow tracking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.
- [59] M. Bölcskei, F. Solt, K. Ceesay-Seitz, and K. Razavi, "Encarsia: Evaluating cpu fuzzers via automatic bug injection," in *USENIX Security '25*.
- [60] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "A scalable formal verification methodology for data-oblivious hardware," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2024.
- [61] Kronos RISC-v (all cascade fixes integrated). Accessed: 2026-04-24. [Online]. Available: <https://github.com/cascade-artifacts-designs/cascade-kronos>
- [62] Boom issue 503. Accessed: 2026-04-24. [Online]. Available: <https://github.com/riscv-boom/riscv-boom/issues/503>
- [63] A. L. e. a. Duque Antón, "Vericheri: Exhaustive formal security verification of cheri at the rtl," in *ICCAD '24*.
- [64] Intel Corporation. Data Operand Independent Timing ISA Guidance. Accessed: 2025-11-10. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html>
- [65] A. Ltd. Dit, data independent timing. Accessed: 2025-11-10. [Online]. Available: <https://developer.arm.com/documentation/ddi0601/2022-06/AArch64-Registers/DIT--Data-Independent-Timing>
- [66] RISC-V International, *The RISC-V Instruction Set Manual Volume I, Unprivileged Architecture*, accessed: 2025-10-24. [Online]. Available: <https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications#ISA-Specifications>
- [67] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "Isa semantics for armv8-a, risc-v, and cheri-mips," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3290384>
- [68] K. Ceesay-Seitz, F. Solt, A. Klukas, and K. Razavi, "Pathfinder: Constructing cycle-accurate taint graphs for analyzing information flow traces," in *ICCAD 2025*, 2025.
- [69] Ibex CPU. Accessed: 2025-10-24. [Online]. Available: <https://github.com/lowRISC/openitan>
- [70] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.
- [71] "IEEE standard for SystemVerilog—unified hardware design, specification, and verification language."
- [72] M. R. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors," *IEEE Transactions on Computers*, vol. 72, no. 01, pp. 222–235, Jan. 2023. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/TC.2022.3152666>
- [73] E. Seligman, T. Schubert, and A. M. V. Kiran Kumar, *Formal Verification. An Essential Toolkit for Modern VLSI Design*, 2nd ed. Elsevier Science & Technology.
- [74] J. Porquet-Lupine, H. N. Sakai, and A. Sohal, "rvcodec.js: An educational converter for risc-v instructions," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2*, ser. SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1396. [Online]. Available: <https://doi.org/10.1145/3545947.3576340>
- [75] Siemens AG. The 2022 Wilson Research Group Functional Verification Study. Accessed: 2025-10-24. [Online]. Available: <https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/>
- [76] H. Zhao, V. Lapotre, and G. Gogniat, "Fault injection in on-chip interconnects: A comparative study of wishbone, axi-lite, and axi," *arXiv preprint arXiv:2509.24929*, 2025.
- [77] Kronos github: Information leakage vulnerability - issue 15. Accessed: 2026-04-24. [Online]. Available: <https://github.com/SonalPinto/kronos/issues/15>
- [78] Kronos github: Information leakage vulnerability - via memory - issue 16. Accessed: 2026-04-24. [Online]. Available: <https://github.com/SonalPinto/kronos/issues/16>
- [79] Lowrisc. Ibex issue 1970. Accessed: 2025-10-24. [Online]. Available: <https://github.com/lowRISC/ibex/issues/1970>
- [80] Y. Yang, Q. Tan, T. Bourgeat, S. Malik, and M. Yan, "Compass: Navigating the design space of taint schemes for rtl security verification," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 442–458. [Online]. Available: <https://doi.org/10.1145/3779212.3790144>
- [81] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *IEEE S&P '21*. IEEE.
- [82] J. Hofmann, E. Vannacci, C. Fournet, B. Köpf, and O. Oleksenko, "Speculation at fault: Modeling and testing microarchitectural leakage of {CPU} exceptions," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7143–7160.
- [83] H. Winderix, M. Bogner, L.-A. Daniel, and F. Piessens, "Libra: Architectural support for principled, secure and efficient balanced execution on high-end processors," in *CCS 2024*, 2024.
- [84] Accessed: 2025-10-24. [Online]. Available: <https://github.com/lowRISC/ibex/blob/92d4c7fea496e5588f1c51effdb22884477c19d9/dv/formal/README.md>
- [85] H. Sun, Z. Yang, X. Chen, H. Xu, and Z. Yuan, "Hardware information flow tracking based on lightweight path awareness," *Computers & Security*, vol. 147, p. 104072, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167404824003778>
- [86] L.-A. Daniel, M. Bogner, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "{ProSpeCT}: Provably secure speculation for the {Constant-Time} policy," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7161–7178.
- [87] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 503–516. [Online]. Available: <https://doi.org/10.1145/2694344.2694372>
- [88] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 555–568. [Online]. Available: <https://doi.org/10.1145/3037697.3037739>
- [89] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1583–1600. [Online]. Available: <https://doi.org/10.1145/3243734.3243743>
- [90] S. Lau, T. Bourgeat, C. Pit-Claudel, and A. Chlipala, "Specification and verification of strong timing isolation of hardware enclaves," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1121–1135. [Online]. Available: <https://doi.org/10.1145/3658644.3690203>

- [91] Z. Yao, S. M. Seyed Talebi, M. Chen, A. Amiri Sani, and T. Anderson, "Minimizing a smartphone's tcb for security-critical programs with exclusively-used, physically-isolated, statically-partitioned hardware," in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 233–246. [Online]. Available: <https://doi.org/10.1145/3581791.3596864>
- [92] G. S. Nicholas, D. V. Aklekar, B. Thakar, and F. Saqib, "Secure instruction and data-level information flow tracking model for risc-v," *Cryptography*, vol. 7, no. 4, 2023. [Online]. Available: <https://www.mdpi.com/2410-387X/7/4/58>
- [93] L. Wu, Y. Gao, J. Zhu, Y. Tai, and W. Hu, "Security verification of risc-v system based on isa level information flow tracking," in *2023 IEEE 32nd Asian Test Symposium (ATS)*, 2023, pp. 1–6.
- [94] K. Chen, Q. Deng, Y. Hou, Y. Jin, and X. Guo, "Hardware and software co-verification from security perspective," in *2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*, 2019, pp. 50–55.
- [95] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *NDSS 2019*, 2019.
- [96] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August, "Rifle: An architectural framework for user-centric information-flow security," in *MICRO*, 2004.
- [97] K. Chen, O. Arias, Q. Deng, D. Oliveira, X. Guo, and Y. Jin, "Finedift: Fine-grained dynamic information flow tracking for data-flow integrity using coprocessor," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 559–573, 2022.
- [98] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an isa that controls timing channels," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 272–27215.
- [99] N. S. Kalani, T. Bourgeat, G. D. H. Hunt, and W. Ozga, "Save what must be saved: Secure context switching with sailor," in *USENIX Security 2025*, 2025.
- [100] P. L. Y. Mao and S. Zdancewicz, "Information integrity policies," in *Workshop on Formal Aspects in Security & Trust (FAST)*, 2003.
- [101] U. Shankar, T. Jaeger, and R. Sailer, "Toward automated information-flow integrity verification for security-critical applications," in *NDSS*, 2006.

Appendix A.

Additional IFI Specification Examples

Figure 11 depicts additional SIF rules for the RISC-V ISA.

A.1. Bit-by-bit taint tracking

We show that tainting one bit at a time and unconstraining the other input bits covers information flows entirely. We consider a combinational cell C (e.g., an and gate or an adder) with input vector I of length n , and input taint vector I^t of same length. We suppose, without loss of generality, that the cell has a single output bit y (if it has multiple output bits, we can consider each output bit as the result of a distinct combinational cell). We assume that there exists an input vector I and a corresponding taint vector I^t such that an output bit b is tainted. We want to show that there exists an input vector I' and a corresponding taint vector I'^t such that b is tainted, and I'^t has exactly one bit set to 1. Remember that if taint flows through C , then there exists two input words I'_0 and I'_1 that match on untainted bits but that produce a different output for C . We consider the subset of bits in I , called I_{tainted} , in which I^t has a 1. We create a gray code for all values that I_{tainted} can take. As we know that there exist two values of I_{tainted} that produce a different output for C . By enumerating the gray code, we will hence find

a pair of input vectors I_0 and I_1 that differ in exactly one bit and that produce a different output for C . Tainting this bit is sufficient to taint the output bit b for a well-chosen I (which is any of I_0 or I_1).

Appendix B.

Additional Implementation Details

B.1. Further specification details

B.1.1. Proving IFI compliance

We first discuss how we prove IFI for one instruction, and then how we prove it for any sequence of instructions. Cases a - b.2.4 mentioned below correspond to the cases in Figure 7. The SIF model defines the modifications of the architectural taint states in space and time caused by the execution of an instruction sequence i_n , starting in some starting state A_0 . and taint states A_{t,i_s}^{SIF} .

IFI for one instruction. By definition, IFI is satisfied in the clock cycle $clk \geq 0$ such that $c_{id,clk} = i_n$, iff $A_{t,i_n}^{CPU} = A_{t,i_n}^{SIF}$. This means that IFI is violated in clk if one of the following two conditions is true:

- a. some state bits $s \in A_{t,i_n}^{CPU}$ are *not* tainted in the CPU, but the corresponding state bits in A_{t,i_n}^{SIF} are tainted.
- b. some state bits $s \in A_{t,i_n}^{CPU}$ are tainted in the CPU, but the corresponding state bits in A_{t,i_n}^{SIF} are *not* tainted.

We verify a. with the freshness property, which ensures that no information from older instructions leaks through the destination states of an instruction.

We verify b. in two parts. IFI is violated if

b.1. (excessive taint in destinations) some $s \in A_{t,i_n}^{CPU}$ is tainted, where the corresponding $s \in dest_{i_n}$ (the instruction's destination registers).

b.2. (excessive taint in non-destinations) some $s \in A_{t,i_n}^{CPU}$ is tainted, where the corresponding $s \notin dest_{i_n}$.

We verify b.1. with the masking property, and b.2. with the confinement property.

For instructions that cause an exception that is defined within the specification, we set $clk \mid c_{id,clk} = i_n$ to the time of the completion of the exception treating within the CPU and verify IFI analogously. The IFs caused in this case are modelled in the corresponding SIF rule, e.g. see rule BEQ-W-COMPRESSED in Figure 5.

If i_n is an illegal instruction (according to the specification), no SIF rule matches, and thus the taint states must remain unmodified. The containment property captures this case.

IFI for instruction sequences. IFI is further violated if an instruction gets executed in some starting state, so that according to the SIF rule of i_{n+1} no taint propagates to some $s \in A_{t,i_{n+1}}^{SIF}$, but the corresponding state in $A_{t,i_{n+1}}^{CPU}$ is tainted. The only way that taint can appear in $A_{t,i_{n+1}}^{CPU}$ in such cases is by taint propagating from some earlier $A_{t,n < n+1}^{CPU}$. For the cases where IFI was already violated in the base case, the extension of the instruction sequence is not of interest. We discuss all other possible cases next.

Assuming instruction execution up to i_n led to some state $A_{t,i_n}^{CPU} = A_{t,i_n}^{SIF}$.

Case a. from above is equally verified by the freshness property verified for i_{n+1} .

Case b.1. (excessive taint in destinations): Some s gets tainted where s is specified to be cleared due to some masking condition specified in the SIF rule of i_{n+1} , in

LOAD	$I(i_n) = \text{load } rd, rs1, offset \quad t_{rd} = R_t[rs1] \vee M_t(rs1 + offset) \quad t_{pc} = R_t[rs1] \vee PC_t$
$\langle R_t, CSR_t, M_t, PC_t, \mu_t, A, \mu \rangle \xrightarrow{\text{load}} \langle R_t[rd \mapsto t_{rd}], CSR_t[mtval] = R_t[rs1], CSR_t[mepc] = t_{pc}, M_t, PC_t = t_{pc}, \mu_t[m_excp] = R_t[rs1], A', \mu[c_{id} \mapsto i_n] \rangle$	
JALR-NO-COMPRESSED	$I(i_n) = \text{jalt } rd, rs1, offset \quad t = R_t[rs1] \vee PC_t$
$\langle R_t, CSR_t, M_t, PC_t, \mu_t, A, \mu \rangle \xrightarrow{\text{jalt}} \langle R_t[rd \mapsto 0], CSR_t[mtval] = R_t[rs1], CSR_t[mepc] = t, M_t, PC_t = t, \mu_t[m_excp] = R_t[rs1], A', \mu[c_{id} \mapsto i_n] \rangle$	
JALR-W-COMPRESSED	$I(i_n) = \text{jalt } rd, rs1, offset \quad t = R_t[rs1] \vee PC_t$
$\langle R_t, CSR_t, M_t, PC_t, \mu_t, A, \mu \rangle \xrightarrow{\text{jalt}} \langle R_t[rd \mapsto PC_t], CSR_t, M_t, PC_t = t, \mu_t, A', \mu[c_{id} \mapsto i_n] \rangle$	

Figure 11: Taint propagation rules for a subset of the RISC-V instructions. JALR-NO-COMPRESSED is for a CPU that does not implement the compressed instruction set extension and thus jalr instructions can trigger misaligned exceptions that leak information about rs1.

dependence of a source register $src_{i_{n+1}}$. This violates the masking property of i_{n+1} .

Case b.2 (excessive taint in non-destinations) can occur in three ways: s gets tainted

b.2.1. in dependence of some $s \in A_{t, i_n}^{CPU} \mid s \in src_{i_{n+1}}$, where however the $dest_{i_{n+1}} \in A_{t, i_{n+1}}^{SIF}$ is not tainted, but the corresponding $s \in A_{t, i_{n+1}}^{CPU}$ is. This includes also all IFs originating in $src_{i_{n \neq n+1}}$ that cause taint in $src_{i_{n+1}}$. This violates the confinement property of i_{n+1} .

b.2.2. in dependence of some $s \in A_{t, i_n}^{CPU} \mid s \notin src_{i_{n+1}}$. This violates the containment property.

b.2.3. in dependence of some $src_{i_{n \neq n+1}}$, where, however, this dependence is not produced by specified IFs through $src_{i_{n \neq n+1}}$. Thus, taint may have propagated from the source of an instruction to some microarchitectural buffer, e.g., a cache, and then reached $A_{t, i_{n+1}}^{CPU}$. This violates the confinement property of $i_{n \neq n+1}$.

b.2.4. independent of an instruction execution. As above, taint may have propagated through some microarchitectural buffer and manifested in A_{t, i_n}^{CPU} before or at the commit time of i_{n+1} . This violates the containment property.