μCFI: Formal Verification of Microarchitectural Control-flow Integrity

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

Flavien Solt
ETH Zurich
Zurich, Switzerland
flsolt@ethz.ch

Kaveh Razavi
ETH Zurich
Zurich, Switzerland
kaveh@ethz.ch

Abstract

Formal verification of hardware often requires the creation of clock-cycle accurate properties that need tedious and error-prone adaptations for each design. Property violations further require attention from verification engineers to identify affected instructions. This oftentimes manual effort hinders the adoption of formal verification at scale. This paper introduces Microarchitectural Control-Flow Integrity (μCFI), a new general security property that can capture multiple classes of vulnerabilities under different threat models, most notably the microarchitectural violation of constant-time execution and (micro-)architectural vulnerabilities that allow an attacker to hijack the (architectural) control flow. We show a novel approach for the verification of μCFI using a single property that checks for information flows from instruction operands to the program counter by injecting taint at appropriate clock cycles. To check arbitrary sequences of instructions and associate property violations to a specific Instruction Under Verification (IUV), we propose techniques for declassifying tainted data when it is being written to registers and forwarded from the IUV through architecturally known paths. We show that our verification approach is low effort (e.g., requires tagging six signals) while capturing all interactions between unbounded sequences of instructions in the extended threat model of μCFI. We verify four RISC-V CPUs against μCFI and prove that μCFI is satisfied in many cases while detecting five new security vulnerabilities (4 CVEs), three of which are in Ibex, which has already been checked by state-of-the-art verification approaches.

CCS Concepts

• **Security and privacy → Logic and verification**.

Keywords

Hardware security; formal hardware verification; side-channels

ACM Reference Format:
Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μCFI: Formal Verification of Microarchitectural Control-flow Integrity. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3658644.3690344

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690344

1 Introduction

With the increasing cost of compromising software due to the plethora of mitigations and security analysis techniques [1, 2, 4, 8, 11, 30, 45, 62, 65, 93, 107, 111], the focus is shifting towards hardware vulnerabilities, highlighting the need for better hardware security analysis [21, 42, 44, 56, 58, 60, 67, 68, 78, 90, 96, 98, 103, 104, 105]. While there are recent advances in the area of hardware fuzzing [16, 17, 19, 26, 48, 54, 55, 59, 88, 95], unlike formal verification, these fuzzers are not complete by design. Formal verification, however, requires significant per-design human effort due to the lack of generic security properties and simple verification methods that apply to various hardware designs. This paper introduces *Microarchitectural Control-Flow Integrity* (μCFI), a generic security property that captures constant-time and control-flow violations at the microarchitectural level and builds a novel approach for formally verifying this new security property on existing open-source RISC-V CPUs.

Hardware verification. To formally verify a hardware design at the Register Transfer Level (RTL), verification engineers often need to specify design-specific (security) properties which can then be evaluated with a model checker. According to a recent study [87], more than half of the human effort in the development of new hardware designs is dedicated to verification, with debugging being the largest effort (47%) during verification. With the increasing popularity of open-source RISC-V CPUs, their fast community-driven development cycles, and the increasing number of hardware vulnerabilities, reducing the verification effort is of utmost importance for ensuring reliable and secure CPUs in the future. New security properties that generalize to existing designs have the potential to capture different classes of security vulnerabilities while reducing the verification effort through automation.

μCFI. We define the microarchitectural control flow as the Program Counter (PC) values at each clock cycle and make a key observation that a single generic security property can capture multiple classes of hardware vulnerabilities, such as constant-time violations or control-flow hijacks. This new property, which we call *Microarchitectural Control-Flow Integrity* (μCFI), enforces that the microarchitectural control flow is only influenced by instructions for which the Instruction Set Architecture (ISA) explicitly allows a control or data path from their operands to the PC. Data-dependent execution timing of a given instruction causes the PC to have different values at certain clock cycles, violating μCFI. Furthermore, (micro-)architectural vulnerabilities that allow an attacker's input to directly control the PC also violate μCFI. Hence, the verification of μCFI captures both classes of security vulnerabilities.

μCFI verification. Previous approaches that aim to verify the constant-time subset of μCFI either require manual extraction of

design conditions and specification of candidate invariants [33, 34, 101], or cannot verify the interaction between secure and insecure instructions [33, 34, 35], and neither can provide security classifications per instruction for unconstrained instruction sequences. Our verification approach aims to address these challenges in the broader threat model of μCFI. We make a key observation that μCFI can be verified using a single property that checks for all information flows from critical input data, encompassing secret or attacker-controlled data, to the PC. By declassifying valid flows through instruction writeback and forwarding paths, μCFI can capture information flows from instructions to in-flight or future instructions as well. The μCFI property can attribute property violations to specific instructions and can check arbitrary and unbounded sequences of instructions, where instructions may operate on any combination of public, secret, or attacker-controlled data.

Formal verification of Information Flow Tracking (IFT) properties using taint logic has so far been limited to individual modules and properties expressed over interface signals [5, 47, 114]. Leveraging and extending the state-of-the-art IFT logic, CellIFT [89], combined with a state-of-the-art model checker [18], we formally verify μCFI expressed over CPU internal signals spanning the entire CPU pipeline. We evaluate μCFI against four open-source in-order RISC-V processors: Kronos [57], PicoRV32 [75], Ibex [49], and Scarv [85]. We automatically (dis-)prove μCFI for an unbounded number of clock cycles, and find five new vulnerabilities: two in Kronos that had previously undergone fuzz testing [88] and three in Ibex that had already been extensively verified [34, 35, 50, 101].

Contributions. We make the following contributions:

- We introduce μCFI, a generic security property, which enforces that cycle-accurate values of the PC are not influenced by microarchitectural data dependencies, except via explicitly ISA-specified control or data paths.
- We formally define μCFI as a generic information flow property.
- We develop the first generic and automated verification method for formally proving the μCFI property using SystemVerilog Assertions and an existing open-source IFT mechanism [89], reusable across in-order RISC-V CPU designs, and capable of verifying arbitrary and unbounded instruction sequences.
- We show that formal verification of cell-level taint tracking can scale to in-order RISC-V CPU designs. We prove μCFI or find counterexamples by verifying four RISC-V CPUs. In particular, we find five new vulnerabilities: three in Kronos that had previously undergone fuzz testing [88] and two in Ibex that had already been extensively verified [34, 35, 50, 101].

Open sourcing. We open source our toolchain to the extent that the commercial licenses permit. More information can be found at: https://comsec.ethz.ch/mucfi. The extended version of this paper, including appendices, can be found in [25].

2 Background

We discuss CT and CFI, two generic software security properties (Section 2.1) before providing background on hardware IFT and formal property verification (Section 2.2). We then motivate why microarchitectural security could similarly benefit from generic hardware security properties and their verification (Section 2.3).

2.1 General software security properties

General software security properties can protect software against various classes of software vulnerabilities like memory safety [10, 27, 83, 100, 102] and Control-Flow Integrity (CFI) [1, 14, 37, 84]. Some properties like constant-time (CT) programming [2, 3, 43, 70] define guidelines for implementing software that is secure against information leakage through software and hardware.

CFI. Control-flow hijacking attacks like ROP [76] or JOP [12] abuse software vulnerabilities like buffer overflows to divert the Control Flow (CF) of programs to carefully picked gadgets that enable attackers to gain complete control over systems [20, 64, 105]. CFI aims to protect systems by enforcing the integrity of the CF of a potentially vulnerable program at runtime [1, 108]. Software CFI mechanisms check the validity of a program's CF at the ISA level and detect variations when an attacker input influences the CF [64]. Like all software security techniques, CFI assumes hardware to be ISA-compliant. However, the continuous discovery of security vulnerabilities in hardware designs demonstrates the need for hardware-level security verification [44, 64, 68, 88, 90, 96].

CT. Timing side-channel attacks observe secret-dependent program execution times to infer secrets. To prevent such leakages, the CT programming principle states that secret data must not (a) influence the control flow of a program, (b) control memory accesses, and (c) be passed to arithmetic instructions with data-dependent execution latencies [53, 61, 69]. A large body of research on formal verification methods attempts to guarantee CT at the software level [2, 3, 7, 21, 31]. These methods rely on abstract models of the hardware, with the ISA being the formal contract between software and hardware. Intel's recent announcement of the DOIT mode [52], RISC-V's Data-Independent Execution Latency (DIEL) mode [81], as well as already-implemented data-dependent optimizations [72, 99] highlight the need for reliable methods for classifying instructions with respect to their CT property at the microarchitectural level.

2.2 Verification of hardware designs

Two known techniques for verifying hardware designs against security vulnerabilities are Information Flow Tracking (IFT) and Formal Property Verification (FPV).

Information Flow Tracking (IFT). Dynamic IFT, also known as taint tracking, was initially designed for following the propagation of information from (typically) user-defined *taint sources* to *taint sinks* through a software program [92]. IFT has been adapted to hardware and used in static and dynamic settings [46]. GLIFT [94] proposes to instrument each logic gate with additional *shadow logic*, which propagates labels that carry information about whether a signal is affected by the value of a taint source, i.e., if the signal is *tainted*. RTLIFT [6] operates on the Hardware Description Language (HDL). CellIFT [89] operates at the (macro-)cell level, is open-source, and has been shown to scale to the simulation of complex open-source CPUs.

Formal Property Verification. Dynamic testing can usually not guarantee the absence of security vulnerabilities because the set of simulated stimuli is rarely exhaustive. Modern model checkers can exhaustively verify a property expressed over hardware design signals, considering all possible input sequences [86] over unbounded clock cycles, using methods like interpolants [66] or inductive invariants [15]. Logic abstraction can be used to disconnect

an internal signal from its driving logic, effectively making it an input. Design inputs and abstracted signals may be left unconstrained to consider arbitrary values, or constrained to specific sequences via formal assumptions [22, 86]. Scalability remains a challenge, which techniques like abstraction, modularization, and the addition of invariants help to overcome [41, 79]. Model checking can efficiently find property violations and present counterexamples that demonstrate an input trace that leads to a violation [28]. The state-of-the-art SystemVerilog Assertions (SVA) language can express Linear Temporal Logic (LTL) properties, enhanced with sequential regular expressions, but does not define an information flow operator [51]. Information flow properties are hyperproperties [29] specified over sets of traces. Such properties can be verified with IFT logic [5], miter circuits [33, 34] or self-composition [101]. Manually writing properties is time-consuming and error-prone. One wrongly specified bit or clock cycle delay could lead to a false proof, leading to wrong confidence in the design's correctness [23, 24]. Generic and automated formal verification methods can alleviate some of these burdens [73, 74, 80].

2.3 Motivation

Software properties like CT and CFI abstract away hardware details and, therefore, cannot consider clock-cycle accurate control flow variations caused by data dependencies. A program may be proven to comply with software security concepts on an ISA-abstracted hardware, while actual processor implementations might undermine these guarantees in many ways [90]. Hence, cycle-granular dependencies must be verified at the hardware level. While formal methods have a long history in hardware verification, completely proving all functional and non-functional (e.g., security) aspects of CPU implementations is often infeasible due to the complexity of the verification problem and the required human effort [38, 79, 86]. Targeted security properties capturing different classes of security vulnerabilities have the potential to increase trust in a hardware design even though it is not completely formally verified, e.g., due to high cost. Generic CPU properties make their application practical and low effort. With the ISA being the interface between HW and SW, such properties can provide microarchitectural guarantees at the instruction level. We define one such property and show how it enables automated verification of hardware against microarchitectural constant-time and control-flow hijacking vulnerabilities.

3 Microarchitectural Control-flow Integrity

The (architectural) control flow of a program is defined as the sequence of architecturally-visible program addresses. Given this, we define the microarchitectural CF as follows:

Definition 1 (μCF). A Microarchitectural control flow (μCF) is the clock-cycle accurate sequence of program counter values.

The difference between architectural and microarchitectural control flow is the granularity of Program Counter (PC) changes. The μCF affects the clock-cycle-accurate PC valuation, while the architectural CF affects the PC value only at the instruction granularity. We posit that the μCF can capture behavior relevant to various classes of microarchitectural vulnerabilities.

As an example, the Constant Time (CT) programming principle selects a group of instructions from the ISA that a program may use

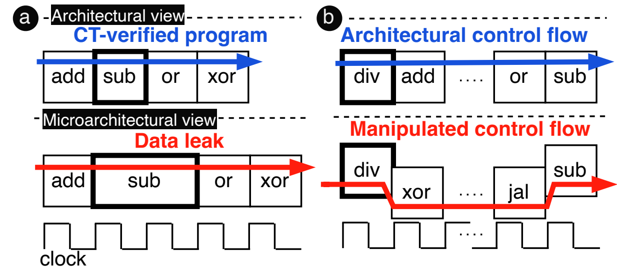

Figure 1: Examples of vulnerabilities only captured in the microarchitectural view. ⓐ Data leak via instruction timing and ⓑ an attacker-diverted control flow.

when operating on secret data [61, 69]. Their timing must be independent of their operand values. In other words, a CT instruction must never influence a program's execution time depending on its operand values. From the perspective of the μCF, the PC values during instruction execution always follow the same pattern for a CT instruction in a given pipeline context, and its operand values do not influence the PC values of any in-flight or future instructions. Figure 1-ⓐ depicts an instruction sequence that is executed with two different data values for instruction *sub*. Architecturally, both execution sequences satisfy the CT programming principle. However, if the microarchitecture implements data-dependent optimizations (i.e., violating the CT principle), the μCF of two executions of the same instruction may differ in a data-dependent manner.

Another example is a vulnerability that enables an attacker to hijack program execution in the absence of software vulnerabilities. Figure 1-ⓑ shows two instruction sequences, where the *div* instruction takes different values in each of them. In an ISA that specifies no arithmetic exceptions, like RISC-V, a *div* instruction must never trap, and execution should continue with the instruction at the next program address. A vulnerable processor implementation may trap in case of a division-by-zero and manipulate the trap return address based on an attacker-provided input. In this case, the *div* instruction may follow a different μCF depending on its operand values, leading to a different architectural CF. As we show in Section 7, issues exist in CPU designs that allow an attacker to hijack the architectural CF even if the software is implemented correctly.

To capture these differences in the μCF, we define a new property called *Microarchitectural Control-Flow Integrity* (μCFI):

Definition 2 (μCFI). Microarchitectural control-flow integrity (μCFI) enforces that the microarchitectural control flow is only influenced by instructions for which the ISA explicitly allows a control or data path from their operands to the program counter.

A hardware implementation that satisfies μCFI guarantees that attacker-controlled data never manipulates the μCF via paths that the ISA does not explicitly allow. Figure 2 shows how μCFI relates to existing software security mechanisms. The CT programming model enforces that the architectural CF of the program should not be secret-dependent. If this property is violated on microarchitectural level, then valid architectural control flows in the program can leak information. The architectural CFI property enforces that the architectural CF of the program should not be manipulated by an attacker outside the valid control flows. Its violation can lead

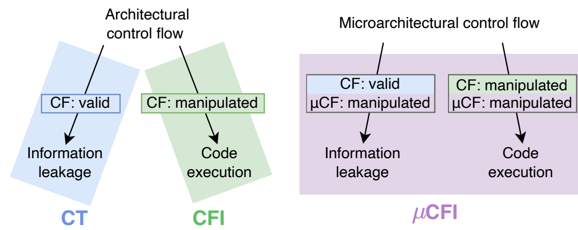

Figure 2: Relating μCFI to CT and CFI. μCFI provides similar security guarantees for the microarchitecture as CT and CFI do for software. Violations of μCFI signal timing violations with valid architectural Control Flow (CF) as well as cases where the architectural CF can be hijacked by an attacker due to microarchitectural vulnerabilities.

Table 1: Information flows verified by μCFI cover four types of information flows and different classes of hardware vulnerabilities.

Architectural Control Flow	Input	
	Secret	**Attacker-controlled**
Invalid (data flow)	Data leak	Control-flow hijack
Valid (timing flow)	CT violation	Delay injection

to control flow hijacking by an attacker. μCFI captures both these properties for a given hardware design at the same time. Note that while μCFI does not capture architectural control-flow issues at the functional level (e.g., invalid branch or jump target calculations), it does capture cases where the architectural CF gets compromised via unspecified microarchitectural control or data paths. Furthermore, violating μCFI does not necessarily lead to an architectural control-flow violation (e.g., operand-dependent instruction timing).

In the rest of this paper, we present a novel formal verification method that detects the violations of μCFI or formally proves their absence automatically by addressing a number of challenges.

4 Threat Model

We assume in-order RISC-V CPUs that have not necessarily undergone full formal verification, as is the case for all RISC-V CPUs that we consider [49, 57, 75, 85]. μCFI verification considers four types of information flows, each leading to a different class of hardware vulnerabilities, as shown in Table 1.

A CF violation occurs when the CPU executes an invalid architectural control flow due to a vulnerability (first row of Table 1). These violations are the result of functional bugs with severe security implications. If a CPU bug triggers an invalid CF that depends on secret data, then the attacker can potentially leak the secret data [110]. For example, imagine that a CPU only triggers a spurious exception if an instruction has a certain operand value. Observing the exception allows the attacker to leak the operand value (**Data leak**). If the invalid CF is caused by attacker-controlled data, then the CPU bug allows the attacker to hijack the architectural control flow, providing them with arbitrary code execution (**Control-flow hijack**). Timing flows can happen even if the architectural CF remains valid (second row of Table 1). In these cases, the information flows are due to timing variations caused by a given instruction provided with different operand values. If the operand is based on secret data, then the information flow through timing results in a **CT violation** that leaks all or part of the operand value. Alternatively, if the operand value is attacker-controlled, then the attacker can perform **Delay injections** attacks, potentially compromising real-time systems or causing instruction re-ordering with architecturally-visible side effects [9].

5 Formalizing μCFI

We now formalize the μCFI property as a Register Transfer Level (RTL) verification problem. Architecturally, the execution time of a program can be measured in instruction retirement counts and in processor clock cycles per instruction [39]. Timing variabilities in the μCF caused by structural hazards, external events, immediate values, and data hazards are not data-dependent and, hence, do not reveal information. However, when the actual number of clock cycles taken by an instruction depends on the data values that are or were being operated on, the timing of the instruction, reflected in when the architectural PC is updated, leaks information about that data. Furthermore, if attacker-chosen data is directly involved in the calculation of the next PC, the attacker can influence and potentially hijack the program's CF [12, 76].

> **Observation 1.** The information flows from instruction's operands to the PC can capture clock-cycle accurate data-dependent timings of instructions, as well as the influence of data on the microarchitectural control flow.

We refer to secret or attacker-controlled data as *critical data*. This observation leads us to define the μCFI property as an information flow property from critical data to the PC. There are different categories of instructions that may operate on critical data and some may legitimately influence the PC, which we discuss next.

Instruction categories. We define instructions as *non-influencing* (*ni*) when the ISA does not explicitly specify an operand-dependent (architectural) CF manipulation. Arithmetic or logic operations are in this category. We define instructions as *control-influencing* (*ci*) instructions if they may only influence the PC via a microarchitectural control path; thus, their operands may only participate in the choice of PC values from a set of otherwise operand-independent targets. For example, operands of branch instructions control whether the branch is taken but are not allowed to change the possible targets, or a *load* may allow a control-flow transfer to a pre-defined exception handler when its operand is a misaligned address. We define instructions as *value-influencing* (*vi*) if the ISA explicitly allows the instruction to manipulate the PC. For example, *jump* instructions are specified to set the PC to a target depending on its operand and the architecturally known immediate. Therefore, the data dependencies of the PC on these instructions' operands are not violating μCFI. When software deliberately allows attacker-controlled data to reach *vi*-instructions, it explicitly allows an attacker to influence the CF. Furthermore, under the CT threat model, compliant software must not pass secret data to *ci*- or *vi*-instructions, as this would violate CT on architectural level.

> **Observation 2.** Instructions can be categorized based on the (dis-)allowed microarchitectural data and control influences of their operands on the PC.

ni-instructions must not influence the μCF and *ci*-instruction may only influence the μCF via control flows. *vi*-instructions are allowed to influence the μCF via their operands.

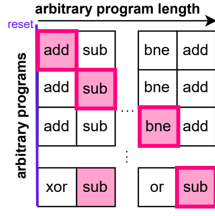

Figure 3: Exhaustively verifying instruction sequences with multiple critical (■) operands. Any instruction can become the IUV (□) at any possible clock cycle in which it could read data from the register file.

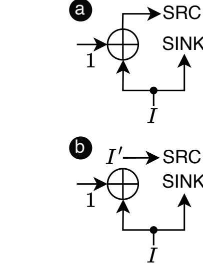

Figure 4: Handling non-causal correlations ⓐ by disconnecting SRC to discover a potential information flow ⓑ.

Arbitrary and interacting instructions. We aim to provide a security classification per instruction with respect to the threat models discussed in Section 4. Classification results should be valid for any program, potentially infinitely long, where any instruction may operate on public, secret, or attacker-controlled data. As shown in Figure 3, we let every instruction become the Instruction Under Verification (IUV) in any position of any instruction sequence starting from the CPU's reset state. Attributing property violations to a specific instruction is challenging [35, 101] due to the inherent parallelism of a pipelined CPU and potential data dependencies of the microarchitectural state. Previous work that studied CT violations proposed excluding insecure instructions (e.g., branches), which violate the CT property, from the instruction stream, providing security guarantees only for programs composed of secure instructions [33, 35]. Real-world programs may interleave secure instructions with insecure instructions that only operate on public data. This seems secure from a software perspective, but due to microarchitectural interactions, an allegedly secure instruction might illegitimately influence an insecure one, even if the latter did not operate on critical data. For example, the *add* in the first row in Figure 3 may influence the timing or target of a younger 'branch if not equal', *bne*, microarchitecturally, hence *add* is insecure when composed with branches. To obtain guarantees for arbitrary contexts, such interactions must also be verified.

To solve this problem, in Section 5.1, we define the μCFI property per instruction and operand, where information flows can be verified in isolation, and each violation can be precisely attributed to one instruction in a specific position in the sequence. However, instructions can interact through architecturally known paths, such as the register file and register-address controlled forwarding paths, which should not be flagged as property violations. UPEC-DIT and ConjunCT avoid this problem by excluding CT-violating instructions from the instruction stream, which reduces the security guarantees [33, 34, 35]. μCFI solves this problem by formalizing generic rules for declassifying legal information flows between instructions, as discussed in Section 5.2.

5.1 The μCFI property

We now define a flow operator and then formalize the μCFI property.

Information flow operator. Equation 1 defines an information flow operator over RTL circuits, which are deterministic finite state machines. Let n be a number of clock cycles, with $n = 0$ the cycle in which the design is in its reset state. Intuitively, an information flow from a source (SRC) to a sink (SINK) signal in a logic circuit exists iff there exists a sequence of values of SRC, $S_{src} := (s_{src,n})_{n\geq 0}$ where a change in at least one bit of a value $s_{src,n}$ of SRC values causes a change of at least one bit of the corresponding sequence of SINK values, $S_{sink} := (s_{sink,n'})_{n'\geq 0}$. To exclude non-causal correlations between SRC and SINK, as illustrated in Figure 4 (a), where a change in input I would affect both SRC and SINK, we disconnect SRC from its driving logic as shown in Figure 4 (b).

Let $\mathbb{I} := (\{0,1\}^j)^{\mathbb{N}}$ be the set of infinite input sequences to the design, where j is the sum of the bit widths of all design inputs, except for SRC, and $\mathbb{N}$ are the natural numbers. There is an information flow for a set $\mathbb{S} \subseteq \mathbb{I}$, denoted by $\overset{IF}{\rightsquigarrow}$, if there exists a sequence $S := (s_n)_{n\geq 0}$ in $\mathbb{S}$, and two (distinct) source sequences $S^A_{src} := (s^A_{src,n})_{n\geq 0}$, $S^B_{src} := (s^B_{src,n})_{n\geq 0} \in \mathbb{U}$ that yield different value sequences of $SINK$ as expressed in Equation 1. Like for $\mathbb{S}$, we define $\mathbb{U}$ as a subset of $(\{0,1\}^b)^{\mathbb{N}}$, where b is the bitwidth of SRC. We define $SINK(S, S_{src})$ as the sequence of values of SINK given the inputs $S \in \mathbb{S}$ and the SRC values $S_{src} \in \mathbb{U}$.

Precise information flow operator:

$$SRC \overset{IF}{\rightsquigarrow}_{\mathbb{S},\mathbb{U}} SINK : \exists S \in \mathbb{S}, S^A_{src}, S^B_{src} \in \mathbb{U} \mid SINK(S, S^A_{src}) \neq SINK(S, S^B_{src}) \tag{1}$$

For being able to express data flows only, we define a data flow operator $\overset{DF}{\rightsquigarrow}$ in the same way as $\overset{IF}{\rightsquigarrow}$, but with the restriction that information flows via control wires of the RTL circuit are excluded. We define control wires as the ones that either control *whether* (time dimension) or *via which path* (spatial dimension) another signal is updated. As we will discuss in Section 6, an existing IFT method can implement the $\overset{IF}{\rightsquigarrow}$ operator, and we build a new data flow tracking method for implementing the $\overset{DF}{\rightsquigarrow}$ operator.

Preliminary definitions. We define μCFI for in-order load-store architecture CPUs with any number of pipeline stages, that load data from memory or a Control and Status Register (CSR) into a General-Purpose Registers (GPR), before an instruction operates on the GPR. We define IW to be a microarchitectural signal that holds the instruction word from which the CPU reads the register addresses $O_{k_addr_{0\leq k<K}}$ needed to read the instruction's operands from the register file. K is the number of operands of the instruction, e.g., $K = 2$ for a branch. We further define O_k as the signal through which operand k's data transitions from the architectural state (the register file) into the microarchitecture. Finally, the microarchitectural PC refers to a register within a logic circuit that fulfills the clock-cycle accurate property $(PC)_{n\geq 0} = (PA)_{n\geq 0}$, where PA refers to the program address of the currently executing instruction.

Defining the μCFI property. Following Definition 1 (Section 3), we define the microarchitectural control flow μCF as a potentially infinite sequence of PC values: $\mu\text{CF} := (PC)_{n\geq 0}$. Intuitively, the μCFI property states that an $ni(IUV)$ never influences the PC via its operands O_k, and that a $ci(IUV)$ only influences the PC via control flows. Following Definition 2, μCFI does not consider vi instructions given ISA-specified data flow from their operands to the PC. A CPU implementation satisfies μCFI if it satisfies the μCFI property for all instructions that it supports, and for all microarchitectural states σ:

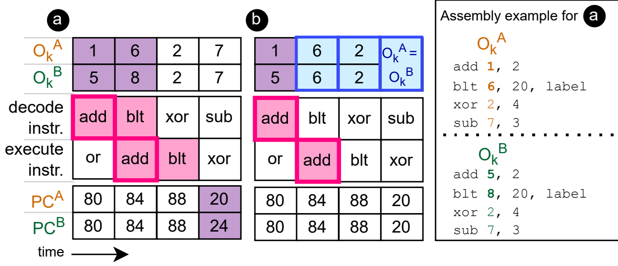

Figure 5: **Precise taint injection.** ⓐ: **Wrong classification:** Both *add* and *blt* read different values (purple background) in sequences O_k^A and O_k^B. Information flow (■) to the PC is wrongly attributed to the *add* (the IUV (■) in this example, marked with a bold border). ⓑ: **Correct classification:** Input sequences to O_k are constrained (■) to be equal between any two sequences compared in one proof whenever O_k is not read by the IUV.

μCFI property:

$$\forall\, IUV, \forall \sigma : \quad (ni(IUV) \implies \neg(\{O_k\}_{k \in K} \overset{IF}{\rightsquigarrow}_{\mathbb{V},\mathbb{O}} PC)) \wedge \quad (ci(IUV) \implies \neg(\{O_k\}_{k \in K} \overset{DF}{\rightsquigarrow}_{\mathbb{V},\mathbb{O}} PC)) \tag{2}$$

where $\mathbb{V} \subseteq \mathbb{I}$ is the set of input sequences that lead to IW equaling IUV's type at a given clock cycle, and $\mathbb{O}$ is the set of source value sequences passed to O_k.

Precise taint injection. By the definition of $\overset{IF}{\rightsquigarrow}$ and $\overset{DF}{\rightsquigarrow}$, O_k must be disconnected from its driving logic, allowing all value sequences to be considered. If two different sequences of O_k can lead to a different value of the PC, then there exists an information flow. Considering the example instruction sequence in Figure 5, if verify the μCFI property without further constraints for O_k, any instruction in that sequence that can read from O_k could potentially cause a property violation. Branch instructions, for example, by definition, influence the PC. When considering unconstrained and infinite input sequences, every possible sequence may be extended by a branch and thus, an information flow from O_k to the PC would be detected. However, in Figure 5, our current IUV is the *add*, so we are not interested in detecting the flow originating from the subsequent 'branch if less than', *blt*. Therefore, while verifying the *add*, instead of excluding branches from the instruction sequence, as done by [33, 34, 35], we constrain the proofs to only consider differences among any two considered sequences to O_k when the *add* can read from O_k, while verifying every possible input sequence.

In Figure 5, case ⓐ exemplifies two differing input sequences to O_k, where both *add* and *blt* read unconstrained data. They cause a different PC value after the *blt* is executed. However, in this example, we were verifying the *add* and would, therefore, wrongly associate the *blt*'s information flow with the *add* instruction. For precise instruction classification, we constrain $\mathbb{O}$ per proof as described in Definition 3 and depicted in ⓑ. No change in input sequences to O_k causes a different PC value when verifying the *add*. When, in a separate proof, the *blt* is the IUV, a difference at the PC can be observed, and it must have been caused by the *blt*'s operands.

Taint injection constraint. To associate a register access with an IUV, we leverage the fact that the register address is part of the instruction word and that it must be read *no later than* its use to access the register. We define n_{addr_k} as a clock cycle in which the register address O_k_addr is read from the instruction word IW. Next, we define $n_{start_k} \mid n_{addr_k} \leq n_{start_k}$ as the next clock cycle in which O_k is read from the register file. Finally, we define $n_{stop_k} \mid n_{start_k} < n_{stop_k}$ as the next clock cycle in which O_k is overwritten. Intuitively, Equation 3 states that O_k^A and O_k^B may only differ when the IUV can read from O_k.

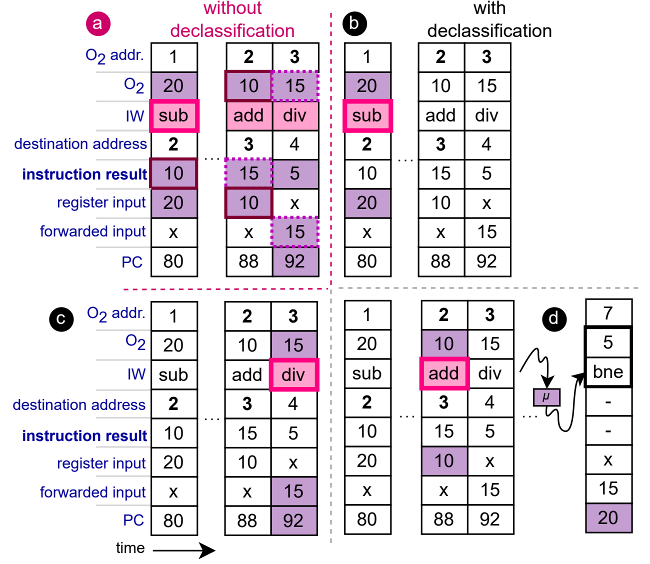

Figure 6: Same instruction sequence, different IUVs (■). ⓐ: Information propagates between instructions through architecturally legal paths (instruction result to O_2, which reads from register or forwarded input). ⓑ-ⓒ: With legal paths declassified, flows can be associated with the causing instruction. ⓓ: Unexpected flows from *add* to a *bne* (here with public operands) are not declassified and thus detected.

Instruction Operand Constraint (IOC):

$$\mathbb{V}, \mathbb{O}_k := \{V, O_k^A, O_k^B \mid IW_{n_{addr}} = IUV \wedge \forall n \notin [n_{start_k}, n_{stop_k}), O_{k,n}^A = O_{k,n}^B\} \tag{3}$$

The IOC isolates the verification of information flows so that only one instruction type can operate on secret data in any sequence. Section 6 shows how our verification method captures information flows from multiple instructions when verifying them individually.

Note that when the IOC forces the SRC (O_k) signals to be equal in some cycles, we do not consider reconvergent flows to the SRC in these cycles. However, when the SRC signals only read from declassified paths (see next) we do not need to check these flows.

5.2 Declassification of legal flows

By definition, information propagates from an instruction's operand to its outputs, i.e., the destination register and register address-controlled forwarding paths. Information furhter propagates to a subsequent instruction that operates on the previous instruction's destination register, which is an architecturally known path. For example, in Figure 6-ⓐ, the IUV is a *sub*. This instruction does not influence the PC, but it passes its result to the subsequent *add* (via 'register input'), which passes its result to the *div* (via 'forwarded input'). Without declassification, the *div*'s potential influence on the PC would be detected when verifying the *sub*. When considering unconstrained instruction sequences, any sequence could be extended

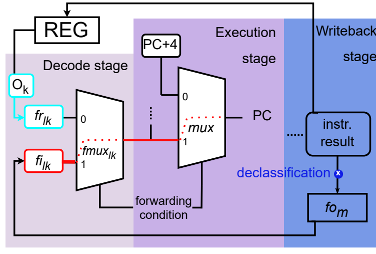

Figure 7: Declassification and detection via forwarding paths. When only monitoring information flows from O_k (cyan), flows from forwarded input, fi_{l_k}, to the l'th forwarding multiplexer, $fMux_{l_k}$, for operand k (red), are missed due to the declassification of forwarded results (fo_m).

with an insecure instruction that receives information from previous instructions via such paths. This architecturally expected interaction between secure and insecure instructions complicates the association of an information flow with a specific instruction. Previous CT verification methods avoid false classifications by restricting the verified instruction sequences by excluding CT-violating instruction types [33, 35], which limits their guarantees to programs composed of secure instructions only (e.g., containing no branches).

In our threat model, the software is responsible for architectural information flows, e.g., from an instruction's input operands to its results. Hence, we declassify information flows via architecturally specified paths during verification, and prove μCFI per instruction type. The IOC constrains the input sequences to pass differing inputs to only one IUV per verified sequence, while it can be surrounded by arbitrary instructions of the same or different types. ⓑ and ⓒ show the same instruction sequence as ⓐ, but with architectural paths (through registers and forwarding paths) declassified. Thus, the *div*'s CT violation does not get detected when verifying *sub* (ⓑ), but when verifying *div* (ⓒ). Unexpected paths between instructions, like depicted in ⓓ, are not declassified and are detected. Since the μCFI property (Equation (2)) is proven for every IUV and every microarchitectural context σ, every instruction in every sequence is considered as IUV in some proof scenario.

Declassification Precondition. Declassification is sound if no unexpected information flows to the PC could have been missed when μCFI is proven. This is guaranteed iff the following precondition is proven on a design: all outgoing paths from a declassified signal converge either into another declassified signal or one of the SRC inputs to instructions before reaching the PC. This precondition ensures that there are no unconsidered information flows in the fanout of declassified signals.

Declassification of register writes. We declassify data written to the destination register of an instruction by disconnecting the register data write signal from its driving logic and adding it to the input sequences in $\mathbb{V}$, allowing the signal to take on arbitrary values that are never influenced by a change in O_k.

Declassification of forwarding paths. CPU implementations can forward instruction results to younger instructions in various pipeline stages. As depicted in Figure 7, we define forwarding multiplexers $fMux_{l_k}$ as the ones that arbitrate between forwarded data inputs fi_{l_k} and register data fr_{l_k}, selected by a condition over matching dependent register addresses between instructions (forwarding condition). l_k is the number of forwarded inputs per operand k (typically present at the input of the execution stage). Forwarded output data fo_m are the signals that convey instruction results via forwarding paths. m is the number of forwarding outputs, e.g., in the execution or writeback stage.

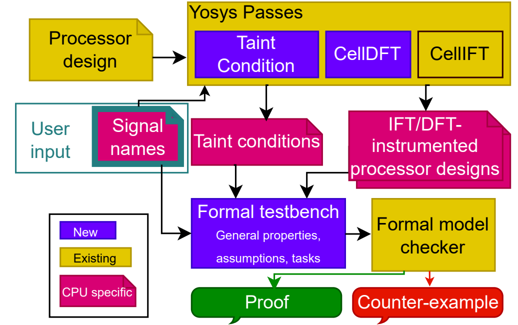

Figure 8: CPU-specific formal verification flow.

To avoid falsely attributing a property violation to an insecure instruction that reads an older instruction's result via forwarding paths, we declassify forwarding output signals, fo_m, by disconnecting them from their driving logic and adding them to the input sequences $\mathbb{V}$. Since an IUV can operate on forwarded data inputs fi_{l_k}, forwarded data must not influence the μCF in unexpected ways. It is possible (although unexpected) that an instruction's operand *only* influences the PC if the instruction operates on forwarded data fi_{l_k}, and not if it operates on register data fr_{l_k}, e.g., as exemplified in Figure 7. When only monitoring data coming from the register, REG, the information flow from fi_{l_k} to the PC would be missed in the scenario shown on the left. Therefore, we add inputs fi_{l_k} to the SRC signals by disconnecting them from their driving logic and extending the IOC constraint as follows, with $\mathbb{F}$ being input sequences to fi_{l_k}:

$$\begin{aligned}\mathbb{V}, \mathbb{O}_k, \mathbb{F}_{l_k} := \{&V, O_k^A, O_k^B, F_{l_k}^A, F_{l_k}^B \mid IOC \\ &\wedge \forall n \notin [n_{start_{l_k}}, n_{stop_{l_k}}), fi_{l_k,n}^A = fi_{l_k,n}^B\}\end{aligned} \quad (4)$$

6 Verifying μCFI

We now present how to express μCFI as RTL design properties using SystemVerilog Assertions (SVAs), logic abstractions, and the bit-precise cell-level IFT [89]. These properties can be verified with any standard model checker that can obtain unbounded proofs for SVAs [18]. First, we prove that cell-level IFT equals the information flow described by the $\overset{IF}{\rightsquigarrow}$ operator defined in Section 5.1 (Section 6.1). We then introduce a new IFT mechanism that only tracks data flows, called CellDFT (Section 6.2). Finally, we describe how to construct the μCFI property in an SVA testbench (Section 6.3) and how to declassify legal flows (Section 6.4).

Our tool flow, depicted in Figure 8, starts with processing a CPU design with three Yosys [106] passes: (i) obtaining the taint conditions (Appendix A.3 in [25]), (ii) cell-level IFT (CellIFT) based on previous work [89], and (iii) CellDFT. The IFT and DFT instrumentations add shadow taint logic to the design. The flow generates a CPU-specific testbench connecting generic SVA assumptions and assertions with the generated CPU-specific logic conditions. Names of the following signals, which can be identified based on their properties (see Section 5.1), must be user-provided: PC, IW (and

register address), signals connected to the read/write port of the register file, and forwarding outputs.

6.1 Modelling information flows with CellIFT

CellIFT [89] defines IFT logic on the cell level but does not specify information flows over spatial and temporal compositions of cells. We prove that if there is an information flow from a source to sink according to Definition 1, CellIFT will propagate taints from the source to the sink. The cell-level information flow rule [89] is given as $C^t(I, I^t)_j = 1 \iff \exists \tilde{I} \mid (I \oplus \tilde{I}) \wedge \overline{I^t} = 0$ and $C(\tilde{I})_j = \overline{C(I)_j}$, where I is a cell's input, I^t its corresponding taint input, C(I) the cell's output, and $C^t(I)$ its corresponding taint output vector. j refers to the j-th bit of the (taint) output vector.

CellIFT instruments designs by adding a so-called shadow logic. Each state bit in the original design is augmented by one state bit in the shadow logic. The shadow logic is designed to convey information flows, i.e., to propagate taints if changes in the tainted input bit values can provoke changes in the output bit values.

If there is an information flow from a SRC to a SINK as defined in Section 5.1, then CellIFT propagates the taint from SRC to SINK, as expressed by Theorem 6.1 proved in Appendix B in [25], where $\mathbb{I}_t$ and $\mathbb{I}_{nt}$ are the sets of input sequences to bits of which the corresponding I^t bits are 1 and 0, respectively and Y is a cell's output.

THEOREM 6.1. *Single-cell CellIFT equivalence.*

$$\overline{I^t} = 0 \text{ and } I \overset{IF}{\rightsquigarrow}_{\mathbb{I}_{nt},\mathbb{I}_t} Y \iff C^t(I, I^t) \neq 0 \quad (5)$$

Oberg et al. [71] showed that taint tracking methods detect timing dependencies. We prove in Appendix B in [25] that CellIFT covers all information flows at design level, i.e., that CellIFT has no false negatives. However, false positives are theoretically possible [89].

6.2 Data flow tracking with CellDFT

Implementing the $\overset{DF}{\rightsquigarrow}$ operator requires distinguishing data and control flows. CellIFT is incapable of it. Yet we observe that the macrocell abstraction level (used by CellIFT) is suitable for making this distinction. We introduce *CellDFT*, a variant of CellIFT that propagates information via data flows only. It expresses data flows by blocking taint propagation via control paths, which we define as paths from a cell's input to its output, that do not perform direct assignments or data-manipulating operations. Control paths are for example paths through comparison results, multiplexer select signals or enable bits, which all control state changes or data flows. CellDFT is a more restrictive variant of CellIFT in the sense that any signal tainted by CellDFT would also be tainted by CellIFT. In the following, we describe the variations from CellIFT that describe CellDFT, which we define in Table 2 and describe below. We have implemented CellDFT as a modified version of the CellIFT Yosys pass and will release it as open source.

State elements with enable condition. We let $(EN_i)_{0 \leq i < N_Q}$ denote the enable condition expanded to the bitwidth N_Q of the output vector Q of the cell, where $\forall i \in (0 \leq i < N_Q)$ holds $EN_i = EN$. D is the input data vector. Intuitively, there is no data flow coming from the enable signal.

Table 2: Cell definitions for CellDFT CellDFT's version of CellIFT rules (where modified), specified over Yosys' cell port names. EN, D, A, B and S are cell inputs, Q_n^t is a state cell output at clock cycle n, Y^t is a combinational cell output, $\circ$ represents shift operators.

Cell Name	Definition
State elems. with enable (EN)	$Q_n^t = (EN \wedge D^t) \vee (\neg EN \wedge Q_{n-1}^t)$
2-input mux, aldff cells [106]	$Y^t = (\neg S \wedge A^t) \vee (S \wedge B^t)$
pmux cells [106]	$Y^t = A[S]^t$
Comparison/reduction cells	$Y^t = 0$
Shift cells	$Y^t = A^t \circ B$

2-input multiplexers and aldff cells [106]. Let S be the bit-expanded select signal (like EN above), with same bit length as Y. Let A and B the selected signals when S is 0 or 1, respectively. Intuitively, there is no data flow coming from S.

pmux cells [106]. Yosys implements multiplexers with multiple inputs $A[0], \ldots, A[n-1]$ as *pmux* cells. The output taint corresponds to the taint of the selected input.

Comparison and reduction cells. This rule describes the following single-output-bit cells: *eq_ne, ge, gt, le, lt, logic_and, logic_or, logic_not, reduce_and, reduce_xor.* None of them propagates data.

Shift cells. We let A denote the data input and B the shift amount. Let $\circ$ denote the cell's operator. Intuitively, the shift amount is not part of the data flow, but controls A's data flow.

6.3 μCFI expressed over taint logic

We now formulate the μCFI property stated in Equation 2 as SVAs [51, 86] over a CellIFT-/CellDFT- instrumented design.

Formal setup overview. A SystemVerilog bind statement inserts SVA assumptions and assertions into the CPU's top module. We abstract [86] the memory so that the CPU receives unconstrained and infinitely long instruction sequences (including illegal ones) via its instruction word input. We leave all original design inputs unconstrained, which models arbitrary external interrupts, bus errors, etc [22]. Definition 1 requires that the information source (SRC, i.e., O_k) is disconnected from its driving logic. Using the taint tracking logic, we only abstract the taint signal of O_k, while leaving the design signal untouched. This models unconstrained inputs, because taint propagation is symbolic, i.e., design bit values, of which the corresponding taint bits are set, are not influencing the taint propagation [89]. Furthermore, we constrain all taint input signals at the top level to constant zero during the whole proof. This guarantees that the operand data is the sole taint source. We constrain the taint source as described in Section 5.1, for automatic identification of a specific instruction and operand combination as root cause of the μCFI property violation, without needing further analysis or human interpretation as is often the case [34, 35, 101]. We implement declassification of valid flows as defined in Section 5.2 and state the μCFI property as an SVA assertion regarding the PC's taint. In the following we detail these steps.

Taint injection location In Section 5.1, we defined O_k as the signal through which register data passes first when entering the microarchitecture (O_k could be the same for all k operands). Figure 9 categorizes cases of O_k interfacing with the register file (REG). In ❶, O_k is a microarchitectural state element that is directly connected to the REG's read port, but is updated only when its enable condition

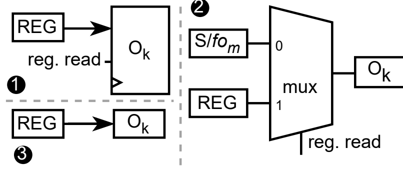

Figure 9: Taint injection through O_k. In ❶, O_k reads from REG conditionally. In ❷, O_k reads from REG via some cell, e.g., via a multiplexer that arbitrates between REG and any other signal "S" (e.g. fo_m). In ❸, O_k reads from REG unconditionally.

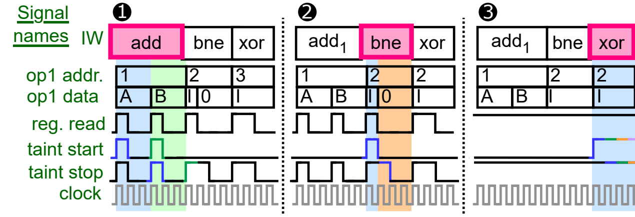

Figure 10: Taint injection examples. Corresponding taint start and stop cycles are drawn in equal line color. Taint injection windows start (e.g., blue taint start) in every clock cycle in which IW matches IUV's (■) type and IUV reads (reg. read high) from the register file (blue/green background) through O_k (op1 data), and stop (e.g., blue taint stop) with the next clock cycle after taint start in which O_k may change. ❶: Two reads per IUV, 2nd one e.g., due to writeback (green background). ❷: O_k is invalidated (orange background) after the read. ❸: Since every clock cycle where IW=*xor* is a taint start and stop condition, there is one taint window per clock cycle.

is true. ❷ refers to all cases where O_k may read from REG via some logic cell. Here, O_k may be a wire or a state element. Finally, ❸ refers to cases where O_k reads from REG unconditionally, i.e., either continuously if it is a wire or in every clock cycle if it is a state element.

Temporal aspects of taint injection. Following the formalization in Section 5.1, and in particular the IOC defined in Equations (3) and (4), we verify μCFI per instruction and operand and let any instruction become the IUV at any possible clock cycle in which it could read data from the register file.

In the formal testbench, we sample the instruction word signal IW in the same condition in which the CPU reads the register address from it. This condition models n_{addr_k}, and we can extract it automatically as discussed in Appendix A.3 in [25]. We associate any register read with the latest sampled instruction. Whenever at the same time a *register reading condition* happens, and IW holds an instruction of the currently examined type, this instruction turns into the IUV, and the reading cycle becomes n_{start_k} (i.e., a **taint start** signal is set), taint is injected into the abstracted taint signal of O_k. A *register reading condition* is any condition in which O_k can read the latest value of REG. Note that the *register reading condition* does not depend on the actual value change of O_k, i.e., a new taint window also starts in case an instruction reads from the same register address as the previous one or if the values in two consecutive instruction's input registers match coincidentally.

Figure 9, cases ❶ and ❷ have a dedicated read enable signal. In case ❸, the reading condition is always true when IW matches the IUV's type, as O_k reads from REG unconditionally. We stop taint injection any time O_k's value **may** change again, i.e., in any clock cycle in which O_k is not updated with its own previous value. Note that the **taint stop** condition is also true when a signal is overwritten with the same value as its previous one, e.g., if a register is updated to an instruction's result that coincidentally matches the previous register value. This **taint stop** condition models n_{stop_k}.

Examples of taint injection. Figure 10 shows an example of taint injection for each of the three cases when reading registers. Taint start and stop signals of one injection window are shown in matching line colors. Example ❶ shows an *add* that reads from register 1 two times. The second read (green background) could happen if the *add* was stalled and register 1 is updated due to a writeback. The IOC constraint considers both reading conditions, each being a **taint start** condition for an individual taint injection window. Hence, the first read's **taint stop** condition overlaps with the second read's **taint start** condition. In Example ❷, O_k is overwritten with non-register values (orange background) after the first read. In this case, the **taint stop** condition is true, but no new **taint start** condition happens. In case ❸, the **taint start** (and **taint stop**) condition is true in every clock cycle in which IW matches the current IUV's type, meaning that a taint flow originating from all of these cycles is verified (individually).

Discussion. In Figure 9, case ❷, unexpected taint flowing to S cannot propagate further, because O_k's taint signal is abstracted, which means that the logic between S and O_k is ignored in the proof. To avoid missing flows, we check that O_k is only driven by declassified flows by either choosing the REG's output port as O_k or by ensuring that the only cell between REG's output and O_k is a forwarding multiplexer through which we inject taint as well (Section 6.4). If an instruction mistakenly would not read any data (e.g., due to a pipeline flush) the μCFI property would trivially hold.

Automated taint injection. We control the taint source with a generic set of SVA assumptions detailed in Appendix A.2 in [25], to which we pass the signal names, the *register reading condition* and the *taint stop condition*. In Appendix A.3 in [25], we detail how we automatically extract these conditions from CPU designs via static design analysis. That way, we do not rely on fully verified CPU functionality as previous work [101], or detailed design knowledge as [33, 101], but rather track data whenever the CPU under verification in its current state of implementation can let architectural data enter the microarchitectural world.

Information flow detection. A CPU design can store the PC in multiple pipeline registers. An appropriate taint sink is one that holds the current PC for every instruction, e.g., one passed to the execution stage. Since tainted data must never reach the PC, we can prove the absence of information flow unconditionally by asserting that the taint signal of the PC is always zero. Since the property is proven in an isolated environment per instruction and operand with the operand as the only taint source, a violation is directly attributable to this instruction and operand combination. A proof of this property guarantees μCFI for the same combination.

From individual to full proofs. In Appendix A.1 in [25], we prove that if a set of inputs and input taints results in some output bit being tainted, then all but one tainted input bit can be untainted while preserving the output bit taint, given a well-chosen valuation of the design bits of which the corresponding taint bits were previously tainted. Given this property of CellIFT, it is sufficient to prove

μCFI for all sequences in which one IUV could have read tainted data to guarantee μCFI for all other sequences in which the IUV may be interleaved with any other instructions that may operate on untainted (public), or tainted (secret or attacker-controlled) data.

6.4 Declassification of legal flows

In Section 5.2, we defined which information flows are legal within our threat models and explained why they need to be declassified. In the formal verification setup, we declassify flows by abstracting the taint signal of a chosen design signal and constraining it to zero with an SVA assumption. As mentioned in Section 5, the declassification precondition must be satisifed per verified CPU design.

Register file. We declassify data written to the register via a signal REG_WRITE_DATA that is directly connected to the register write port. Checking the functional correctness of register writes is out of the scope of μCFI verification. The declassification precondition is guaranteed if register write data can only propagate to the signal connected to the register read port, which is our taint source.

Forwarding paths. Recall that the forwarding outputs fo_m forward instruction results to earlier pipeline stages, fi_{l_k} and fr_{l_k} are the forwarded and register inputs to an instruction, respectively, and $fMux_{l_k}$ selects between the two. Our tool takes user-provided fo_m signals for declassification and checks that all of their fanout logic feeds into fi_{l_k} or REG_WRITE_DATA signals. If we then consider all fi_{l_k} as additional taint sources to the IUV, the declassification precondition is fulfilled.

Constraining taint of forwarded inputs. To avoid having to identify all clock cycles in which an instruction can operate on fi_{l_k} in different pipeline stages and to keep our method generic across CPU designs, we let the IUV operate on fr_{l_k} in the forwarding cases by abstracting and unconstraining the $fMux_{l_k}$' select signals, while the corresponding taint signal remains unchanged, and keep O_k as the only taint sources. However, although register data can reach fr_{l_k} when the forwarding condition is false, it might not reach it in the forwarding case. Therefore, we extend the IOC (Equation (4)) with the following:

$$\mathbb{V}, \mathbb{O}_k, \mathbb{F}_{l_k} := \{V, O_k^A, O_k^B, F_{l_k}^A, F_{l_k}^B \mid IOC \quad (6)$$
$$\land \forall n \in [n_{start_{l_k}}, n_{stop_{l_k}}) \mid fr_{l_k,n}^A \neq fr_{l_k,n}^B \implies fi_{l_k,n}^A \neq fi_{l_k,n}^B\}$$

Intuitively, this equation states that whenever an instruction can read from O_k, if fr_{l_k} is tainted, also fi_{l_k} is tainted. Taint reachability condition: This is guaranteed, as we show in Appendix D.1 in [25], if we can prove per verified CPU that fr_{l_k} is an unconditional and undelayed assignment from O_k, or, if there is a cell on the path from O_k to fr_{l_k}, it is another $fMux_{l_k}$ that fulfills the same condition.

Lastly, if we inject taint not directly into the register reading port but into O_k (to account for potential conditional reads), the declassification of the *mux* shown in example ❷ of Figure 9 would happen (structurally) before O_k. Therefore, if there is an $fMux_{l_k}$ between O_k and REG, where fr_{l_k} equals REG, the **taint start** condition has to be extended by the case where O_k reads from fo_m.

Automated declassification. Our Yosys pass automatically extracts $fMux_{l_k}$ select signals via static design analysis, based on the user provided fo_m signals. It verifies the *declassification precondition* by traversing all outgoing cell connections of fo_m until reaching an $fMux_{l_k}$, O_k or a declassified signal, and checking that the user provided PC signal has not been passed. It further checks the *taint reachability condition* and informs the taint condition generation about additional $fMux_{l_k}$ before O_k.

Table 3: Design complexity comparison of the uninstrumented (U), CellDFT- (D), and CellIFT (I)-instrumented design considering nets, gates, and register (R.) bits. All values in thousands (k).

Design	Nets [k]			Gates [k]			R. Bits [k]		
	U	D	I	U	D	I	U	D	I
Kronos	1.4	3.0	6.8	13.0	43.4	77.1	2.0	3.9	3.9
PicoRV32	1.6	3.8	9.5	27.0	67.6	114.3	3.2	5.1	5.3
Scarv	6.7	11.5	30.1	58.7	176.6	309.3	2.3	4.6	4.6
Ibex (small)	4.5	8.0	17.5	39.9	82.5	160.5	2.4	4.7	4.7
Ibex (custom)	4.6	8.5	18.3	40.7	86.2	166.7	2.5	4.9	5.0

7 Evaluation

We evaluate the proposed μCFI verification method in terms of runtime and verification results for four open-source RISC-V designs. PicoRV32 [75] (f00a88c3) is a size-optimized CPU written almost entirely in one Verilog module. Kronos [57] (a41629d) is a CPU designed for FPGA applications, written in SystemVerilog. For PicoRV32 and Kronos we verify CPU versions with fixes for bugs found by the state-of-the-art Cascade CPU fuzzer [88]. We also include Scarv [85] (bb52627) which is a side-channel hardened CPU implementing the RISC-V scalar cryptography extensions [81], and Ibex [49] (bbb91c56, opentitan fork), an extensively verified CPU [34, 35, 50, 101, 113] used in real-world designs such as the OpenTitan root of trust [63].

Testbed. We formally verified the SVAs with Cadence's Jasper Formal Property Verification (FPV) App, v2022.09 [18], configured to provide proofs for an unbounded number of cycles, executed on a server with the following configuration: Intel Xeon, 3.4 GHz, 60 logical cores, 1.25 TB RAM.

Cost of instrumentation. While CellIFT is designed to have a small performance and area overhead [89], its overhead in terms of netlist composition has not yet been extensively studied. Table 3 reports design complexity statistics. The instrumentation multiplies the number of nets by a factor 4.0 (Ibex small, secure) to 5.9 (PicoRV32). The number of gates is multiplied by a factor of 4.1 (Ibex small, secure) to 5.9 (Kronos). CellDFT only adds a net and gate overhead of max. 2.4x (PicoRV32) and 3.4x (Kronos) respectively. The instrumentation serves verification purposes only and is absent in physical CPU implementations; hence, area overheads are not relevant in the verification context. CellIFT logic added a larger gate overhead to Scarv despite similar state bit counts to Ibex, which explains the larger proof runtime on Scarv.

Annotation burden. 6 signals need to be manually extracted from all CPU designs, based on their mentioned properties in Section 5.1 and Section 6.3: IW, one register address, signals connected to the register read and write ports and the PC. Forwarding output annotations depend on the number of pipeline stages that forward results: 0 for PicoRV32, 1 for Ibex and 0 for Kronos, because it reuses the register write signal. Scarv has wider outputs for cryptography instructions, which adds 5 annotations. For the forwarding declassification precondition check the name of the register file is needed.

Table 4: Verification runtime (MM:HH:SS) for all four considered RISC-V CPUs. (I = CellIFT, D = CellDFT, t. = time, c = custom, s = small). Ibex configurations: secure + slow multiplier.

	↶ Method / Design →	Kronos	PicoRV32	Ibex (s)	Ibex (c)	Scarv
I	Mean t. FAIL	0:00:37	1:05:47	2:25:22	3:18:06	0:10:56
	Mean t. PROVE	0:16:29	16:55:43	6:14:33	8:46:12	14:16:56
	Max. t. PROVE	0:29:19	21:42:28	7:15:27	11:01:26	23:58:31
	Peak memory [GB]	22.9	126.9	63.0	64.0	77.3
D	Mean t. FAIL	0:00:15	0:08:29	0:03:04	0:06:14	0:34:20
	Mean t. PROVE	0:00:30	0:08:22	0:04:35	0:10:05	0:50:50
	Max. t. PROVE	0:00:58	0:19:07	0:09:01	0:15:30	1:27:31
	Peak memory [GB]	4.6	37.6	12.0	20.7	36.7

7.1 Verification runtime

The generation of taint conditions, discovery of multiplexer select signals and the declassification precondition check are implemented as Yosys passes and Python scripts, which complete in less than 5 minutes per verified CPU.

Table 4 summarizes the verification runtimes with CellIFT and CellDFT. For a fair comparison, we verify all RV32I instructions for both versions and report the mean time over all assertion failures, as well as the mean and maximum time over all instructions that were proven. Aggregated runtime and peak memory usage numbers are based on the reports produced by Jasper FPV. The results are for unbounded proofs over unrestricted instruction sequences. All other top level inputs are unconstrained as well, except on Kronos (see below). CellDFT is significantly faster than CellIFT, due to the lower instrumentation cost.

Due parallelization the actual total runtime was much lower than the aggregated one. Initially, for Scarv, the Jasper FPV did not produce results after 24 hours. We then chose Jasper's engine modes M, N, AM and Mp, because they can exchange proof results. For Scarv we also included Ht. For each taint logic state we added a helper assertion stating that the state never gets tainted. The formal tool may use their results as invariants [24]. Together with these assertions, the μCFI property could be proven in the reported time. We extended our tool flow to generate these supportive assertions automatically with a Python script after obtaining the taint state signal names from the Yosys CellIFT pass [89]. For a fair comparison, we add these assertions to all cores. However, they do not always improve the runtime.

7.2 μCFI violations in existing hardware designs

Table 5 shows satisfaction (✓) and violation (✗) of μCFI per CPU and instruction, which we will discuss in detail next. Instructions are grouped per category defined in Section 5. Non-influencing (*ni*) instructions are verified using CellIFT. Control-influencing (*ci*) instructions are verified using CellDFT. *vi* instructions are specified to influence the PC, thus their information flows do not violate μCFI. However, our toolchain is able to check for their operand's information flows and confirms a data flow to the PC using CellDFT.

Most instructions provably satisfy μCFI. We also show CellDFT results for instructions that fail using CellIFT. If proven with CellDFT, these instructions are not causing CF violations. While we are the first to study μCFI violations, Scarv and Ibex (in small configuration) were previously verified for CT violations. Besides confirming known vulnerabilities, we discovered five new vulnerabilities. In the following we discuss our newly discovered security vulnerabilities and reference the corresponding GitHub issues and newly assigned CVE numbers.

Table 5: μCFI results per instruction grouped by instruction category (*ni*, *ci*). μCFI satisfactions are marked with ✓, violations with ✗. Ibex (custom config.) results are for non-secure / secure mode with IUV started in data-independent timing mode and slow multiplier. *ni* instructions that violate μCFI with CellIFT (I) are checked with CellDFT (D) as well. *ci* instructions are only verified with CellDFT. O_k = operand k.

		Instruction	Kronos	PicoRV32	Scarv	Ibex
ni	I	add, and(I), or(I), slli, slt(u), srli, srai, sub, xor(I)	✓	✓	✓	✓
	I	sll, sra, srl	✓	✗	✓	✓
	D	sll, sra, srl	✓	✓	✓	✓
	I	slti(u), addi	✗	✓	✓	✓
	D	slti(u), addi	✗	✓	✓	✓
	I	$\mathbf{O_1}$: div(u), mul(h), mulhsu, mulhu, remu	–	✓	✓	✓/ ✓
	I	$\mathbf{O_1}$: rem	–	✓	✓	✗/ ✗
	D	$\mathbf{O_1}$: rem	–	✓	✓	✓/ ✓
	I	$\mathbf{O_2}$: div(u), mul(h), mulh(s)u, rem(u)	–	✓	✓	✗/ ✗
	D	$\mathbf{O_2}$: div(u), mul(h), mulh(s)u, rem(u)	–	✓	✓	✓
	I	lb(u), sb	✓	✓	✗	✓
	D	lb(u), sb	✓	✓	✓	✓
	I	Scalar crypto	–	–	✓	–
ci	D	lh, lhu, lw, sh, sw	✗	✓	✓	✓
	D	beq, bge(u), blt(u), bne	✓	✓	✓	✗

Listing 1: μCFI violation through data dependency.

```
# load program address into general-purpose register x8
0x800001F8: lui     x8, 0x80000
0x800001FC: addi    x8, x8, 0x400
0x80000200: csrrw x0, mtval,x3       # legal instruction
0x80000204: (il)legal instr          # arbitrary instruction
0x80000208: addi    x6, x8, 0x3050  # μCFI tracks data operand
0x8000020C: (il)legal instr
...
# trap can steer the control flow to attacker-chosen value
0x80000400 <trap_handler>:
0x80000400:      ...
```

7.2.1 Kronos: Kronos does not implement the RISC-V M extensions; therefore, these instructions are marked with '-'.

Kronos: Control-flow hijack (CVE-2024-44927) - Issue 17. If Kronos is integrated with a memory that does not always acknowledge requests in the next clock cycle, there are cases where a 'jalr' or branch instruction reads from the previous instruction's input register. If the previous instruction was operating on attacker-controlled data, an attacker can hijack the CF. If that data was secret, it would be leaked by the data-dependent CF deviation. Therefore, without input constraints, μCFI was violated for all instructions.

Kronos: Control-flow hijack (CVE-2023-51973) - Issue 12. We further verified a setup with 'data_ack' and 'instr_ack' inputs constrained to be always high. Table 4 and Table 5 show results with this setup. These constraints model fault injections on the bus, or a memory that does not comply with the pipelined Wishbone protocol. We discovered a control-flow hijack vulnerability, where a data operand of an *addi, slti* or *sltiu* could be directly copied into the mtvec CSR, which is the machine trap-vector base address. On

RISC-V, by default, all traps are handled in machine mode. Such vulnerabilities allow attackers with only user-mode access to influence the architectural CF in machine mode.

Listing 1 shows assembly code replicating a counterexample returned by Jasper FPV for the *addi* instruction. The two instructions load a program address into GPR x8. Then, a legal *csrrw* instruction is decoded, followed by an arbitrary instruction. Then follows an *addi*, of which the μCFI property tracked the data operand. Due to a CPU bug, an internal CSR write enable signal is high when the *addi* instruction is executed. That signal depends on a CSR decode signal that was earlier set due to the legal CSR instruction. In the example shown, the higher-order bits of the immediate with value 0x305 are interpreted as the address of the *mtvec* CSR. The value stored in x8 gets stored into `mtvec`. A subsequent trap can steer the CF to the value chosen by an attacker. Execution will continue at this address in machine mode. This enables, for example, powerful code-reuse attacks [12, 76, 105]. Constraining the formal tool allows obtaining diverse violation traces. When forbidding CSR instructions, the bug was not revealed, which confirms the necessity for a legal *csrrw* instruction to be used together with the malicious *addi*. UPEC-DIT [34] and ConjunCT [35] could not have found this bug.

Kronos: Constant time violation (CVE-2023-51974) - Issue 13. In the same setup with unexpected bus behavior we further discovered a timing flow from *addi, slti*, and *sltiu* instructions, violating the CT principle if the immediate value matched a performance counter address. The performance counter increase takes one clock cycle longer whenever the upper word of the counter needs an increment.

Furthermore, loads can cause a control-flow violation. If a misaligned load leads to an exception, the faulting address is loaded into the CSR 'mtval' according to the ISA [82]. If it is followed by a *csrrw(i)* and an illegal instruction where the uppper bits match the 'mtval' register address, the faulting address can be transferred into the 'mepc' register due to a bug. A subsequent *mret* would jump to that address. This bug has the same root cause as the already reported issue 15.

When using input assumptions to model one specific scenario where a memory always responds within one clock cycle, all instructions satisfy μCFI on Kronos. If these memory interface assumptions can be proven in the integration setup, Kronos can be trusted to be free from μCFI violations.

7.2.2 Ibex: Table 5 shows results for a custom Ibex configuration with writeback stage and branch predictor enabled (see Appendix C.1 in [25]). On Ibex, multiplications and divisons were known to be non-CT in its small, non-secure configuration [34, 35, 101]. However, previous methods did not discover the CT violations caused by *div/mul*-type instructions that we discuss below, even though they affected the small configuration. Table 5 shows results for a custom Ibex configuration (see Appendix C.1 in [25]). None of the earlier methods tracked data flows; hence, none of them could have detected the control flow violation caused by branches.

Ibex: Data leakage (CVE-2024-28365) - Issue 2144. Ibex reacts to an external memory data error signal, even if it has not started a memory operation. This caused μCFI to fail for *rem*, operand 1. While investigating, we discovered a data leakage to arbitrary architectural registers. The underlying bug in the multiplication and division state-machine was fixed by the maintainers of Ibex.

Ibex: CT violation - Issue 2144. Operand 2 of *div*/*mul*-type instructions that were executed in Ibex' data-independent timing (dit) mode can influence the timing of younger instructions, which are executed after data-independent timing mode is disabled via a *csrrw*. UPEC-DIT [34] and ConjunCT [35] cannot find such violations because a *csrrw* is required to trigger them, which is an instruction that these methods exclude. LeaVe did not scale to the full unconstrained pipeline of Ibex [101]. Thus, these methods cannot reason over instruction sequences that alter the dit-mode configuration. The fix for CVE-2024-28365 also resolved this violation and another where all instructions were affected in the Ibex small configuration in case of unexpected errors on the data bus.

Ibex: Control-flow hijack - Issue 2169. Using CellDFT, we discovered a control-flow violation in Ibex in our custom configuration. The operands of branches can direct the PC to arbitrary values (instead of the branch targets only), which can allow attackers to execute arbitrary code. According to Ibex maintainers, our configuration (writeback stage without branch target ALU) is unsupported. However, this was nowhere documented and thus poses a hidden risk.

7.2.3 PicoRV32 and Scarv: PicoRV32 implements (and documents) data-dependent shift durations in its default configuration. Apart from that, it satisfies μCFI.

Our results confirm a known violation in Scarv, where byte-wise memory accesses leak whether a memory-mapped IO address is accessed [34]. Scarv additionally implements the RISC-V scalar cryptography extension [81], which has not been verified against CT before. We provide results for single and multi-cycle versions in Appendix C in [25]. All cryptography instructions satisfy μCFI.

8 Discussion

Scope of μCFI verification. Because *ci* and *vi* instructions are allowed to influence the PC in some ways, we cannot verify their CT property. However, in the CT threat model this is of little interest, since these instructions are not CT-secure at architectural level. There could be bugs where a CT-violating instruction influences the timing of younger instructions other than through its own variable latency. None of the state-of-the-art CT verification methods [33, 35, 101], including μCFI, can detect these. While irrelevant under CT, these cases could be of interest if software is protected by time-balancing [77]. To ensure the functional correctness of the μCF, functional verification methods [79] for *ci* and *vi* instructions can be applied.

State of functional verification. μCFI verification does not require as precondition that a CPU has undergone complete formal functional verification, which is rarely the case in practice [79, 87]. Naturally, as for any verification method, there remains the possibility that bugs mask other bugs until the CPU has been completely formally verified. For example, because we associate every read with an instruction, if a CPU contains faults in its register reading logic, in the worst case, we could potentially associate a μCFI violation with a wrong instruction. Violations of μCFI on functionally verified designs are either of timing or CF violation nature. If a

Table 6: Verified threat models and automation support in μCFI and related work. We compare our four considered threat models (see Section 4) and support for automation, instruction classification (Instr. class.), and arbitrary instructions (Arb. instrs.).

Method	Threat model				Auto-mation	Instr. class.	Arb. instrs.
	Data leak	CF hijack	CT viol.	Delay inj.			
μCFI	●	●	●	●	◕	●	●
ConjunCT [35]	○	○	●	●	◕	◐	○
LeaVe [101]	○	○	●	●	◐	○	●
UPEC-DIT [33, 34]	○	○	●	●	○	◔	○

CPU has undergone a complete formal functional verification, μCFI violations will always be related to timing.

Verifying larger CPUs. Advanced optimizations (e.g., macro-op fusions, speculative execution, or data-dependent prefetch) could cause μCFI violations if they are value- and not register address-dependent. Developing declassification rules for these optimizations is an interesting direction for future work. Our current implementation does not track data loaded from memory into a GPR. This can be addressed by identifying the memory data ports and tainting incoming data like for GPRs. In the cores we studied, memories were abstracted. Including caches into the verification would require further declassification rules, which we leave for future work, or memory instrumentation [71, 91]. We obtained unbounded proofs on the studied cores. However, for larger cores, typical abstraction techniques will be required [86], which opens future research directions for applying μCFI.

9 Related Work

Table 6 compares different aspects of μCFI with related work: **Threat model:** Related methods focus on timing varieties only [35, 40, 97], explicitly disregarding data flows [33, 34, 101]. UPEC-DIT and ConjunCT determine a set of instructions that produce CT-secure programs when only these instructions are combined [33, 34, 35]. **Automation:** LeaVe [101] and UPEC-DIT [33] require manual extraction of design conditions and specification of candidate invariants to exclude known or discovered (or in case of UPEC also spurious) violations. While LeaVe can manually declassify data used by a CT-violating instruction via contract specification, it is not able to generate these non-trivial contract statements automatically [101]. UPEC needs the verification engineer to decide about the validity of information flows in multiple iterations before obtaining a proof [33, 34]. ConjunCT [35] learns invariants and requires only the annotation of a few signals, in the same order as μCFI. **Instruction classification:** LeaVe [101] can precisely find the root cause of a violation, but requires manual relation of the resulting design condition to instructions. UPEC-DIT [33] and ConjunCT [35] provide results per instruction, but with limitations. UPEC-DIT excludes violating instructions from the proof [33], leading to a reduced threat model. ConjunCT may misclassify instructions in certain cases [35]. μCFI precisely relates a violation to an instruction by design. **Arbitrary instructions:** UPEC-DIT and ConjunCT cannot detect cases where CT instructions interact with non-CT instructions, because the non-CT instructions need to be removed from the proof [33, 35]. Most importantly, these two methods cannot detect cases where an instruction that is CT in itself influences the timing of a younger instruction. For example, in Kronos we found that an *add* instruction that satisfies μCFI on its own can cause a μCFI violation only if it is followed by a *csrrw*. Using an approach like ConjunCT or UPEC-DIT, the *csrrw* would have to be excluded from the verified instructions due to its own influence on the PC and therefore the violation would not be detected. While LeaVe can detect such case (related to CT), it requires manual association of the result to a specific instruction. LeaVe declassifies CT violations by adding relational constraints to the contract. μCFI leverages a property of CellIFT's taint tracking logic that allows the individual verification of instructions (see A.1 in [25]). Therefore, the instruction sequence does not need to be restricted for both methods. [36] verifies the CT property of a small custom CPU without forwarding.

Verification automation. Zeng et al. [112] proposed a method to generate update functions from an RTL design. This function captures the data a signal is updated to, whereas we capture the condition under which it is updated. Some RISC-V cores implement the riscv-formal interface [109]. Being an observational circuit, it cannot be used for taint injection. Furthermore, the connection of signals to this interface is a manual effort and most signals are bulk updated in the last pipeline stage [109], making it difficult to check intermediate states. Borkar et al. [13] propose a fuzzer for finding timing violations. Being a simulation-based method, it cannot provide guarantees of absences.

IFT verification. Kastner et al. [32] presented a tool to mine information flow properties from simulation traces to test them with a proprietary tool. Information flow properties that are specified allow taint flows under certain conditions. The method we propose can be extended in the same manner. Since the condition would always be true in case of the μCFI property, we let our IFT property unconditioned. Ardeshiricham et al. [5] proposed a timing flow logic implemented in a proprietary tool. Such a logic could be used to distinguish between timing and control dependencies.

10 Conclusion

We presented the μCFI property that ensures that instructions are data oblivious and do not influence the control flow unless explicitly specified by the ISA. We developed the first automated and reusable method for formally verifying μCFI based on detecting information flows from instruction's operands to the PC. Our taint injection and flow declassification mechanisms allow us to verify interactions between arbitrary instruction combinations in unbounded sequences, where each of them may operate on public, secret or attacker-controlled data. Leveraging taint logic based IFT verification of the entire pipeline, μCFI verification requires tagging only a few signals and can associate the violation of μCFI to an offending instruction, simplifying verification and triaging. Applying μCFI verification to four in-order RISC-V cores, we proved the μCFI property in many cases and found five new security vulnerabilities, including in previously verified CPUs.

Ethical considerations. We have reported the property violations to the maintainers of the respective repositories.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback, Patrick Jattke for his help in formatting the paper, and the maintainers of the designs we verified for their support

in understanding and fixing some of the bugs. This work was supported by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM TOPS '09*, 13, 1, Article 4, (Nov. 2009). https://doi.org/10.1145/1609956.1609960.

[2] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. [n. d.] Verifying Constant-Time Implementations. In *USENIX Security 2016*.

[3] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, and Tiago Oliveira. [n. d.] Certified compilation for cryptography: Extended x86 instructions and constant-time verification. In *INDOCRYPT 2020*. Springer.

[4] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. [n. d.] Towards verified, constant-time floating point operations. In *ACM SIGSAC 2018*.

[5] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. [n. d.] Clepsydra: Modeling timing flows in hardware designs. In *IEEE/ACM ICCAD 2017*.

[6] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. [n. d.] Register transfer level information flow tracking for provably secure hardware design. In *IEEE DATE 2017*.

[7] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. [n. d.] System-level non-interference for constant-time cryptography. In *ACM SIGSAC 2014*.

[8] Gilles Barthe, Sandrine Blazy, Benjamin Gregoire, Remi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. [n. d.] Formal verification of a constant-time preserving C compiler. In *ACM POPL 2019*.

[9] Mohammad Behnia et al. 2021. Speculative interference attacks: breaking invisible speculation schemes. In *ASPLOS 2022*.

[10] Emery D. Berger and Benjamin G. Zorn. [n. d.] DieHard: Probabilistic memory safety for unsafe languages. In *ACM PLDI '06*.

[11] Sandrine Blazy, David Pichardie, and Alix Trieu. 2019. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, 27.

[12] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. [n. d.] Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS 2011*.

[13] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. WhisperFuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors. ().

[14] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. [n. d.] Provably-Safe Multilingual Software Sandboxing Using Webassembly. In *USENIX Security 2022*.

[15] Aaron R. Bradley. 2011. Sat-based model checking without unrolling. In Springer Berlin Heidelberg.

[16] N. Bruns, V. Herdt, D. Große, and R. Drechsler. [n. d.] Efficient cross-level processor verification using coverage-guided fuzzing. In *VLSI 2022*.

[17] N. Bruns, V. Herdt, E. Jentzsch, and R. Drechsler. [n. d.] Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging. In *DATE 2022*.

[18] [n. d.] Cadence jasper formal property verification (fpv) app. Accessed: 2024-07-10. 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.

[19] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi. [n. d.] ProcessorFuzz: Processor fuzzing with control and status registers guidance. In *HOST 2023*.

[20] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. [n. d.] Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security 2015*.

[21] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. [n. d.] Constant-time foundations for the new spectre era. In *ACM SIGPLAN 2020*.

[22] Katharina Ceesay-Seitz, Hamza Boukabache, and Daniel Perrin. 2020. A functional verification methodology for highly parametrizable, continuously operating safety-critical fpga designs: applied to the cern radiation monitoring electronics (crome). In *Computer Safety, Reliability, and Security*. António Casimiro, Frank Ortmeier, Friedemann Bitsch, and Pedro Ferreira, (Eds.) Springer International Publishing, Cham, 67–81. ISBN: 978-3-030-54549-9.

[23] Katharina Ceesay-Seitz, Hamza Boukabache, and Daniel Perrin. [n. d.] Semi-formal reformulation of requirements for formal property verification. In *accellera DVCON EUROPE 2019*.

[24] Katharina Ceesay-Seitz, Sarath Kundumattathil Mohanan, Hamza Boukabache, and Daniel Perrin. [n. d.] Formal property verification of the digital section of an ultra-low current digitizer asic. In *accellera DVCON EUROPE 2021*.

[25] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μCFI: formal verification of microarchitectural control-flow integrity. In *Extended version of this paper*. https://comsec.ethz.ch/wp-content/files/mucfi_ccs24.pdf.

[26] C. Chen, R. Kande, N. Nyugen, F. Andersen, A. Tyagi, A. R. Sadeghi, and J. Rajendran. HyPFuzz: Formal-assisted processor fuzzing. ().

[27] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. [n. d.] Code-Level Model Checking in the Software Development Workflow. In *ACM/IEEE ICSE 2020* (Icse-Seip 2020).

[28] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. 2018. *Handbook of Model Checking*. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, (Eds.) Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-10575-8.

[29] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security*, 18, 6.

[30] John Criswell, Nathan Dautenhahn, and Vikram Adve. [n. d.] KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE S&P 2014*.

[31] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. [n. d.] Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *IEEE S&P 2020*.

[32] Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. [n. d.] Isadora: Automated information flow property generation for hardware designs. In *ACM ASHES 2021*.

[33] Lucas Deutschmann, Johannes Müller, Mohammad R. Fadiheh, Dominik Stoffel, and Wolfgang Kunz. 2024. A scalable formal verification methodology for data-oblivious hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1–1. DOI: 10.1109/TCAD.2024.3374249.

[34] Lucas Deutschmann, Johannes Müller, Mohammad R. Fadiheh, Dominik Stoffel, and Wolfgang Kunz. [n. d.] Towards a formally verified hardware root-of-trust for data-oblivious computing. In *ACM/IEEE DAC 2022*.

[35] S. Dinesh, M. Parthasarathy, and C. Fletcher. [n. d.] ConjunCT: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks. In *IEEE S&P 2024*.

[36] Ning Dong, Roberto Guanciale, Mads Dam, and Andreas Lööw. [n. d.] Formal verification of correctness and information flow security for an in-order pipelined processor. In *FMCAD 2023*.

[37] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. [n. d.] Holistic control-flow protection on real-time embedded systems with kage. In *USENIX Security 2022*.

[38] Limor Fix. 2008. Fifteen Years of Formal Property Verification in Intel. In *25 Years of Model Checking: History, Achievements, Perspectives*. Orna Grumberg and Helmut Veith, (Eds.) Springer Berlin Heidelberg, 139–144. https://doi.org/10.1007/978-3-540-69850-0_8.

[39] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. [n. d.] A security RISC: Microarchitectural attacks on hardware RISC-v cpus. In *IEEE S&P 2023*.

[40] Klaus v Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. [n. d.] IODINE: Verifying Constant-Time Execution of Hardware. In *USENIX Security 2019*.

[41] Aman Goel and Karem Sakallah. [n. d.] Model checking of verilog rtl using ic3 with syntax-guided abstraction. In *NFM 2019*. Springer.

[42] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. [n. d.] ASLR on the line: Practical cache attacks on the MMU. In *NDSS 2017*.

[43] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. [n. d.] Hardware-software contracts for secure speculation. In *IEEE S&P 2021*.

[44] Lorenz Hetterich and Michael Schwarz. [n. d.] Branch different-spectre attacks on apple silicon. In *DIMVA 2022*. Springer.

[45] Karine Heydemann, Jean-François Lalande, and Pascal Berthome. 2019. Formally verified software countermeasures for control-flow integrity of smart card C code. *Computers & Security*, 85.

[46] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware information flow tracking. *ACM Comput. Surv.*, 54, 4, Article 83, (May 2021), 39 pages. DOI: 10.1145/3447867.

[47] Wei Hu, Lingjuan Wu, Yu Tai, Jing Tan, and Jiliang Zhang. 2020. A unified formal model for proving security and reliability properties. In *2020 IEEE 29th Asian Test Symposium (ATS)*, 1–6. DOI: 10.1109/ATS49688.2020.9301533.

[48] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. [n. d.] Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE S&P 2021*.

[49] [n. d.] Ibex CPU. Accessed: 2024-04-16. https://github.com/lowRISC/opentitan.

[50] [n. d.] Ibex documentation - verification stages. Accessed: 2024-04-16. https://ibex-core.readthedocs.io/en/latest/03_reference/verification_stages.html.

[51] 2018. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*.

[52] Intel Corporation. [n. d.] Data Operand Independent Timing ISA Guidance. [Updated 13-February-2023]. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[53] Jan Jancar, Marcel Fourne, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. [n. d.] "They're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *IEEE S&P 2022*.

[54] N. Kabylkas, T. Thorn, S. Srinath, P. Xekalakis, and J. Renau. [n. d.] Effective processor verification with logic fuzzer enhanced co-simulation. In *IEEE/ACM MICRO 2021*.

[55] R. Kande, A. Crump, G. Persyn, P. Jauernig, A. R. Sadeghi, A. Tyagi, and J. Rajendran. [n. d.] TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *USENIX Security 2022*.

[56] Paul Kocher et al. [n. d.] Spectre attacks: Exploiting speculative execution. In *IEEE S&P 2019*.

[57] [n. d.] Kronos RISC-v (cascade fixes integrated). Accessed: 2024-07-10. https://github.com/cascade-artifacts-designs/cascade-kronos.

[58] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. [n. d.] NetCAT: Practical cache attacks from the network. In *IEEE S&P 2020*.

[59] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen. [n. d.] RFUZZ: Coverage-directed fuzz testing of RTL on fpgas. In *ICCAD 2018*.

[60] Moritz Lipp et al. [n. d.] Meltdown: Reading kernel memory from user space. In *USENIX Security 2018*.

[61] Chang Liu, Michael Hicks, and Elaine Shi. [n. d.] Memory Trace Oblivious Program Execution. In *IEEE CSF 2013*.

[62] Arthur Costa Lopes and Diego F Aranha. 2017. Benchmarking tools for verification of constant-time execution. *SBSEG. Bento Goncalves, Brazil: SBC*.

[63] lowRISC contributors. [n. d.] OpenTitan root of trust. Accessed: 2024-07-10. https://opentitan.org/.

[64] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. [n. d.] Bypassing memory safety mechanisms through speculative control flow hijacks. In *IEEE EuroS&P 2021*.

[65] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazieres. [n. d.] CCFI: Cryptographically enforced control flow integrity. In *ACM SIGSAC 2015*.

[66] K. L. McMillan. [n. d.] Interpolation and sat-based model checking. In *CAV 2003*.

[67] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. MemJam: a false dependency attack against constant-time crypto implementations. *IJPP 2019*, 47, 4, (Aug. 2019), 538–570.

[68] Daniel Moghimi. [n. d.] Downfall: Exploiting speculative data gathering. In *USENIX Security 2023*.

[69] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. [n. d.] The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *USENIX Security 2005*.

[70] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. [n. d.] Axiomatic hardware-software contracts for security. In *ACM ISCA 2022*.

[71] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2014. Leveraging gate-level properties to identify hardware timing channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33, 9, 1288–1301.

[72] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. [n. d.] Revizor: Testing black-box CPUs against speculation contracts. In *ASPLOS 2022*.

[73] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. [n. d.] AutoSVA: Democratizing formal verification of RTL module interactions. In *ACM/IEEE DAC 2021*.

[74] Marcelo Orenes-Vera, Hyunsung Yun, Nils Wistoff, Gernot Heiser, Luca Benini, David Wentzlaff, and Margaret Martonosi. [n. d.] AutoCC: Automatic discovery of covert channels in time-shared hardware. *MICRO 2023*, 51.

[75] [n. d.] PicoRV32 - a size-optimized RISC-V CPU. Accessed: 2024-07-10. https://github.com/YosysHQ/picorv32.

[76] Marco Prandini and Marco Ramilli. [n. d.] Return-oriented programming. In *IEEE S&P 2012*. Vol. 10.

[77] Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. [n. d.] Dejitleak: eliminating jit-induced timing side-channel leaks. In (ACM ESEC/FSE 2022).

[78] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. [n. d.] Crosstalk: Speculative data leaks across cores are real. In *IEEE S&P 2021*.

[79] Alastair Reid et al. [n. d.] End-to-end verification of processors with ISA-formal. In *CAV 2016*.

[80] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. 2022. Verica - verification of combined attacks: automated formal verification of security against simultaneous information leakage and tampering. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022, 4, (Aug. 2022), 255–284. doi: 10.46586/tches.v2022.i4.255-284.

[81] RISC-V. 2024. RISC-v cryptography extension. Accessed: 2024-09-23. https://github.com/riscv/riscv-isa-manual/releases/tag/20240411.

[82] RISC-V. 2024. Risc-v isa specifications. Accessed: 2024-09-23. https://riscv.org/technical/specifications/.

[83] Grigore Roşu, Wolfram Schulte, and Traian Florin Şerbănuţă. [n. d.] Runtime verification of C memory safety. In *RV '2009*. Springer.

[84] Sarwar Sayeed, Hector Marco-Gisbert, Ismael Ripoll, and Miriam Birch. 2019. Control-flow integrity: attacks and protections. *Applied Sciences*, 9, 20.

[85] [n. d.] SCARV: processor core implementation. Accessed: 2024-07-10. https://github.com/scarv/scarv-cpu.

[86] Erik Seligman, Tom Schubert, and Achutha M. V. Kiran Kumar. 2023. *Formal Verification. An Essential Toolkit for Modern VLSI Design*. (2nd ed.). Elsvier Science & Technology.

[87] Siemens AG. [n. d.] The 2022 Wilson Research Group Functional Verification Study. Accessed: 2024-4-28. https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/.

[88] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. [n. d.] Cascade: CPU Fuzzing via Intricate Program Generation. In *USENIX Security 2024*.

[89] Flavien Solt, Ben Gras, and Kaveh Razavi. [n. d.] CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL. In *USENIX Security 2022*.

[90] Flavien Solt, Patrick Jattke, and Kaveh Razavi. [n. d.] RemembERR: Leveraging microprocessor errata for design testing and validation. In *IEEE/ACM MICRO 2022*.

[91] Flavien Solt and Kaveh Razavi. [n. d.] Hybridift: scalable memory-aware dynamic information flow tracking for hardware. *ICCAD 2024*.

[92] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39, 11.

[93] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Ulfar Erlingsson, Luis Lozano, and Geoff Pike. [n. d.] Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security 2014*.

[94] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. [n. d.] Complete information flow tracking from the gates up. In *ASPLOS 2019*.

[95] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks. [n. d.] Fuzzing Hardware Like Software. In *USENIX Security 2022*.

[96] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. [n. d.] INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution. In *USENIX Security 2023*.

[97] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. [n. d.] Solver-aided constant-time hardware verification. In *ACM SIGSAC 2021*.

[98] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. [n. d.] RIDL: Rogue in-flight data load. In *IEEE S&P 2019*.

[99] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. [n. d.] Augury: Using data memory-dependent prefetchers to leak data at rest. In *IEEE S&P 2022*.

[100] Huibo Wang et al. [n. d.] Towards memory safe enclave programming with rust-sgx. In *ACM SIGSAC 2019*.

[101] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. [n. d.] Specification and verification of side-channel security for open-source processors via leakage contracts. In *ACM SIGSAC 2023*. Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, (Eds.) ACM, 2128–2142. https://doi.org/10.1145/3576915.3623192.

[102] Tobias Wiersema, Stephanie Drzevitzky, and Marco Platzner. [n. d.] Memory Security in Reconfigurable Computers: Combining Formal Verification with Monitoring. In *IEEE FPT 2014*.

[103] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. [n. d.] Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks. In *WOOT 2022*.

[104] Johannes Wikner and Kaveh Razavi. [n. d.] RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security 2022*.

[105] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. [n. d.] Phantom: Exploiting decoder-detectable mispredictions. *MICRO 2023*.

[106] Clifford Wolf, Johann Glaser, and Johannes Kepler. [n. d.] Yosys-a free Verilog synthesis suite. In *Austrochip 2013*.

[107] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. [n. d.] Confirm: Evaluating Compatibility and Relevance of Control-Flow Integrity Protections for Modern Software. In *USENIX Security 2019*.

[108] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. [n. d.] In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *USENIX Security 2022*.

[109] Yosys Open SYnthesis Suite. [n. d.] RISC-V Formal Verification Framework. Accessed: 2024-07-10. https://github.com/YosysHQ/riscv-formal.

[110] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. [n. d.] All your PC are belong to us: Exploiting non-control-transfer instruction BTB updates for dynamic PC extraction. In *ACM ISCA 2023*.
[111] Bin Zeng, Gang Tan, and Greg Morrisett. [n. d.] Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM CCS 2011*.
[112] Yu Zeng, Aarti Gupta, and Sharad Malik. [n. d.] Automatic generation of architecture-level models from RTL designs for processors and accelerators. In *IEEE DATE 2022*.
[113] Anthony Zgheib, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Max Dutertre. [n. d.] A CFI verification system based on the RISC-v instruction trace encoder. In *IEEE DSD 2022*, 456–463.
[114] Qizhi Zhang, Jiaji He, Yiqiang Zhao, and Xiaolong Guo. 2020. A formal framework for gate- level information leakage using z3. In *AsianHOST*, 1–6. DOI: 10.1109/AsianHOST51057.2020.9358257.

A Information Flow Tracking Property Details

A.1 Combining taints

We now prove that it suffices to verify taint flows individually per input bit value to capture also all taint flows possible in the presence of any combination with other input taints. In particular, we prove that every taint flow can be reduced to a single input taint value. This allows us to verify taint flows per instruction and input sequence in isolation, while guaranteeing that no taint flows are missed. Intuitively, the proof states that taint never removes taint from other bits.

In the following, we prove the property P_n defined in Equation 7 of the set C_n of cells with n input bits and one output bit. Note that a cell with m output bits can be transformed into m cells with one output bit each without loss of generality. We use the notations introduced by CellIFT [89]. I is a set of input bits i, I^t is a set of tainted input bits i^t, and H is the hamming weight function.

Theorem A.1.

$$P_n : \forall C \in C_n \left[\left(\exists I, I^t \mid C(I, I^t)^t \right) \implies \left(\exists I, I^t \mid C(I, I^t)^t \wedge \left[H(I^t) = 1 \right] \right) \right] \quad (7)$$

We prove P_n by induction on n. The base case P_1 holds because at least one bit of the input must be tainted for the output to be tainted, hence $\tilde{I} := I$ and $\tilde{I^t} := I^t$ is a satisfying assignment. Now let $n > 1$ and assume that P_{n-1} holds. Let's consider a cell $C \in C_n$, and let $I := i_0, \cdots, i_{n-1}$. Assume that there exist I and I^t such that $C(I, I^t)^t$. Additionally consider the cells $C^0 \in C_{n-1}$ and $C^1 \in C_{n-1}$ defined as C with i_{n-1} respectively fixed to 0 and 1 and $i^t_{n-1} = 0$. We distinguish three cases. (a) If there is an I such that $i^t = \mathbb{1}_{t=n-1}$ and $C(I, i^t)^t = 1$, then P_n holds for C. (b) Else, if there exist I and I^t such that $i^t_{n-1} = 0$ such that $C(I, I^t)^t$, then by P_{n-1}, P_n holds for C. (c) Else, we know that there is no I and I^t such that $C(I, I^t)^t \wedge H(I^t) = 1$. We show that this case can only happen for a constant cell. If there existed I and $\tilde{I}$ such that $i_{n-1} = \tilde{i_{n-1}} \wedge C^{i_{n-1}}(I) \neq C^{i_{n-1}}(\tilde{I})$, then information flows through $C^{i_{n-1}}$, hence case (b) would apply. Therefore, we know that the value of C only depends on i_{n-1}. Additionally, because (c) excludes (a), information never flows exclusively form i^t_{n-1} to the cell's output. Given that the other input bits of the cell do not influence its output (and hence, no information flows), we conclude that C is a constant cell, which contradicts the existence of I and I^t such that $C(I, I^t)^t$, hence case (c) cannot happen. We proved that P_n holds for all $n > 0$.

We hence conclude that when finding a correct assignment of non-tainted inputs, which is provided by the formal verification engine, there exists a single input bit that can be tainted to taint the output bit.

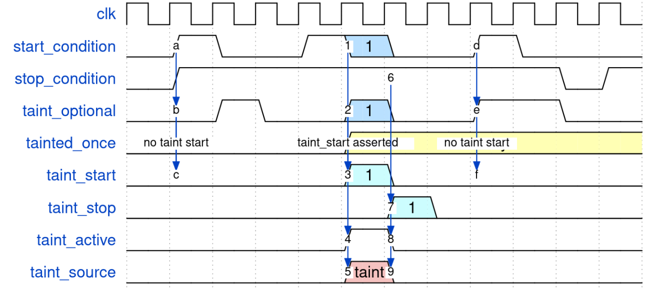

Figure 11: Taint injection example.

A.2 Generic taint injection assumptions

In the following, we describe our generic taint injection assumptions that act as input constraints. They are placed inside a SystemVerilog checker, together with the μCFI property that checks for taint flows to the PC. The checker takes three parameters: a taint start and stop condition and the taint source. For each instruction and operand we instantiate one checker in a separate task. Within this task, the operand signal's corresponding taint signal is abstracted and controlled by the assumptions. The taint start condition is a conjunction of the generated CPU-specific register reading condition (per operand) and an instruction type check (performed on the sampled instruction word). The taint stop condition is true whenever the register reading signal can receive new data. See Appendix A.3 for details.

Figure 11 shows an example wave form that we will use to explain the signals involved in taint injection. Signals *start_condition* and *stop_condition* are the input parameters described above. Taint injection starts, when the *taint_start* signal rises. For being able to precisely attribute a counter example to a specific instruction within a sequence, we allow only one instruction per sequence to read tainted data. Appendix A.1 proved that taint injection per single instruction generalizes to sequences where multiple instructions read tainted data. This is controlled by the *tainted_once* signal that remains high forever as soon as taint was injected once. If we let the *taint_start* signal rise whenever the *start_condition* is true, it would always rise whenever the *start_condition* is true for the fist time. The *taint_once* signal would then prevent that *taint_start* rises again, which means we would only verify taint injection on the first possible occurrence of the currently verified instruction type within any sequence. Even though a this first occurrence may happen at any possible clock cycle in which new instructions can read from the register file, there are cases that we would miss to verify: If the same instruction type appears multiple times within one sequence, we would only verify μCFI for the first occurrence. To be exhaustive, while tainting only one instruction at a time, we introduce another signal: *taint_optional*. This signal is a free signal, which means it is not driven by any logic, and treated by the formal tool like an unconstrained input. The *taint_start* is constrained to only rise when the *start_condition* is true, *tainted_once* is not yet high, and *taint_optional* is true. As for all inputs, the formal tool will consider all possible input sequences to *taint_optional*. This means, every sequence in which in some clock cycles the *start_condition* is true and *tainted_once* is low at the same time, will be verified with *taint_optional* being 0 and 1 at these times. Thus, like shown in Figure 11, taint is not injected on every first occurrence of a

start_condition (a->c). It is injected if all three conditions are met (1->5), and not injected if *tainted_once* is already high (d->f).

Another free signal, called *taint_active*, is controlled via an assumption that states that *taint_active* shall be high starting in any clock cycle in which *taint_start* is high and no taint check has been started before, until one clock cycle before the *taint_stop* signal becomes high. The *taint_stop* signal is assigned in the same clock cycle in which the stop condition is true and only if the taint was active in the previous clock cycle. Taint injection is stopped before the *stop_condition* is true, because the *stop_condition* equals the register reading signals 'update condition' (see Appendix A.3), which is true whenever the register reading signal reads new data. This new read may already belong to the next instruction. (If it does not, a sequence in which *taint_start* is high in this clock cycle is considered as well by the formal tool).

One assumption controls that *taint_active* remains low before *taint_start* and another one forces *taint_active* low until *taint_start* would rise again (if we would remove the *tainted_once* signal).

The SystemVerilog checker that contains the taint injection assumptions takes the data source signals' taint signal as untyped input. One assumption forces this signal to 0 iff *taint_active* is low. Another assumption treats the taint source as signed signal and forces it to -1 iff *taint_active* is high. That way, leveraging SystemVerilog's sign extension, we force all taint bits to 1, regardless of the actual bit width of the taint source.

A.3 Taint injection condition extraction

In Section 6, we discussed the taint start and stop conditions needed for creating the μCFI property. Algorithm 1 presents a novel method at cell level to automatically construct the Update Condition (UC) of a given signal by traversing back its driving logic. We define the UC as the condition in which a signal is updated with another value than its own previous value. The Yosys pass finds the provided signal name in the flattened design and traverses its driving logic backwards. The pass or-connects all conditions in which a signal can be assigned. It constructs the UC by and-ing control signals (enable and multiplexer selects) with 0 or 1, depending on whether a path leads to a self-assignment or not. The taint stop condition is the generated UC of O_k.

A similar algorithm constructs the condition under which a signal is assigned with values from specific other signals (provided as parameters to the Yosys pass). This version returns 1 for paths in which the signal is assigned with the data from the specified signals, and 0 otherwise. We use this second option to construct the condition under which the register address signal is read from the instruction word. The formal setup then samples the instruction word whenever this condition is true. The taint start condition is obtained by generating the condition in which the register reading signal (O_k) reads from the register file. In the formal setup it is conjoined with the sampled instruction word. This is important to synchronize the start of taint injection with the register read of the currently examined instruction.

We implemented the algorithm as a Yosys pass that creates a new update signal for some given input signal and adds it to the design. Then, we reconstruct the UC from Yosys' HDL output with a Python script that traverses back the UC logic and replaces all Yosys-generated itermediate signals with their assigned expressions until ultimately experessions are obtained that consist of orignal design signal values only. It is also possible to directly use the generated signal in the formal testbench, if the pass is executed in the same Yosys flow as the CellIFT/CellDFT pass.

Algorithm 1 (Update condition). Automatically constructs the Update Condition (UC) of a given signal.

```
procedure UC(signal)
    if IS_MODULE_INPUT(signal)||IS_CONST(signal) then
        return 1
    else if ASSIGNED_WITH_WIRE(signal) then
        return UC(assignee)
    else if ASSIGNED_WITH_STATE_CELL(signal) then
        return $past(UC(assignee))
    else if ASSIGNED_WITH_STATE_EN_CELL(signal) then
        return $past(enable&&UC(assignee))
    else if ASSIGNED_WITH_MUX_CELL(signal) then
        return !s&&UC(mux_in_a)||s&&UC(mux_in_b)
    else if ASSIGNED_WITH_LOGIC_CELL(signal) then
        return 1
    end if
end procedure
```

```
assert property: (##1 $changed(O_k)
  |-> UC(O_k));
```

Listing 2: Update condition sanity check.

The property in Listing 2 guarantees that we have correctly constructed the UC, basically serving as a sanity check of our pass' output. We successfully proved this property per operand and CPU.

B CellIFT Covers IFT Over Sequential circuits

In this Appendix, we show that CellIFT covers information flows. At cell level, Theorem 6.1 matches exactly the definition of information flow tracking introduced and covered by CellIFT [89]. Beyond the single-cell case we proceed by induction on the maximal number d (depth) of cells on the longest path between the input and a given output bit. Theorem 6.1 establishes the base case $l = 1$. Let's consider a circuit C of depth $l > 1$. Assume that there is an information flow from some input to some output of C. We take a minimal subset of the input bits SRC such that removing one more bit would vanish the information flow. There is an information flow through the first layer: each output bit of the first layer is influenced by at least one input bit of SRC. By Theorem 6.1, the taint of the first layer is propagated to the second layer. By the induction hypothesis for depth $l - 1$, the taint of the second layer is propagated to the output. Finally, the equivalence between CellIFT and the information flow operator holds in time, as the taint as well as the information flow is propagated with one cycle latency through stateful elements (flip-flops and latches). Hence, any information flow in space and time will be captured by CellIFT.

C Additional Results

C.1 Configurations

Runtime and design statistics were shown for PicoRV32 with the following deviations from default configurations: PicoRV32 with ENABLE_MUL=1, ENABLE_DIV=1, COMPRESSED=1.

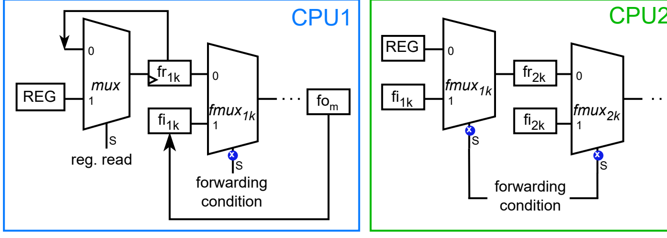

Figure 12: CPU1: Temporally incorrect $fMux_{l_k}$ declassification. CPU2: Correct $fMux_{l_k}$ declassification.

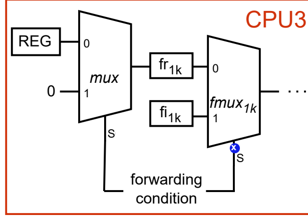

Figure 13: Incorrect $fMux_{l_k}$ declassification.

Our custom Ibex configuration is as follows: PMPEnable=0, PMPGranularity=0, PMPNumRegions=4, MHPMCounterNum=0, MHPMCounterWidth=40, RV32E=0, RV32M=RV32MSlow, RV32B=RV32BNone, RegFile=RegFileFF, BranchTargetALU=0, WritebackStage=1, ICache=0, ICacheECC=0, BranchPredictor=1, DbgTriggerEn=0, DbgHwBreakNum=1, SecureIbex=1, ICacheScramble=0, RndCnstLfsrSeed=RndCnstLfsrSeedDefault, RndCnstLfsrPerm=RndCnstLfsrPermDefault, DmHaltAddr=32'h1A110800, DmExceptionAddr=32'h1A110808, Lockstep=0, ResetAll=1

C.2 Scarv

The Scarv processor implements the RISC-V scalar cryptography extensions [81] of which we verified the μCFI property for the following instructions in both the single- and multi-cycle configuration: AES64DSM, SHA512SIG1H, AES32ESMI, SHA512SUM0R, SM3P0, AES32DSI, AES32DSMI, AES32ESI, AES64KS2, AES64KS1I, AES64DS, SHA512SUM0, SHA256SUM1, SHA256SUM0, AES64ESM, SHA256SIG0. All of them satisfy μCFI.

D Additional Proofs

D.1 Sound and unsound $fMux_{l_k}$ select abstraction

We first discuss some examples of designs where the forwarding abstraction is sound, and some where it is not. The abstraction is sound if register taint reaches the $fMux_{l_k}$ in case of forwarding. For example, in Figure 12, in CPU 2, taint always reaches frd_{2_k}, because the only cells that could block taint propagation from the register file REG are $fMux_{l_k}$ with abstracted select signals. However, in CPU 1, taint injection may have stopped when the forwarding condition is true, because frd_{1_k} was overwritten with the untainted data of the next instruction (if reg. read was true one clock cycle before the forwarding condition was true). In this case, fr_{l_k} would equal O_k (because it is the first signal through which register data passes) and so we would see a potential μCFI violation when verifying any next instruction and obtain false positives, but no false negatives. However, in Figure 13, fr_{l_k} would be connected to constant 0 in the forwarding case and therefore, declassification with $fMux_{l_k}$ select abstraction would be incorrect, as for any input sequence A,B: $\forall n : fdi^A_{l_k,n} = \text{frd}^A_{l_k,n} = fdi^B_{l_k,n} = \text{frd}^B_{l_k,n}$.

Proving the declassification precondition for forwarding paths. The constraint expressed in Equation 6 states that taint must reach fr_{l_k} whenever IUV would receive data from fi_{l_k}. This is satisfied if we can prove that O_k always reaches fr_{l_k}, independent of the forwarding condition. The declassification precondition can be fulfilled by disconnecting fi_{l_k} from its driving logic and either constrain it to allow arbitrary values when an instruction can read from it, or we ensure that O_k always reaches fr_{l_k} and disconnect the driving logic of the corresponding forwarding multiplexer $fMux_{l_k}$'s select signal and leave it unconstrained. The declassification precondition can be proven on a design as follows: If there exists a case where the register data does not reach the forwarding multiplexer, there must exist some logic cell on the path between the register file and the fi_{l_k} that blocks the information flow. Therefore, if we can show that no other gates exists between the register file and fr_{l_k}, then the register data is guaranteed to reach the forwarding multiplexer independently of the microarchitecural state. Thus, for any $fMux_{l_k}$ of which fr_{l_k} is an unconditional and undelayed assignment from the register file the declassification precondition is trivially fulfilled.

Otherwise, the only cells between the register file output (REG) and fr_{l_k} that would not block the flow are either: (a) cells that select between new data from REG and fr_{l_k} or (b) other forwarding multiplexers that fulfill the declassification precondition. Our Yosys pass for finding forwarding multiplexers verifies via static design analysis that the verified designs fulfill these structural patterns. The checks are true for all the CPUs that we verified. If a CPU does not fulfill these patterns, our declassification method can be used if one finds another way to prove that taint reaches the $fMux_{l_k}$ in the forwarding case.