MileSan: Detecting Exploitable Microarchitectural Leakage via Differential Hardware-Software Taint Tracking

Tobias Kovats
tkovats@ethz.ch
ETH Zurich
Zurich, Switzerland

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

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

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

Abstract

Microarchitectural performance optimizations introduce information flows inside CPU implementations that exceed those defined by the Instruction Set Architecture (ISA). Microarchitectural vulnerabilities, such as constant-time violations and various classes of transient execution attacks, are subsets of these excessive information flows. We observe that an *exploitable microarchitectural leakage* is an excessive information flow that can affect the time it takes for the CPU to execute a particular instruction, creating a timing covert channel. We design MileSan, the first RTL sanitizer that is capable of detecting exploitable microarchitectural leakage by checking for the architecturally-observable differences between architectural and microarchitectural information flows. For a given program and CPU implementation, MileSan computes architectural flows using software taint tracking and microarchitectural flows using RTL taint tracking. Evaluating the exploitability of proof of concepts generated by previous microarchitectural fuzzers, we find cases that are in fact not exploitable and discover the particular microarchitectural components that enable exploitation for the rest.

In addition to assessing exploitability, MileSan enables the generation of random test programs with strictly-defined architectural information flows of secret data using a novel technique called taint-aware in-situ simulation. Leveraging this capability, we build RandOS, a new microarchitectural fuzzer that generates random programs traversing different privilege levels and address spaces, akin to *random operating systems*. Evaluation using five RISC-V CPUs shows that RandOS not only detects known exploitable vulnerabilities 4.5× faster than the state of the art, but also discovers 19 new constant-time violations and transient execution vulnerabilities in well-tested CPUs, such as BOOM, CVA6 and OpenC910.

CCS Concepts

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

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765066

Keywords

Hardware security; RTL fuzzing; side-channels

ACM Reference Format:
Tobias Kovats, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2025. MileSan: Detecting Exploitable Microarchitectural Leakage via Differential Hardware-Software Taint Tracking. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765066

1 Introduction

Extensive previous work has shown that processors are affected by microarchitectural vulnerabilities that can be exploited to leak confidential information [1, 3, 6, 8, 9, 28, 30, 32, 34–36, 39, 40, 44, 47, 50–53, 56–58]. Detecting these vulnerabilities typically involves first imagining a potential vulnerability and then writing complex test programs to check for the vulnerability. Once detected, fixing them usually carries a significant performance cost. Pre-silicon fuzzing is a scalable alternative, but existing approaches are all incapable of detecting *exploitable* information leakage in a *generic* manner. This paper builds such a mechanism for the first time and shows its applications by enabling the design of a new fuzzer that is capable of discovering new microarchitectural vulnerabilities in a variety of scenarios, from constant-time violations to various classes of transient execution leakage.

The overfitting problem. Existing pre-silicon microarchitectural fuzzers overfit in three fundamental ways. First, they overfit to *particular microarchitectural structures* by manually tagging the ones where information may leak from (*source components*) or to (*sink components*) [14, 17, 38]. Tagging source components misses out on other leaky structures, and making assumptions on sink components may result in detecting unexploitable cases of information propagation. Second, they overfit to *particular vulnerabilities* by bootstrapping program generation with seeds that trigger known vulnerabilities [17, 19, 38]. Consequently, they have difficulties generating programs that sufficiently deviate from these seeds to trigger different vulnerabilities. Third, they overfit to *particular classes* of vulnerabilities by basing program generation on templates tailored to trigger particular classes of vulnerabilities, like constant-time violations [7] or specific kinds of transient execution leakage [14, 23]. As such, they fail to generate test cases that sufficiently deviate from these templates to trigger different classes of microarchitectural

vulnerabilities. The crux of the overfitting problem is the lack of a generic mechanism for detecting exploitable information leakage without making assumptions on the information leakage path or the shape of the programs that can trigger the information leakage.

Information flows in a CPU. The ISA defines how information flows through programs architecturally. For example, an addition that operates on secret data will produce a result that contains secret information. Similarly, a branch that depends on secret data will cause the program's control flow to carry secret information. In an ISA-compliant CPU, any architectural information flow defined by the ISA will materialize in the concrete CPU implementation. The architectural information flows are thus a subset of all microarchitectural information flows. The converse is not always true; in some cases, the microarchitectural implementation of the CPU will expose more information than defined by the ISA. Not all excess information flows in the microarchitecture constitute an information leakage vulnerability, however. For example, an entry in some structure with unset validity bit may contain another security domain's information. If this entry remains invisible to software, this excess microarchitectural information flow does not constitute an information leakage vulnerability. We make a key observation that if an excessive microarchitectural flow affects the timing of a particular instruction, it becomes architecturally visible as software can now measure the timing of that particular instruction (i.e., forming a covert channel for the excessive microarchitectural flow). Hence, if we can detect the excessive microarchitectural information flows that affect the timing of any instruction, then we have a generic mechanism for detecting exploitable information leakage.

MileSan. The Microarchitectural leakage Sanitizer (MileSan) is a new mechanism that detects microarchitectural leakage for random test programs, CPU implementations, and memory locations that are tagged as secret. MileSan utilizes the insight that microarchitectural information leakage vulnerabilities can be detected based on the difference between architectural and microarchitectural information flows from the secret. MileSan calculates architectural flows with software-level taint tracking when considering the program and secret memory locations (i.e., taint sources), and microarchitectural flows through (dynamic) hardware-level information flow tracking [2, 43, 45, 49] when executing such programs and considering the same secret memory locations. To detect whether the excessive microarchitectural information flows can ever become architecturally visible, MileSan checks for the propagation of such flows to the Program Counter (PC). A tainted PC signals the existence of a secret-transmitting instruction in the test program with secret-dependent execution timing, resulting in a covert channel that transmits the secret information. Hence, MileSan does not assume any microarchitecture-specific source or sink components, enabling the detection of exploitable microarchitectural leakage in a generic way. We have applied MileSan on Proof of Concepts (PoCs) from previous microarchitectural fuzzers to discover that in fact a third of them are not exploitable (i.e., there exists no covert channel that transmits the secret information). In the remaining cases, MileSan shows which microarchitectural component can transmit the secret information.

RandOS. An architectural taint explosion due to a tainted pointer or PC blinds all potentially excessive microarchitectural information flows. In addition to enabling the detection of exploitable information leakage, MileSan enables the generation of random test programs by leveraging *Taint-Aware In-Situ Simulation* (TAISS), a novel technique that allows arbitrarily complex computations on tainted data to create interesting microarchitectural information flows, while avoiding architectural taint explosions. TAISS enables the generation of random programs with strictly-determined architectural information flows of tainted data without assuming any particular vulnerability or a class of vulnerabilities. Leveraging TAISS, we build RandOS, an RTL fuzzer that generates fully-randomized operating systems (without relying on any template), consisting of randomized instructions running in randomized address spaces and privilege levels which we collectively refer to as domains. TAISS enables RandOS to perform computations on tainted (secret) data in *taint source domains* without triggering architectural taint explosion and closing all architectural information flows of tainted data to *taint sink domains*. RandOS detects microarchitectural vulnerabilities through taint flows to the PC, flagged by MileSan. The mapping of taint source and sink domains flexibly captures various classes of microarchitectural vulnerabilities, without overfitting towards any particular case. To provide some examples, a tainted PC in the taint source domain signals intra-domain leakage, like the various classes of intra-domain Spectre or constant-time violations, while a tainted PC in a taint sink domain signals the detection of cross-domain information leakage, like cross-domain Spectre, Meltdown or Microarchitectural Data Sampling (MDS) type vulnerabilities [1, 3, 6, 8, 9, 28, 30, 32, 34–36, 39, 40, 44, 47, 50–53, 56–58].

We evaluate RandOS on a set of RISC-V processors of various complexities (Kronos [29], CVA6 [59], Rocket [4], BOOM [5] and OpenC910 [13]) and demonstrate that RandOS is effective in discovering a diverse set of known and new vulnerabilities. RandOS provides a 4.5× speedup in comparison with SpecDoctor, the state of the art template-based microarchitectural fuzzer that finds exploitable information leakage. In addition, MileSan detects 19 new microarchitectural vulnerabilities triggered by RandOS, including previously undiscovered constant-time violations and Spectre-V1 on OpenC910, cross-privilege MDS and Spectre-SLS on CVA6, and transient Meltdown though the TLB on BOOM.

Contributions. Our contributions are as follows:

- We introduce MileSan, the first microarchitectural leakage sanitizer that is capable of detecting *exploitable information leakage*. Using MileSan, we find that a third of the PoCs from existing microarchitectural fuzzers are in fact false positives.
- Relying on MileSan for detection and program generation, we build RandOS, a new microarchitectural fuzzer that constructs fully randomized operating systems for exploring microarchitectural information flows within and across security domains.
- We evaluate RandOS on five RISC-V processors with varying complexity and show that it is 4.5× faster than SpecDoctor and discover a diverse set of unknown vulnerabilities, ranging from constant-time violations of single instructions to various types of transient execution vulnerabilities, including Spectre-SLS, MDS and Meltdown-like cross-privilege leakage through the TLB.

Ethical considerations and open sourcing. We initiated responsible disclosure with the developers of affected CPUs. Additional information including the source code of MileSan and RandOS can be found in the following link: https://comsec.ethz.ch/milesan

2 Background

In this section, we provide background on RISC-V, information flow tracking, microarchitectural pre-silicon fuzzing, and its overfitting problem.

2.1 RISC-V

RandOS targets RISC-V [55], which is a free and open Instruction Set Architecture (ISA) that is well-represented in the open-source hardware community. RISC-V consists of a base integer ISA and extensions such as F (floating-point), D (double-precision), M (integer multiplication and division), A (atomic), and C (compressed instructions). Compared to other established ISAs, the RISC-V ISA has relatively few instructions. RISC-V commonly supports up to three privilege levels: machine (M), supervisor (S) and user (U) mode. The M mode is the only mandatory privilege level. Interrupts and exceptions allow transitioning upward in the privilege hierarchy, and the `mret` and `sret` instructions allow transitioning downward. RISC-V supports virtual memory in S and U modes.

2.2 Information flow tracking

Information Flow Tracking (IFT), also known as taint tracking, monitors the influence of *sources* on *sinks*, represented by taint. Dynamic IFT (DIFT) tracks information flows at execution time depending on the program's (i.e., software DIFT) or hardware design's (i.e., hardware DIFT) inputs. In comparison, Static IFT (SIFT) calculates the information flows before execution. Information flows can be *explicit* (e.g., the result of an addition is affected by both operands) or *implicit* (e.g. a variable's value assignment depends on which side of a branch is taken). Certain information flows quickly lead to the creation of more information flows in the program or design, to the point that they can cover all possible information flows inside a program or a hardware design. This phenomenon is known as *taint explosion* [26, 41] which typically happens when memory pointers or control flows get tainted. Software IFT is implemented by tracking how the information flows with respect to the instructions defined in the ISA [27, 31]. In hardware, sinks are influenced through the circuit semantics. Hardware DIFT is implemented by automatically inserting additional circuitry to track dependencies dynamically [2, 43, 45, 49]; such tracking has been used in the past for integrity and confidentiality enforcement [21, 22], timing flow identification [33] and information flow isolation [33].

2.3 Pre-silicon microarchitectural fuzzing

Traditionally, discovering microarchitectural vulnerabilities involves manually writing test cases and expert reasoning about the microarchitecture. This process is laborious and it is easy to miss cases. Pre-silicon microarchitectural fuzzers aim to automatically uncover such vulnerabilities and increase the confidence in the design. Microarchitectural fuzzers must consider the following aspects.

Design preparation. Before fuzzing, some microarchitectural fuzzers require the user to prepare the design under test (DUT). This enables instrumentation for coverage feedback [14, 23] or monitoring particular microarchitectural structures [7, 14, 23].

Program generation. Once the DUT is set up, the fuzzers generate programs for testing the DUT. Usually, program generation requires templates or smart seeds derived from known vulnerabilities [17, 23]. Some fuzzers rely on coverage feedback to improve program generation in subsequent rounds [14, 23].

Bug detection. Finally, the fuzzers might detect microarchitectural vulnerabilities in the DUT when running the programs. Prior work either relies on introspection to monitor contents of specific microarchitectural structures [7, 14, 17] or measurements of the execution times of specific instructions [23]. They detect a vulnerability when the expected and observed behavior somehow deviate.

2.4 Overfitting in microarchitectural fuzzers

Existing pre-silicon microarchitectural fuzzers make certain assumptions to make their program generation and bug detection tractable. First, they assume that microarchitectural leakage paths of known and unknown vulnerabilities match closely [14, 17, 38]. Based on this assumption, they tag specific microarchitectural structures where information may leak from (*source components*) or to (*sink components*). This means that they fail to account for other leaky structures when selecting sources, or detect false positives when flagging leakage to selected sinks that remain architecturally inaccessible. Second, they overfit to particular vulnerabilities, like Spectre or Meltdown [17, 38]. They follow mutation-based approaches using seed programs that trigger particular vulnerabilities and therefore struggle to generate programs that are sufficiently different from these original seeds to trigger different vulnerabilities. Third, they usually look for a particular class of vulnerabilities, such as constant-time violation [7] or transient execution leakage [14, 17, 23]. They therefore base their program generation mechanisms on templates derived from PoCs that trigger particular classes of vulnerabilities and cannot generate test cases that deviate sufficiently from these templates to trigger different classes of vulnerabilities.

The first assumption leads to *hardware overfitting*, as vulnerabilities are assumed to leak through particular sources and sinks in a given microarchitecture. The second and third assumptions lead to *software overfitting*, as vulnerabilities are assumed to be triggered by specific programs. We can alleviate hardware overfitting using a generic mechanism that can detect any exploitable information leakage. Furthermore, if this mechanism can be program agnostic, then it would enable the generation of arbitrary programs, alleviating software overfitting. Designing such a mechanism and effectively using it bring certain challenges which we discuss next.

3 Overview of Challenges

It is not immediately clear how we can distinguish architectural from microarchitectural information flows in a microarchitecture-agnostic manner and detect whether the difference can be architecturally observed. It is also unclear how this can help us generate

random programs for fuzzing. These points lead us to our first challenge:

Challenge 1. Design a generic mechanism for detecting exploitable information leakage and facilitating program generation.

Section 4 introduces MileSan, a microarchitectural leakage sanitizer that leverages differential hardware-software taint tracking to detect exploitable information leakage. MileSan relies on the key observation that software-level taint tracking considers expected architectural information flows, while hardware-level taint tracking considers all microarchitectural information flows in a given CPU implementation. Microarchitectural information leakage emerges as the architecturally observable difference between the two. For a CPU that implements the ISA correctly, the difference between the two levels of information flow is only observable if it affects the timing of an instruction that acts as the covert channel. This insight allows MileSan to provide a generic detection mechanism for exploitable information leakage that only considers memory as the taint source and the PC as the taint sink since a variable-time instruction leads to information flows to the PC. To facilitate program generation, MileSan employs Taint-Aware In-Situ Simulation (TAISS), a novel mechanism that relies on software taint tracking during program generation to avoid architectural taint explosion that blind microarchitectural flows. Our next challenge is using MileSan for designing an effective microarchitectural fuzzer:

Challenge 2. Employ MileSan for effective microarchitectural fuzzing.

Section 5 introduces RandOS, a new microarchitectural leakage fuzzer that leverages MileSan for program generation and information leakage detection. RandOS generates random test cases with strictly-determined architectural information flows of secret data. A test case generated by RandOS contains randomized data and instructions across randomized address space layouts and privilege levels, interacting through system calls, exceptions and shared memory, akin to a fully *random operating system*. Given that vulnerabilities may be triggered by complex interactions between various components of a complex test case, understanding the vulnerability and triaging the root cause of the information leakage become a challenge:

Challenge 3. Identify and triage information leakage.

Section 6 describes RandOS's triaging mechanisms, which shorten and simplify leakage-revealing test cases that originally contain up to several thousands of instructions. These mechanisms allow us to identify how the information is leaked and the instructions that contribute to the leakage. As an example, using these mechanisms, we could triage a complex Meltdown-like vulnerability that relies on speculative execution and is (only) observable through the TLB.

```
add s0, a0, a1
```

Listing 1: An example of explicit information flows from `a0` to `s0`. `a0` is tainted.

```
beq a0, a1, target_addr
```

Listing 2: An example of implicit information flow from `a1` to the program counter. `a1` is tainted.

4 MileSan

This section introduces MileSan, a microarchitectural leakage sanitizer that allows detection of arbitrary microarchitectural leakage vulnerabilities. We introduce the relevant concepts, namely architectural and microarchitectural information flows and observability in Sections 4.1 to 4.3. We then discuss taint-aware in-situ simulation, the core MileSan mechanism for treating architectural information flows to enable arbitrary program generation in Section 4.4. Finally, we discuss MileSan in the context of architectural isolation domains such as address spaces and privilege levels where information can leak microarchitecturally in Section 4.5.

4.1 Architectural information flows

Definition. Architectural Information Flows (AIFs) are information flows derived from the ISA. They can be either *explicit* or *implicit*. For example, arithmetic instructions explicitly propagate information flows from their source registers to their destination registers. Listing 1 shows an `add` instruction where the source register `a0` is tainted. Execution of this instruction explicitly taints the destination register `s0`. In contrast, the execution of a conditional branch, like the `beq` in Listing 2, implicitly propagates taint from the source register to the PC. Both for conditional and unconditional branches, the sequence of PC values, and thus the following sequence of executed instructions, is determined by the values of the branch's operand registers. Therefore, if the PC gets tainted, all subsequent instructions also carry an *implicit* information flow originating in the values of the source registers of previous branches, and are thus tainted. This effect is a case of *architectural taint explosion* [26, 41], which we need to avoid when using MileSan as we will show in Section 4.3. Another case of architectural taint explosion which we should avoid is tainting memory pointers, which can lead to tainting the PC (e.g., through a possible page fault).

Implementation. To the best of our knowledge, there is no open-source solution for tracking bit-wise information flows through RISC-V programs. To implement software IFT for RISC-V programs, we leverage the observation that explicit information flows of instructions can be directly derived from the taint propagation rules of the corresponding RTL macro cells [43]. We provide a full list of the correspondence between arithmetic instructions and RTL macro cells in Table 6 in the Appendix. We leverage our implementation of software IFT when guiding program generation as discussed in Section 4.4.

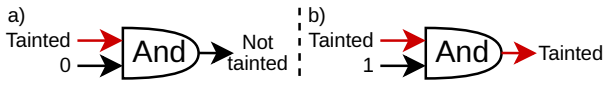

Figure 1: An example of microarchitectural information flow [49].

4.2 Microarchitectural information flows

Definition. Microarchitectural Information Flows (MIFs) are the logical information flows in the CPU's microarchitectural implementation. Figure 1 shows an example MIF through an AND gate. Depending on actual and taint values in the design at a given time, the gate's output will be tainted or not. Hardware taint tracking captures these flows and can be expanded to an entire design [21, 43]. In CPUs, MIFs are a superset of AIFs, as any information flow that is architecturally observable materializes in the microarchitecture, which concretely implements the architectural specification.

To improve performance, the microarchitecture typically implements caches and buffers to hide memory latency, may execute instructions out of order or speculatively, and may employ data-dependent optimizations [12, 28, 30, 34–36, 50, 51, 53]. For example, the latency of a load instruction depends on the contents of the cache, which is determined by the addresses of previous memory operations. Thus, the address of the load determines the number of cycles that the CPU must wait until it can perform computations that depend on the requested data, introducing new MIFs to the PC (through timing). Hence, these performance enhancements introduce MIFs that are disjoint from AIFs.

Contrary to AIFs to the PC that determine the architectural control flow, these MIFs to the PC determine the *microarchitectural valuations of the PC* over time, i.e., the microarchitectural control flow [11], and *not the architectural sequence of PC values*. This results in a *microarchitectural taint explosion*, since the timed sequence of instructions executed on the CPU results in the propagation of the MIFs to all architecturally observable elements in the CPU, such as registers or memory, whose sequences of values over time depend on the microarchitectural control flow. The difference between architectural and microarchitectural taint explosions is subtle but crucial: architectural taint explosions emerge because an instruction determines the architectural PC values based on its tainted input. However, microarchitectural taint explosions emerge because an instruction determines the *particular points in time* when the PC takes its architectural values based on its tainted input (i.e., forming a timing side channel). As we will soon discuss, such microarchitectural taint explosions enable MileSan to capture exploitable information leakage.

Implementation. We rely on recent work on hardware DIFT [43, 45] to track the MIFs in the CPU. We preliminarily instrument the CPUs with either CellIFT [43] or HybriDIFT [45], which allow tracking taints in the CPU automatically during program execution.

4.3 Observability

Definition. An information flow is said to be architecturally observable if it influences the valuation of some architecturally visible component at some point in time. All AIFs are architecturally observable, as they are the expected flows from the ISA and therefore must materialize in the architecturally observable elements at commit time. Yet not all MIFs are architecturally observable. For example, a MIF in the design may be cleared by unsetting a validity bit, without ever becoming architecturally observable. Similarly, a MIF to a physical register might remain inaccessible as long as no architectural register maps it.

Microarchitectural information overflows. Microarchitectural Information Overflows (MIOs) are the architecturally observable differences between MIFs and AIFs.

MileSan foundation: MIO = obs. [MIF - AIF]

While architectural taint explosions *conceal* MIOs due to the abundance of AIFs, microarchitectural taint explosions can *amplify* MIOs. This results from the fact that only very few MIFs are needed to trigger microarchitectural taint explosions, which can be observed from any architectural component in the CPU, such as memory, PC or the register file. Contrary to their architectural counterpart, microarchitectural taint explosions can thus be leveraged to *expose* MIOs. However, in a correct CPU implementation, there can be no MIO that violates the ISA. Hence, MIOs can only affect the microarchitectural valuations of some architecturally visible components over time, rather than the sequence of the values on these architecturally visible components. This is because the architectural sequence of values is strictly determined by the ISA. In addition, as soon as MIOs affect the commit time of some instruction, they propagate to the PC, and as such become architecturally observable and can be measured from software through timing.

MileSan. MileSan leverages this insight, and monitors taint propagation to the PC as taint sink. This ensures that any flagged leakage is architecturally observable, contrary to prior work that detects leakage to selected microarchitectural buffers [14, 17, 38] whose contents might remain architecturally hidden.

4.4 Taint-aware in-situ simulation

To leverage MileSan for generating arbitrary test cases (i.e., programs), the only requirement is avoiding architectural taint explosions that blind MIOs, yet involving computation on tainted data to potentially form MIOs. We achieve this with a novel technique which we refer to as Taint-Aware In-Situ Simulation (TAISS). TAISS enables arbitrary computations on tainted data by providing an interface that allows a fuzzer to query the architectural values and taints during each step of the program generation.

First, the fuzzer selects the arbitrary memory regions it wishes to taint. TAISS then provides a set of safe instructions (including operands), that the application can append to the program without triggering an architectural taint explosion. At the start of program generation, all instructions are allowed because taint has not yet been introduced into the data flow. Once the application chooses to append an instruction that loads tainted data, memory operations with tainted addresses as well as branches with tainted operands are forbidden. TAISS then computes the architectural taint propagation for the instructions that the application chooses to append to the program: for memory operations, it tracks the tainted and untainted memory regions. For arithmetic instructions, it computes the taint

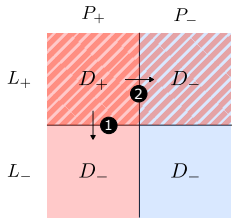

Figure 2: Architectural taint access control restricts access to tainted data to taint source domains $D_+ = (L_+, P_+)$. MIOs can cross layout- or privilege boundaries (❶ and ❷ respectively), and violate those the access policies.

propagation from the source to the destination registers based on the software IFT rules. Thus, for each instruction added to the program, TAISS precisely tracks the information flow of tainted data through the program. Fuzzers (or other applications) can leverage TAISS to generate programs that avoid architectural taint explosion by guiding architectural taint propagation.

4.5 Domain support

Modern CPUs implement isolation mechanisms such as privileges and virtual address spaces that restrict access to architectural elements, such as certain control and status registers or different regions of memory. These isolation mechanisms allow running untrusted software on a host system by strictly confining its capabilities, ensuring that the software cannot access any resource that was not explicitly allocated to it. However, MIOs can cross privilege and/or address space boundaries and thus violate these architecturally-enforced isolation mechanisms.

To allow the users of MileSan to generate arbitrary multi-domain programs and reason about isolation mechanisms in terms of MIOs and taint propagation, we extend taint sources and sinks to different ISA-defined isolation mechanisms. To this end, we define taint address space layouts, taint source and sink privileges, and taint source and sink domains as follows:

Definition 1: Taint layouts. Taint source and taint sink layouts, denoted by L_+ (respectively L_-), are sets of virtual address space layouts that (respectively do not) map pages containing tainted data or code that may interact with it.

Definition 2: Taint privileges. Taint source and taint sink privileges, denoted by P_+ (respectively P_-), are sets of privileges that are granted (respectively denied) access to tainted data.

Definition 3: Taint domains. Taint source and taint sink domains, denoted by D_+ (respectively D_-), are pairs of layouts and privileges, i.e. $D_i = (L_j, P_k), i, j, k \in \{+, -\}$ where

$$i = \begin{cases} + & \text{if } j = k = + \\ - & \text{else} \end{cases}$$

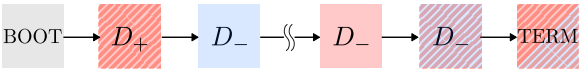

Figure 3: Overview of an example RandOS program. Starting from the initial BB, an arbitrarily long chain of fuzzing BBs, traversing various privileges and layouts, leads to the terminating BB that halts execution. The colors and patterns illustrate the taint source (D_+) and sink (D_-) domains, see Figure 2.

Figure 2 illustrates the resulting access control policies at the architectural level. MileSan can detect MIOs that remain within the same domain (e.g., constant-time violation [7] or intra-domain Spectre [28]) when it detects a tainted PC in the source domain, and MIOs across domains (e.g., Meltdown [30] or cross-domain Spectre [56, 57]) when it detects a tainted PC in the sink domain. These concepts enable leveraging MileSan to capture arbitrary leakage scenarios as we will demonstrate in the next section.

5 RandOS

We introduce RandOS, a microarchitectural fuzzer that leverages MileSan for program generation and leakage detection. RandOS produces random and unbiased test cases that traverse address space layouts and privileges while maintaining a strictly-defined architectural information flow of secret data, showcasing the capabilities of MileSan's TAISS. The resulting programs are akin to random operating systems, composed of random instructions running in different privileges and virtual address spaces. For each test case, RandOS randomly selects a domain to be trusted (i.e., taint source) and others to be untrusted (i.e., taint sinks). RandOS can automatically trigger and detect various known and unknown microarchitectural information leakage vulnerabilities by checking if the PC gets tainted in any of the sink domains (i.e., taint sinks). This means we do not have to make any assumption on the required sequences of instructions or involved microarchitectural components, demonstrating the generality of MileSan.

The randomized OS structure adopted by RandOS consists of a basic block (BB) executed in M-mode that boots the CPU, followed by a sequence of BBs that execute in either trusted or untrusted domains in different privilege levels and address spaces. Execution finishes with a terminating BB that signals the CPU to halt. Figure 3 depicts an overview of the resulting chain of BBs. The BBs perform random computations and interact across privilege levels and virtual address spaces through system calls, machine calls, exceptions and shared memory.

For a given test case, either the machine mode running in the physical address space, or kernel or user mode, running in particular virtual address spaces, is tagged as trusted (taint source domain) and can operate on tainted data. Execution in taint source domain must carefully manage the computation on taint in order to avoid architectural taint explosions. TAISS thus blocks the use of tainted data for control-flow decisions and operations with memory pointers in the taint source domain. Random computations overwrite tainted registers with untainted data before entering taint sink domains from the taint source domain. Therefore, BBs executing in taint sink

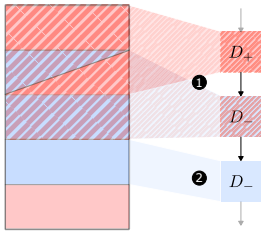

Figure 4: An example data memory mapping. The topmost data page holds tainted data and can only be accessed by domains in D_+ because only those domains have both a layout that maps the page to it and the permissions set accordingly. Both domains in D_+ and D_- can access shared data pages if they have appropriate layouts and permissions (❶), while unshared data pages might only be accessible from a single privilege and layout (❷).

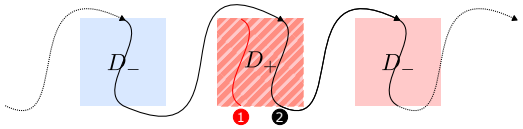

Figure 5: BBs executing in D_+ have strictly separated tainted (❶) and untainted (❷) data flows. The untainted data flow affects the registers used by control flow instructions, creating interdependencies between the control flow and untainted data flow. The tainted data flow may never architecturally spill to the untainted data flow, as this would trigger architectural taint explosions when the input register of a control flow instruction gets tainted. Since BBs executing in D_- do not have access to tainted data, their architectural data flows remain untainted.

domains can never architecturally encounter tainted data as all architectural information flows remain closed at all times, i.e., there is no architectural information flow of tainted data from taint source to taint sink domains. RandOS can thus leverage MileSan to detect leakage in taint source and sink domains: if the PC gets tainted while executing in a taint source domain, the test case has triggers intra-domain leakage, e.g., a variant of intra-domain Spectre or a constant-time violation. If the PC gets tainted while executing in a taint sink domain, the test case has triggered cross-domain leakage, e.g., Meltdown or cross-privilege Spectre.

5.1 Memory preparation

Taint sources. RandOS starts by reserving randomly chosen physical memory frames for the page tables, and booting and terminating BBs. RandOS then distributes a set of data frames, filled with random content, in random offsets in physical memory. Using MileSan, RandOS taints a random subset of these data pages to later inject taint into the data flow and track taint propagation during program generation. The taint sources are therefore architectural only, avoiding overfitting towards specific microarchitectural leakage sources as discussed in Section 2.4.

Architectural isolation. To enforce architectural isolation between taint source and sink domains, RandOS relies on page tables. RandOS iterates over the list of physical page frames and checks for each physical frame whether it has already been reserved for code, page tables or data. We discuss each case next.

If the frame is reserved for the booting BB, then RandOS does not map it in any of the address space layouts as the test case executes it in M-mode only. RandOS maps the frame that contains the terminating BB as executable in all privilege levels in all layouts. For the rest of the frames that contain code pages, RandOS maps each code page randomly to one of the domains. For data pages that are untainted, RandOS maps them to a randomized selection of domains as shared memory. Lastly, and perhaps most importantly, for frames that contain a tainted data page, RandOS maps it to a D_+ domain. RandOS also optionally maps the tainted data page in the L_+ layout and P_- privilege without permissions to detect cases where the CPU microarchitecturally accesses tainted data without the right privilege. Figure 4 illustrates a simple example of data page mapping in a RandOS test case. After deciding the mapping of code and data between different domains, RandOS proceeds to generate the randomized code in each domain.

5.2 Code generation

Starting execution. The code generation begins with the booting BB that brings the CPU into a predefined state. It is composed of a fixed sequence of instructions, completing with a control flow instruction that guides execution into the first randomized fuzzing BB. RandOS randomly chooses the target (taint source or sink) domain that this fuzzing BB executes from and must therefore find a free memory region that is mapped appropriately to place the BB. After identifying a suitable memory location, RandOS generates the BB using TAISS instruction by instruction in the taint source domain and otherwise randomly in the taint sink domains. We discuss both cases next.

Executing in taint source domains. The BBs executing in taint source domains have access to tainted data. They load tainted data from memory and involve it in their computations, thus naturally spreading taint to microarchitectural structures, such as the register file, caches or load and store buffers. However, control flow instructions and memory pointers should not consume tainted data, as this would trigger an architectural taint explosion that overshadows microarchitectural leakage that MileSan aims to detect. Therefore, RandOS must carefully orchestrate architectural information flow of tainted data. RandOS relies on MileSan's TAISS to achieve this. It thus queries MileSan during each step of the program generation to obtain the architectural taints in memory and registers. It then avoids architectural taint explosions by never providing tainted registers as operands to control flow instructions or as memory pointers, while allowing for otherwise random computations.

Executing in taint sink domains. RandOS does not grant BBs that execute in taint sink domains architectural access to tainted data pages, and no architecturally accessible components store tainted data when the test case transitions from a taint source into a taint sink domain. Therefore, the computations in the taint sink domains remain random and unconstrained with the only restriction that the control flow must remain valid. This means that the execution

must continue at the beginning of the next BB once the current BB has finished.

Domain transitions. RandOS test cases must be able to arbitrarily transition between taint source and sink domains. To do so, RandOS needs to prepare registers to point to the locations of the next BBs, considering their physical location, address space and privilege. These can be general purpose registers (GPRs) used as input operands for branches, or control and status registers (CSRs) used by system calls and exceptions. Similar to [42], RandOS employs a number of finite state machines (FSMs) that prepare registers to hold these target addresses, while involving only untainted registers in their computations to ensure that the target register remains untainted. Those FSMs run asynchronously, interleaved by other random instructions. The random instructions that return untainted results may affect the computations of the FSMs, as they do not risk propagating taint to the register being prepared. Once a FSM has set up a register, a control flow instruction can consume it and arbitrate execution to the next BB. This essentially results in two separated data flows: The untainted data flow, that may affect the control flow, and the tainted data flow, that may never architecturally spill to the untainted data flow. Figure 5 illustrates this concept.

Finally, before execution can traverse from a taint source to a taint sink domain, we clear all tainted architectural components that the taint sink domain could access. RandOS therefore queries MileSan to obtain the list of tainted registers and shared memory locations and adds random computations that overwrite them with untainted data before allowing the transition to a BB in a taint sink domain. Therefore, all architectural information flow paths of secret data from taint source to sink domains remain closed at all times.

Terminating execution. After some number of fuzzing BBs, the test case finishes execution with the terminating BB that signals the CPU to halt. RandOS maps the terminating BB executable to all domains. Therefore, any BB from any domain can enter it when RandOS has generated sufficiently many BBs.

6 Leakage Identification

RandOS produces test cases that can be hard to reason about due to their length and random cross-domain transitions. To ease root cause analysis, RandOS provides reduction facilities that simplify the test cases to the minimal set of instructions and tainted data that are required to reveal the leakage, as depicted in Figure 6.

Primer and leaker instructions. Leakage can have various root causes. In the simplest case, single instructions with data-dependent execution times cause information flows from their operands to the PC [11]. However, microarchitectural optimizations that trigger leakage might require more complex instruction patterns composed of primer and leaker instructions, as introduced in Definition 4.

> **Definition 4: Primer and leaker instructions.** The set of primer instructions I_{prime} prepares the microarchitectural buffers with sensitive data while the leaker instruction i_{leak} exposes the leakage architecturally through timing.

While several primer instructions might be required to prime the CPU into a specific microarchitectural state, one leaker instruction

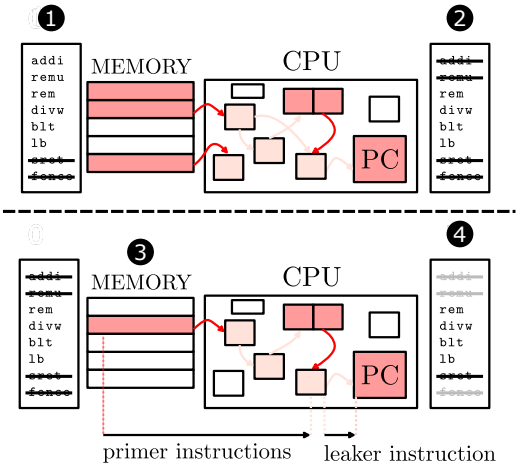

Figure 6: Program reduction overview. Some test case generates architectural and microarchitectural information flows that leak secret data to the PC. Reduction starts skipping instructions from the back to identify the leaker instruction ❶. It then skips instructions from the front to find the first primer instruction ❷. Taint reduction then identifies the minimal set of tainted address required to trigger the leakage ❸ and dead code reduction identifies code regions that contribute to the leakage non-architecturally ❹. The result is a minimal test case, consisting of possibly multiple primer instructions and a single leaker instruction, of which some might contribute to the leakage non-architecturally (e.g., through transient execution), and the minimal set of tainted memory locations required to trigger the leakage.

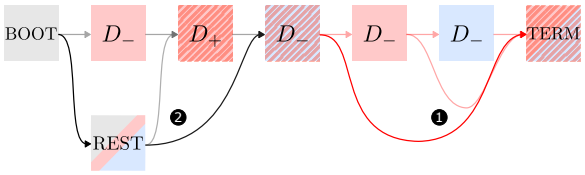

Figure 7: Control flow reduction. ❶ skips BBs from the back until the leakage disappears. ❷ replaces BBs from the front with a restoring BB (*REST*). The remaining BB contains the leaker instruction.

suffices to finally propagate the taint to the PC. In essence, the leaker instruction acts as the covert channel that leaks the information from the primer instructions. Having several primer instructions hampers manual examination of the test case and calls for reduction facilities.

Identifying the leaker instruction. Control flow reduction aims to shorten test cases as much as possible while preserving the microarchitectural information flow to the PC. It thus starts by iteratively skipping BBs and replacing the last instruction in each BB - that is the instruction that moves the PC to the next BB - with a jump to the terminating BB, starting from the penultimate BB at the end of the test case (❶ in Figure 7), before re-simulating the RTL. Importantly, since the terminating BB is mapped for each privilege and layout, any BB can reach it with a jump, independent of the

domain it executes from. When the leakage disappears after RTL re-simulation, the control flow reduction re-includes the most recently skipped BB and continues this strategy on an instruction level, starting with the last instruction, replacing each instruction in a BB with a jump to the terminating BB until the leakage disappears again. The most recently removed instruction is the leaker instruction. If the leaker instruction is executed from a taint sink domain, the shortened test case triggers leakage that violates the imposed access control policies discussed in Section 4.5.

Isolating the primer instructions. To identify the first instruction that is necessary to prime the microarchitecture, RandOS reduces the test case further from the front. Concretely, starting with the first BB up until the BB that contains the leaking instruction, a BB is skipped and the initial BB jumps to a restoring BB that sets the architectural state (including the taints) according to the skipped BBs and resumes execution at the next considered BB (❷ in Figure 7). Software IFT provides the required information about architectural taints and values required for restoration. We then apply the same procedure at instruction level to find the first primer instruction required to trigger the leakage. Since this reduction from the front might change the leakage mechanism, RandOS checks during each step that removing the leaker instruction removes the leakage as well, thus ensuring its invariance.

Dead code reduction. RandOS removes code from the test case by skipping BBs while still keeping them in the program memory. Despite not being architecturally executed, dead code might contribute to leakage, e.g., through transient execution. To identify relevant code segments, we first check if removing all dead code from program memory hides the leakage or not. If the leakage disappears, then dead code is involved in the set of primer instructions. The reduction then iteratively reduces dead code following a binary search algorithm until it reveals the minimal contiguous region of dead code required to trigger the information leakage. This dead code snippet could correspond to a transiently-executed gadget, hence its identification might be paramount for understanding the vulnerability.

Taint reduction. Finally, RandOS reduces the amount of architectural taints to enable a better understanding of the leakage. Initially, a potentially large amount of taints is present in memory and can be imported into the microarchitecture through multiple independent loads, which can interact with each other. To identify the specific data that is being leaked among all the initially-present tainted data, we reduce the taint in the tainted data pages following a binary search algorithm. Similar to dead code reduction, RandOS iteratively remove half of the tainted locations from memory until it finds the minimal number of tainted memory locations required to triggers the leakage. We thus obtain the exact memory locations of the leaked information, further facilitating triaging by providing the specific starting point of the path taken by the leaked data from its initial location in memory up until the PC.

7 Evaluation

We first evaluate test case generation and execution performance of RandOS in Section 7.1. We then analyze the performance of RandOS in discovering known vulnerabilities in Section 7.2, and provide an in-depth discussion of some of the newly found vulnerabilities in Section 7.3. We implement an end-to-end exploit based on one of the discovered vulnerabilities, leaking arbitrary kernel memory from user space in Section 7.4. Finally, we use MileSan to evaluate the exploitability of PoCs from previous microarchitectural fuzzers in Section 7.5.

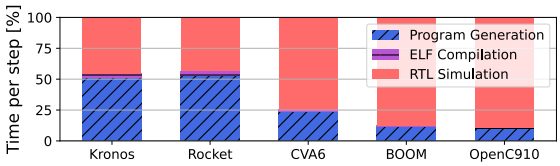

Figure 8: Fuzzing performance breakdown.

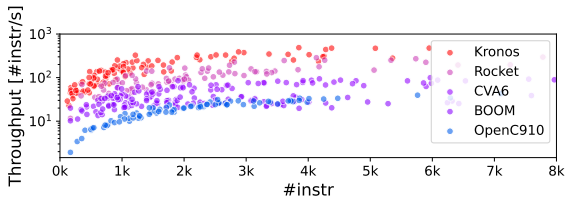

Figure 9: Fuzzing throughput for increasing program length.

Evaluation setting. We obtained the performance results on 60 threads in a Docker environment running on a machine equipped with two AMD EPYC 7H12 processors at 2.6 GHz with 1 TB of DRAM. For RTL simulation, we use Verilator 5.029 and Modelsim 2022.3_1. The performance evaluation in Section 7.1 were carried out using Modelsim.

Evaluated RTL designs. We evaluated RandOS on five different RISC-V cores: Kronos, Rocket, CVA6, BOOM and OpenC910. Kronos (commit `b857643`) is a simple 3-stage 32bit design optimized for FPGA emulation. It does not support virtual memory and is therefore only fuzzed in M-mode in a single domain. Rocket (Chipyard commit `004297b6`) and CVA6 (commit `109f9e9e`) are more complex 5- and 6-stage respectively in-order 64-bit CPUs. BOOM (Chipyard commit `004297b6`) and OpenC910 (commit `e0c4ad8e`) are a 10- and 9-stage out-of-order CPUs.

Instrumentation. We instrument Kronos, Rocket, CVA6 and BOOM using CellIFT [43] and OpenC910 using HybriDIFT [45].

7.1 Fuzzing performance

Component-wise fuzzing performance. Test program generation, compilation and execution (RTL simulation) are factors that determine the end-to-end fuzzing performance. As illustrated in Figure 8, the bottleneck is test-case generation for small designs, and RTL simulation for larger designs.

Length of test programs. Figure 9 shows the fuzzing performance for increasing program length. Fuzzing throughput increases with program length and plateaus after around 4k instructions because the cost of program generation amortizes over the program length, while RTL simulation does not.

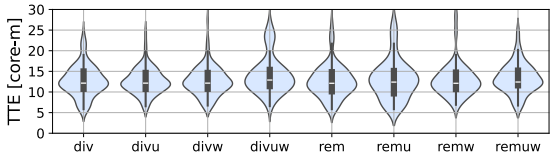

Figure 10: TTE of known constant-time violations on CVA6.

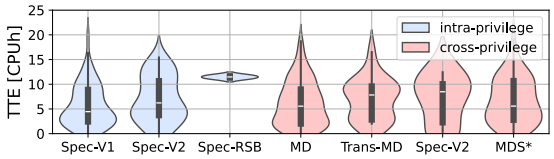

Figure 11: TTE of discovered transient vulnerabilities on BOOM. *We applied the patch provided by [14] to introduce an MDS vulnerability into BOOM.

Vuln.	Mean	Median	Std.Dev.	Speedup †
Spectre-V1	5h35m	4h12m	3h52m	4.8x
Spectre-V2	7h50m	7h56m	4h40m	3.9x
Spectre-RSB	11h28m	11h28m	0h17m	/
Meltdown	6h10m	5h33m	4h33m	5.6x
Trans. Meltdown	6h59m	7h13m	4h12m	3.8x
cp-Spectre-V2	7h41m	8h58m	4h32m	/
MDS*	6h44m	5h34m	4h33m	/
Mean	**7h30m**	**7h16m**	**3h49m**	**4.5x**

Table 1: TTE statistics of known vulnerabilities on BOOM in core hours. *MDS patch provided by [14]. †Speedup relative to SpecDoctor [23].

7.2 Rediscovery of known vulnerabilities

Classification. RandOS classifies known vulnerabilities based on the minimal test case obtained during program reduction. If the leakage is triggered by a single instruction executing with a tainted operand in a taint source domain, it classifies it as a constant-time violation. RandOS classifies intra-domain leakage caused by mispredicted conditional branches, indirect branches, returns or misordered loads followed by transiently executed gadgets as Spectre-V1, Spectre-V2, Spectre-RSB and Spectre-V4, respectively. RandOS classifies cross-domain leakage caused by page faults, followed by transient gadgets, as Meltdown or MDS, which can be distinguished based on the address where the data is leaked from: Meltdown leaks the data that the page fault loads from, while MDS leaks from an unrelated address that S-mode previously loaded a value from. Finally, RandOS classifies cross-domain leakage caused by a mispredicted conditional or indirect branch followed by transient gadgets as transient Meltdown or cross-privilege Spectre-V2, respectively.

Time To Exposure (TTE). RandOS generated exploitable test cases for all known vulnerabilities on CVA6 and BOOM. RandOS discovers all constant-time violations on CVA within minutes as we summarize in Figure 10. MileSan detects constant-time violations with only a single execution of the respective instruction

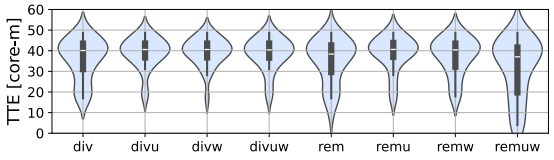

Figure 12: TTE of constant-time violations on OpenC910.

with a tainted input operand, which taints the PC due to the input dependent latency of the instruction. Since RandOS effectively involves tainted data in computations using TAISS, it quickly generates such a test case. RandOS detects all known transient vulnerabilities on BOOM within less than 12 core hours. Figure 11 and Table 1 summarize the results. Compared to SpecDoctor, the state-of-the-art microarchitectural fuzzer that discovers exploitable information leakage, RandOS is between 3.8× and 5.6× faster while detecting several additional vulnerabilities, such as Spectre-RSB, cross-privilege Spectre-V2 and MDS.

7.3 New vulnerabilities

RandOS discovered numerous new vulnerabilities on BOOM, CVA6 and OpenC910. They include constant-time violations on OpenC910, as well as transient vulnerabilities on BOOM, CVA6 and OpenC910. We provide a full list of the discovered vulnerabilities, including 19 new vulnerabilities with the involved domains and covert channels in Table 2. In the following, we discuss a few interesting cases of new vulnerabilities.

Exploitable Meltdown on BOOM (CVE-2025-29343). RandOS discovered a variant of Meldown [30] that allows an unprivileged attacker in one domain to leak data from another domain by probing the Translation Lookaside Buffer (TLB) à la TLBleed [18] . The reduced test case generated by RandOS is depicted in Listing 4 in Appendix A.1. Concretely, the value of an operand register of a branch is determined by a series of `rem[u]` and `divw` instructions. The branch is predicted taken, and a gadget is executed transiently. The gadget loads from a privileged data page and executes another secret-dependent load, which primes the TLB with secret data. A later load leaks to the PC. Analysis of the signal traces (using Pathfinder [10]) reveals that while a page fault is triggered, secret data is nevertheless loaded into the register file. A dependent load uses the register holding the page-faulting data as address and encodes the secret in the TLB. While the subsequent load from the physically indexed cache is blocked, the state of the TLB now encodes the secret data and leaks it during the next address translation. While the original program generated by RandOS was long and complex, our debugging facilities reduced the program to only a few instructions that are executed architecturally. It identified the speculatively executed Meltdown gadget as well as the memory location of the leaked data.

Straight-line speculation on CVA6 (CVE-2025-29340). RandOS discovered a novel case of straight-line-speculation [58] on CVA6 that leaks privileged data from S- to U-mode. Concretely, while executing in S-mode, an `sret` instruction is encountered. The `mstatus.SPP` bit is set to S-mode, and transient execution continues after the `sret` instruction. During the speculative window, a

Vulnerability	DUT	Covert Channel	$\{P_+\}$	$\{P_-\}$	Previously found by	CVE
Spectre-V1	BOOM	DCACHE	$\{S,U\}$	$\{S,U\}$	[14, 23, 38]	/
Spectre-V1-TLB	BOOM	TLB	$\{S,U\}$	$\{S,U\}$	[23]	/
Spectre-V2	BOOM	TLB	$\{S,U\}$	$\{S,U\}$	[14, 23, 38]	/
Spectre-V2-TLB	BOOM	TLB	$\{S,U\}$	$\{S,U\}$	/	requested
Spectre-V4	BOOM	DCACHE	$\{S,U\}$	$\{S,U\}$	[14, 23]	/
Spectre-V4-TLB	BOOM	TLB	$\{S,U\}$	$\{S,U\}$	/	requested
Spectre-RSB	BOOM	DCACHE	$\{S,U\}$	$\{S,U\}$	[14, 23]	/
Spectre-RSB-TLB	BOOM	TLB	$\{S,U\}$	$\{S,U\}$	/	requested
Meltdown	BOOM	TLB	$\{S\}/\{U\}$	$\{U\}/\{S\}$	[14, 17, 23]	/
Trans. Meltdown	BOOM	TLB	$\{S\}/\{U\}$	$\{U\}/\{S\}$	/	CVE-2025-29343
cp. Spectre V2	BOOM	TLB	$\{S\}/\{U\}$	$\{U\}/\{S\}$	/	requested
MDS*	BOOM	TLB	$\{S\}/\{U\}$	$\{U\}/\{S\}$	[14]‡	/
Spectre-SLS	CVA6	TLB	$\{S,U\}$	$\{S,U\}$	/	CVE-2025-29340
cp-Spectre-SLS	CVA6	TLB	$\{S\}/\{U\}$	$\{U\}/\{S\}$	/	CVE-2025-29340
MDS	CVA6	TLB	$\{S\}/\{U\}$	$\{U\}/\{S\}$	/	CVE-2025-46004
Trans. MDS	CVA6	TLB	$\{S\}/\{U\}$	$\{U\}/\{S\}$	/	CVE-2025-46004
`div`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[20]	/
`divu`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[7, 20]	/
`divw`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[20]	/
`divuw`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[20]	/
`rem`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[20]	/
`remu`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[20]	/
`remw`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[20]	/
`remuw`†	CVA6	DIV	$\{M,S,U\}$	$\{M,S,U\}$	[20]	/
Spectre-V1	OpenC910	DCACHE	$\{S,U\}$	$\{S,U\}$	/	requested
Spectre-V1-TLB	OpenC910	TLB	$\{S,U\}$	$\{S,U\}$	/	requested
`div`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005
`divu`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005
`divw`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005
`divuw`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005
`rem`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005
`remu`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005
`remw`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005
`remuw`†	OpenC910	DIV	$\{M,S,U\}$	$\{M,S,U\}$	/	CVE-2025-46005

Table 2: Overview of all vulnerabilities discovered by RandOS including the involved leakage paths and domains. $\{P_+\} = \{P_-\} = \{S,U\}$ denotes intra-domain leakage both within user and supervisor space, whereas $\{P_+\} = \{U\}/\{S\}$,$\{P_-\} = \{S\}/\{U\}$ denotes cross-domain leakage in both directions between supervisor and user space. *MDS patch provided by [14]. † Constant-time violation triggered by the four respective instruction. ‡[14] detects MDS only when artificially tainting the relevant buffers.

secret-dependent memory access is executed, encoding secret data in the TLB. Since the TLB is not flushed during privilege transitions, a later load from U-mode recovers the secret data from the TLB through a timing side channel. The original program that triggered this vulnerability was long and complex, spanning over several privileges and address space layouts. However, our reduction facilities reduced the test case s.t. only the relevant instructions and tainted data remained, allowing us to discover the mechanisms of the vulnerability through analysis of the signal traces [10]. We provide the reduced test case in Listing 7 in Appendix A.4.

MDS on CVA6 (CVE-2025-46004). RandOS discovered a case of MDS on CVA6. S-mode code loads privileged data and involves it in its computations. At some later point during execution, U-mode code attempts to load privileged data and triggers a page fault. However, the privileged data that S-mode has previously loaded is transiently returned. The speculative window is sufficiently long to perform a dependent memory access and encode the privileged data in the TLB. The triaging facilities in RandOS identified the relevant instructions and the address of the leaked data in the test case. We include the reduced test case in Listing 6 in Appendix A.3. The relevant instructions include a load that page faults and a transiently executed dependent load, similar to a Meltdown-like vulnerability. However, taint reduction identified the leaked memory location, which did not match the page faulting address, which helped us identify the root cause of this vulnerability. RandOS also discovered a transient version of this vulnerability through a transient window that is caused by an `ecall` instruction. We include the relevant reduced test case in Listing 5 in Appendix A.2.

Constant-time violations on OpenC910 (CVE-2025-46005). RandOS discovered 8 new constant-time violations on OpenC910,

caused by `div[u][w]` and `rem[u][w]`. The root cause is the common use of the division unit that has data-dependent execution time.

Discussion. The reason why prior work [14, 23] fail at detecting the transient execution vulnerabilities is twofold. First, their rigid program structures fail to account for leakage that requires execution of architecturally secure code in privileged domains. RandOS generates programs that actively involve secret data in their computations in privileged domains, while ensuring that no secret data can architecturally leak to the unprivileged domains. Second, their insensitive detection mechanisms fail to detect leakage through coarse grained side channels such as the TLB. Relying on MileSan, RandOS can efficiently test the execution times of a given program for all possible tainted input values in a single RTL simulation, allowing immediate discovery of a possible covert channel that can leak tainted data.

```
0  la ra, secret
1  la a2, buffer
…  ...              # Evict the TLB, delay t0 results.
6  beqz t0, correct_target
7  ld a0, 0(ra)     # Access secret.
8  andi a0, a0, 1   # Mask out a single bit.
9  slli a0, a0, 12  # Shift secret bit by page offset.
10 add a2, a2, a0   # Add secret bit to buffer.
11 ld a2, 0(a2)     # Encode the secret bit in the TLB.
12 correct_target:
…  ...              # Resume at correct path.
```

Listing 3: Excerpt of end-to-end exploit to read arbitrary kernel memory from user mode on BOOM.

7.4 End-to-end exploit on BOOM

To showcase the effectiveness of MileSan in detecting end-to-end exploitable leakage, we implement a PoC exploit that leaks arbitrary kernel memory from user space based on our new transient Meltdown vulnerability that leaks through TLB on BOOM. Listing 3 shows an excerpt of the exploit. We first evict all entries from the TLB since `sfence.vma` cannot be used from user mode. We then create a transient window through a sequence of floating-point instructions that delay a subsequent branch decision. During the transient window triggered by `beqz`, we access privileged data and encode a single bit of the secret in the TLB. Probing the TLB later reveals the secret bit. The end-to-end exploit leaks approximately 1 bit of kernel memory every 2k cycles, corresponding to a bandwidth of 1Mbps when BOOM is clocked at 2GHz which is inline with existing attacks of this nature [23].

7.5 Assessing PoCs from prior work

Using MileSan, we verified the exploitability of various PoCs provided by prior work. These include constant-time violations [7, 20], as well as transient vulnerabilities [14, 23]. Note that we tested for each PoC the relevant version of the CPU, therefore some of them do not apply to the versions used in the fuzzing campaigns by RandOS.

Constant-time violations. WhisperFuzz [7] report several constant-time violations on CVA6. We tested the provided PoCs by WhisperFuzz using MileSan by tainting the operand values of the affected instructions. By completeness of the underlying hardware IFT mechanism [43], the PC must get tainted if the PoC indeed triggers a constant-time violation. However, our experiments show that this is not the case for a majority of the PoCs. As we show in Table 3, more than half of the reported constant-time violations are in fact false positives of WhisperFuzz. A careful analysis of the PoCs reveals that WhisperFuzz accidentally compares pairs of programs of different instruction counts, naturally leading to these false positives [25].

Instruction(s)	DUT	Covert Channel	$\{P_+\}$	$\{P_-\}$
`divuw`	CVA6	DIV	$\{M\}$	$\{M\}$
`remw`	CVA6	DIV	$\{M\}$	$\{M\}$
`c.add`	CVA6	✗	$\{M\}$	$\{M\}$
`c.sub`	CVA6	✗	$\{M\}$	$\{M\}$
`c.and`	CVA6	✗	$\{M\}$	$\{M\}$
`c.or`	CVA6	✗	$\{M\}$	$\{M\}$
`c.xor`	CVA6	✗	$\{M\}$	$\{M\}$
`c.mv`	CVA6	✗	$\{M\}$	$\{M\}$

Table 3: MileSan validation of WhisperFuzz [7] PoCs.

Vulnerability	DUT	Covert Channel	$\{P_+\}$	$\{P_-\}$
Boombard	BOOM	BPU	$\{S\}$	$\{U\}$
Spectre-RSB	BOOM	DCACHE	$\{S\}$	$\{S\}$
Spectre-TLB	BOOM	TLB	$\{S\}$	$\{S\}$

Table 4: MileSan validation of SpecDoctor [23] PoCs.

Vulnerability	DUT	Covert Channel	$\{P_+\}$	$\{P_-\}$
Spectre-V1*	BOOM	TLB+DCACHE	$\{S\}$	$\{S\}$
Spectre-V2*	BOOM	TLB	$\{S\}$	$\{S\}$
Spectre-RSB*	BOOM	TLB	$\{S\}$	$\{S\}$
Spectre-V4	BOOM	TLB+DCACHE	$\{S\}$	$\{S\}$
Meltdown	BOOM	TLB	$\{S\}$	$\{U\}$
MDS†	BOOM	TLB	$\{S\}$	$\{U\}$

Table 5: MileSan validation of Phantom Trails [14] PoCs. *These PoCs did not encode the secret into any microarchitectural structures. We therefore added dependent loads, which the authors used in the remaining PoCs. † Can only be detected when explicitly tainting the load and store buffers.

Transient vulnerabilities. We test the manually-crafted PoCs provided by SpecDoctor [23] and Phantom Trails [14] for their reported transient leakage vulnerabilities. SpecDoctor provides PoCs that include covert-channel transmission of the secret. However, only some PoCs of Phantom Trails transmit the secret, while others only load secret data into the register file. In those cases, we manually added secret-dependent loads. We tested the PoCs by tainting the memory locations that hold secret data and observing taint propagation to the PC. Tables 4 and 5 depict results including the identified covert channels and affected domains.

8 Discussion

Hardware DIFT without instrumentation. Instead of a CellIFT instrumentation, another way of tracking microarchitectural information flows is to repeat the execution of a program on the CPU

with many different values of the tainted data and observe whether the PC changes. However, the state space is large, and leakage is often related to corner cases, which are hard to encounter by chance, but are entirely covered by CellIFT or similar instrumentations [43, 45, 49].

Precision. Hardware DIFT can theoretically induce false positives [43]. We did not encounter such a case in our evaluation.

Simulator bugs. Due to the additional hardware constructs introduced by CellIFT, simulators sometimes produce incorrect results [46]. Due to a bug in Verilator, we have encountered numerous false positives that disappeared in commercial simulators. Contrarily, we have encountered a bug in commercial simulators, where the instrumentation directly affects the valuations of the original CPU signals. While this can also create false positives, it practically breaks the test program's control flow in all cases. Such mis-simulated programs can then be excluded from further analysis. We encourage future work on RTL simulator reliability, similar to [46].

Impact of discovered vulnerabilities. The maintainers of BOOM and CVA6 plan to fix vulnerabilities caused by meltdown-type leakage in future generations. This includes CVE-2025-29343, CVE-2025-29340 and CVE-2025-46004. The authors of [24] propose a patch that blocks page-faulting data from reaching the register file on BOOM. We experimentally verified that this patch also resolves CVE-2025-29343.

9 Related work

We discuss related work that aims to detect microarchitectural vulnerabilities before tapeout, namely formal and fuzzing-based approaches.

9.1 Formal approaches

Several formal approaches focus on data-oblivious instruction execution and hence relate to RandOS. UPEC [37] leverages unique program execution to verify speculative non-interference and requires extensive manual effort to identify which of the alerts provided by the tool are acceptable. Recent formal approaches introduce a higher degree of automatization. μCFI leverages IFT to formally prove microarchitectural control flow integrity, which demands the absence of unspecified information flows from instruction operands to the PC [11]. RTL2MμPATH [20] formally proves the data independent execution time property of instructions by extracting μpaths from the RTL design. However, they often hit scalability limitations [11, 15, 20, 48, 54].

9.2 Fuzzing-based approaches

Prior work in microarchitectural pre-silicon fuzzing predominantly focuses on specific leakage mechanisms and the involved microarchitectural structures. IntroSpectre [17] observes the propagation of magic values in the microarchitecture under transient execution, Specure [38] identifies changes of selected microarchitectural structures during speculation and SpecDoctor [23] identifies timing side-channels capable of revealing secret data. These approaches all rely on targeted templates for program generation. Phantom Trails [14] combines software taint tracking on the verilated RTL model with program generation based on AFL++ [16] to detect transient leakage to the (physical) register file. WhisperFuzz observes module-level timing variations to identify constant-time violations of single instructions. Both fuzzers make assumptions on either the kind of programs that trigger vulnerabilities or their leakage paths. MileSan is the first generic mechanism for detecting microarchitectural information leakage without making any assumptions on the programs that trigger them or the components that carry the leakage, enabling RandOS to generate random and unbiased programs that trigger various kind of leakages, from transient vulnerabilities to constant-time violations.

10 Conclusion

We presented MileSan, the first hardware sanitizer that is capable of detecting exploitable microarchitectural information leakage. MileSan is based on the key insight that microarchitectural leakage stems from the observable difference between architectural information flows, derived from the ISA, and microarchitectural information flows, as observed in the concrete RTL implementation. Since MileSan relies on fundamental properties of how information flows inside software and hardware, it can detect exploitable information leakage in a generic manner without making any assumption on the source of information leakage and its destination in a given CPU design. MileSan further facilitates the generation of arbitrary test cases using Taint-Aware In-Situ Simulation (TAISS), a novel technique for avoiding architectural taint explosions that blind information leakage. Using TAISS, we then built RandOS, a microarchitectural leakage fuzzer that is capable of generating complex multi-privilege and multi-address space test cases that are akin to randomized operating systems. Relying on MileSan for leakage detection, RandOS effectively discovers many known and 19 new vulnerabilities on RISC-V CPUs of varying complexity. Some of these vulnerabilities are constant-time violations of single instructions, while others are transient execution vulnerabilities, where in several cases secret information flows across privilege boundaries.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback, Quentin Bordier for his contributions to virtual memory management in RandOS and the maintainers of the designs we tested for their support in understanding some of the vulnerabilities we discovered. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] 2018. CVE-2018-3639. Available from MITRE, CVE-ID CVE-2018-3639.
[2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017.* IEEE, 1691–1696.
[3] Arm. 2022. Speculative Processor Vulnerability.
[4] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. 2016. The rocket chip generator. *Tech. Rep. UCB/EECS-2016-17* (2016).
[5] Krste Asanovic, David A Patterson, and Christopher Celio. 2015. *The berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized risc-v processor.* Technical Report. University of California at Berkeley Berkeley United States.
[6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. 2019. SMoTherSpectre: exploiting speculative execution

through port contention. In *ACM SIGSAC*.
[7] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors. *arXiv preprint arXiv:2402.03704* (2024).
[8] J.V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX SEC*.
[9] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, et al. 2019. Fallout: Leaking data on meltdown-resistant cpus. In *ACM SIGSAC*.
[10] Katharina Ceesay-Seitz, Flavien Solt, Alexander Klukas, and Kaveh Razavi. 2025. Pathfinder: Constructing Cycle-accurate Taint Graphs for Analyzing Information Flow Traces. In *ICCAD 2025*.
[11] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μCFI: Formal Verification of Microarchitectural Control-flow Integrity. In *ACM CCS*.
[12] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *Proc. USENIX Secur. Symp.* 1–21.
[13] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, et al. 2020. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product. In *ISCA*.
[14] Alvise de Faveri Tron, Raphael Isemann, Hany Ragab, Cristiano Giuffrida, Klaus von Gleissenthall, and Herbert Bos. 2025. Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks. In *USENIX Security*.
[15] S. Dinesh, M. Parthasarathy, and C. Fletcher. 2024. ConjunCT: Learning Inductive Invariants to Prove Unbounded Instruction Safety against Microarchitectural Timing Attacks. In *IEEE SP*.
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.
[17] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. 2021. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 874–887.
[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. 955–972.
[19] Muhammad Monir Hossain, Nusrat Farzana Dipu, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. TaintFuzzer: SoC security verification using taint inference-enabled fuzzing. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.
[20] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P Mulligan, Gustavo Petri, Christopher W Fletcher, and Caroline Trippel. 2024. RTL2Mu PATH: Multi-Mu PATH Synthesis with Applications to Hardware Security Verification. *arXiv preprint arXiv:2409.19478* (2024).
[21] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware information flow tracking. *ACM Computing Surveys (CSUR)* (2021).
[22] Wei Hu, Jason Oberg, Janet Barrientos, Dejun Mu, and Ryan Kastner. 2013. Expanding gate level information flow tracking for multilevel security. *IEEE Embedded Systems Letters* (2013).
[23] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. SpecDoctor: Differential fuzz testing to find transient execution vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1473–1487.
[24] Tobias Jauch, Alex Wezel, Mohammad R Fadiheh, Philipp Schmitz, Sayak Ray, Jason M Fung, Christopher W Fletcher, Dominik Stoffel, and Wolfgang Kunz. 2023. Secure-by-construction design methodology for CPUs: Implementing secure speculation on the RTL. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.
[25] jooohndoe. 2023. Github issue '[BUG] Multiple compressed instructions and the MV instruction create time side-channels in CVA6'. Retrieved November 13,2024 from https://github.com/openhwgroup/cva6/issues/1547
[26] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *NDSS*.
[27] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: practical dynamic data flow tracking for commodity systems. In *VEE*.
[28] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *IEEE SP*.
[29] Kronos. 2024. *Kronos RISC-v (All Cascade Fixes Integrated)*. Retrieved November 13,2024 from https://github.com/cascade-artifacts-designs/cascade-kronos
[30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).
[31] LLVM. 2025. *DataFlowSanitizer*. Retrieved April 10, 2025 from https://clang.llvm.org/docs/DataFlowSanitizer.html
[32] G Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *ACM SIGSAC*.
[33] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2014. Leveraging gate-level properties to identify hardware timing channels. *TCAD* (2014).
[34] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX SEC*.
[35] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. 2024. GhostRace: Exploiting and Mitigating Speculative Race Conditions. In *USENIX Security*.
[36] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. 2021. Crosstalk: Speculative data leaks across cores are real. In *IEEE SP*. Institute of Electrical and Electronics Engineers Inc.
[37] Mohammad Rahmani Fadiheh. 2022. *Unique Program Execution Checking: A Novel Approach for Formal Security Analysis of Hardware*. Ph. D. Dissertation. Technische Universität Kaiserslautern.
[38] M. Rostami, S. Zeitouni, R. Kande, C. Chen, P. Mahmoody, J. Rajendran, and A. Sadeghi. 2024. Lost and Found in Speculation: Hybrid Speculative Vulnerability Detection. *DAC 2024* (2024).
[39] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. 2019. ZombieLoad: Cross privilege boundary data sampling. In *ACM SIGSAC*.
[40] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. 2019. Netspectre: Read arbitrary memory over network. In *ESORICS*.
[41] Asia Slowinska and Herbert Bos. 2009. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*. 61–74.
[42] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU fuzzing via intricate program generation. In *USENIX Security 2024*. 1–18.
[43] Flavien Solt, Ben Gras, and Kaveh Razavi. 2022. CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL. In *USENIX Security*.
[44] Flavien Solt, Patrick Jattke, and Kaveh Razavi. 2022. RemembERR: Leveraging Microprocessor Errata for Design Testing and Validation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1126–1143.
[45] Flavien Solt and Kaveh Razavi. 2024. HybriDIFT: Scalable Memory-Aware Dynamic Information Flow Tracking for Hardware. *ICCAD* (2024).
[46] Flavien Solt and Kaveh Razavi. 2025. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. (2025).
[47] J. Stecklina and T. Prescher. 2018. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).
[48] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. 2025. RTL Verification for Secure Speculation Using Contract Shadow Logic. *ASPLOS* (2025).
[49] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *ASPLOS*.
[50] Daniel Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing new attack surfaces with training in transient execution. In *USENIX Sec*.
[51] Jo Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE SP*.
[52] S. Van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. 2019. RIDL: Rogue in-flight data load. In *IEEE SP*.
[53] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. 2021. CacheOut: Leaking data on Intel CPUs via cache evictions. In *IEEE SP*.
[54] Zilong Wang, Gideon Mohr, vom Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-Channel Security for Open-Source Processors via Leakage Contracts. In *ACM CCS*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.).
[55] Andrew Waterman and Krste Asanovic. 2023. *The RISC-V Instruction Set Manual*. Retrieved June 4, 2023 from https://github.com/riscv/riscv-isa-manual
[56] Johannes Wikner and Kaveh Razavi. 2022. {RETBLEED}: Arbitrary speculative code execution with return instructions. In *USENIX Sec*.
[57] Johannes Wikner and Kaveh Razavi. 2025. Breaking the Barrier: Post-Barrier Spectre Attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*.
[58] Johannes Wikner, Daniel Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting decoder-detectable mispredictions. In *IEEE/ACM MICRO*.
[59] Florian Zaruba and Luca Benini. 2019. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE VLSI* (2019).

RISC-V	Cell	IFT rule
(C).L(U)I	wire	A^t
(C.)ADD(I/W/16SP/4SP)	add	$[(A \wedge \bar{A^t}) + (B \wedge \bar{B^t})] \oplus [(A \vee A^t) + (B \vee B^t)] \vee [A^t \vee B^t]$
(C.)SUB(I/W)	sub	$[(A \vee A^t) - (B \wedge \bar{B^t})] \oplus [(A \wedge \bar{A^t}) - (B \vee B^t)] \vee [A^t \vee B^t]$
(C.)SL(L/A/I/W)	wire	$[A^t << B] \vee \{B^t\}$
(C.)SR(L/A/I/W)	wire	$[A^t >> B] \vee \{B^t\}$
(C.)SLT(U/I/W)	lt	$[(A \vee A^t) < (B \wedge \bar{B^t})] \oplus [(A \wedge \bar{A^t}) < (B \vee B^t)]$
(C.)XOR(I)	xor	$A^t \vee B^t$
(C.)OR(I)	or	$[(A^t \wedge \bar{B}) \vee (B^t \wedge \bar{A})] \vee [A^t \vee B^t]$
(C.)AND(I)	and	$[(A^t \wedge B) \vee (B^t \wedge A)] \vee [A^t \vee B^t]$
AUIPC(I)	add	$[(A \wedge \bar{A^t}) + B] \oplus [(A \vee A^t) + B] \vee A^t$
MUL(H/S/U/W)	mul	$\{A^t\} \vee \{B^t\}$
DIV(U/W)	div	$\{A^t\} \vee \{B^t\}$
REM(U/W)	div	$\{A^t\} \vee \{B^t\}$
(C.)S(B/H/W/Q/SP)†	wire	A^t
(C.)L(B/H/W/Q/SP)†	wire	A^t

Table 6: Correspondance between RISC-V instructions, the CellIFT [43] shadow logic cells and the taint propagation rules. A, A^t, B, B^t are the cell inputs and their corresponding taints. $\bar{I}, I^t$ and $\{I\}$ denote the negation, taint vector and extended vector of I. The extended vector $\{I\}$ of I where $I, \{I\} \in \{0,1\}^n$ is defined as $\{I\}_{i \in [0,n-1]} = \bigvee_{i=0}^{n-1} I_i$. *The architectural taints of the PC is the all-zeros vector at all times, therefore $B^t = 0$. †Load and store instructions do not allow tainted addresses.

A Reduced PoCs

The following provides PoC snippets obtained through reduction as described in Section 6.

A.1 TLBleed on BOOM (CVE-2025-29343)

```
...
0x28778: remu a2,s0,gp
0x28780: rem a2,a2,tp
0x287c8: divw ra,gp,a2
# Branch below mispredicted taken.
0x287d0: blt gp,ra,0x28e00
...
# Meltdown gadget executed transiently.
0x28e00: lb sp,-1587(t1)
0x28e04: lh s1,-1010(sp)
```

Listing 4: Reduced program that leaks kernel memory through the TLB on BOOM.

A.2 Transient MDS on CVA6 (CVE-2025-46004)

```
...
# Loads privileged data from S-mode.
0x301c223888: lbu ra, 757(s0)
...
# Calls debugging environment.
0x301c15aa5c: ebreak
# Transiently samples privileged data.
0x301c15aa60: lh a6, 2041(t2)
# Dependent load encodes privileged data in TLB.
0x301c15aa64: lhu t0, 1022(a6)
```

Listing 5: Reduced program that leaks kernel memory through MDS on CVA6.

A.3 MDS on CVA6 (CVE-2025-46004)

```
...
# S-mode loads privileged data.
0x82815c5f0: lbu t2, -660(s3)
...
# U-mode triggers page fault.
0x828138e28: lw gp, 1652(s2)
# Dependent load encodes the secret in the TLB.
0x828138e2c: lb t1, -1384(gp)
```

Listing 6: Reduced program that leaks kernel memory through MDS on CVA6.

A.4 Cross-privilege Spectre-SLS on CVA6 (CVE-2025-29340)

```
...
# Load some privileged data from S-mode.
0xa244c: lw s2, 456(ra)
...
# SPP=1, so we remain in S-mode.
0xa2dd8: sret
...
# The store executes transiently
# and leaks privileged data to the TLB.
0xa2ddc: sb t0, -1338(a0)
...
# SPP=0, so we go to U-mode.
0xd0f50: sret
...
# A store now leaks from the TLB.
0x67b50: sh t0, -754(t2)
```

Listing 7: Reduced program that leaks kernel memory through SLS on CVA6.