ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks

Alvise de Faveri Tron
Vrije Universiteit Amsterdam

Raphael Isemann
Vrije Universiteit Amsterdam

Hany Ragab
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Klaus von Gleissenthall
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

Abstract

Transient execution vulnerabilities have affected CPUs for the better part of the decade, yet, we are still missing methods to efficiently uncover them at the design stage. Existing approaches try to find programs that leak explicitly defined secrets, sometimes including the transmission over a side-channel, which severely restricts the space of programs that can trigger detection. As a result, current fuzzers are forced to constrain the search space using templates of known vulnerabilities, which risks overfitting. What is missing is a general detection mechanism that (1) makes it easy for the fuzzer to trigger a violation and (2) catches vulnerabilities at their root cause — similarly to *sanitizers* in software.

In this paper, we propose Phantom Trails, an efficient yet generic method for discovering transient execution vulnerabilities. Phantom Trails relies on a fuzzer-friendly detection model that can be applied without the need for templating. Our detector builds on two key design choices. First, it concentrates on finding microarchitectural data leaks independently of the covert channel, thereby focusing on the core of the attack. Second, it automatically infers all secret locations from the architectural behavior of a program, making it easier for the detector to find leaks. We evaluate Phantom Trails by fuzzing the BOOM RISC-V CPU, where it finds all known speculative vulnerabilities in 24-hours, starting from an empty seed and without pre-defined templates, as well as a new Spectre variant specific to BOOM — Spectre-LoopPredictor.

1 Introduction

Transient execution attacks are a critical security threat that has plagued CPUs for the best part of the last decade. After the initial discoveries of Spectre [39] and Meltdown [42], recent years have brought on a variety of new attacks [6, 40, 44, 46, 52, 65, 66, 70]. Once discovered, these issues are unfortunately not easy to fix: post-silicon mitigations have often proven to be either incomplete [6, 13, 45, 69, 71], opening the door for new attacks, or so detrimental to performance as to render them impractical [28]. Ideally, such vulnerabilities should be found, and fixed, at the *pre-silicon* stage, i.e., during the design phase of the CPU. However, automatically detecting them in hardware designs is challenging.

Pre-Silicon Fuzzing. While exhaustive approaches such as formal verification [19, 21, 26, 67] are difficult to scale to real-world CPUs, a promising approach for finding hardware bugs in RTL designs is *fuzzing*, which has been applied to both architectural [11, 34, 38, 59, 64, 72] and microarchitectural [25, 33] bugs. For CPUs, pre-silicon fuzzing generally requires to iteratively generate random inputs (i.e., programs) and verify their behavior on a cycle-accurate simulation of the Design Under Test (DUT). This approach poses some unique challenges when compared to traditional software fuzzing – especially when looking for transient execution vulnerabilities. First, the size of the input space – the space of all possible programs, initial memory states, and CPU configurations – paired with the complexity of the designs and the slow speeds of cycle-accurate simulations make efficient fuzzing hard. Second, hardware does not inherently "crash", which raises the problem of how to detect violations during fuzzing. Transient execution vulnerabilities represent a further challenge: while architectural bugs can be detected through HDL assertions or golden reference models, modelling transient vulnerabilities at the RTL level is still an open problem.

Problem Statement. Current state-of-the-art fuzzers for transient vulnerabilities address the problem of navigating the huge search space by employing some form of *templating*, i.e., by either (1) breaking up known end-to-end attacks into individual stages that serve as a blueprint for creating new variants [33], or (2) providing the fuzzer with program snippets such as "try to access a secret" or "slow down an instruction" to mimic the behavior of known PoCs [25]. While these restrictions help make the search practical, they bias the fuzzer towards known issues, which risks overfitting. Our main insight is that current fuzzers need restrictions like templates because their underlying detection models overly constrain the space of programs that can trigger a violation.

In particular, current detection models rely on *explicitly defined secrets*, i.e., values that should not be leaked by the microarchitecture, which are defined as all data residing in specific memory regions protected by specific hardware flags [25, 33, 60]. On top of this, state-of-the-art fuzzers [33] only detect violations *after* a secret is transmitted through a covert channel, thereby requiring full end-to-end attacks. Both choices make life for the fuzzer unnecessarily hard.

Phantom Trails. With Phantom Trails, we present a new approach to efficiently finding transient vulnerabilities without templates or smart seeds. Phantom Trails builds on a fuzzer-friendly detector which imposes fewer constraints on the programs and detects violations at their root cause. Instead of end-to-end exploits, Phantom Trails concentrates on finding *transient data leaks* — ways in which secrets can enter the microarchitecture through transient execution — independently of the side-channel transmission, providing early detection of vulnerabilities. Unlike previous methods, our detection model *implicitly* defines secrets specific to a program by deriving which memory locations are never accessed architecturally. Taking a key insight from software sanitizers such as ASAN [57] and MSAN [61], whose implicit "tainting" of most of a program's memory greatly increases the probability of detecting memory errors, Phantom Trails's tainting of *all* memory not accessed architecturally maximizes the likelihood of finding violations. This allows to model a wide variety of vulnerabilities including Spectre-v1, Spectre-v2, Spectre-RSB, Spectre-SSB, and Meltdown variants.

Evaluation. To demonstrate the practical benefits of our approach, we run a fuzzing campaign on BOOM [5], a popular open-source RISC-V core equipped with an out-of-order pipeline and speculation. On BOOM, Phantom Trails is able to reliably detect all Spectre and Meltdown variants known on this core within 24 hours without the need for templating, unlike the state of the art [25, 33]. Phantom Trails also uncovers Spectre-LoopPredictor – a new Spectre variant, specific to BOOM, through which an attacker can cause mispredictions on an *uncontrolled branch* by training a nearby control-flow instruction. We disclosed Spectre-LP to the maintainers of BOOM, who acknowledged the issue.

Contributions. We make the following contributions:

1. We describe a new, fuzzer-friendly detection model offering sanitizer-like functionality for transient execution vulnerabilities in CPU designs;
2. We build an extensible, software-only detector based on LLVM and Verilator to enforce our model on CPU simulations, with minimal knowledge of the DUT and no hardware modifications;
3. We integrate the detector into an open-source fuzzer that can find Spectre and Meltdown samples within 24 hours on BOOM without templating or smart seeds;
4. We uncover a new speculation primitive on BOOM (Spectre-LP) that can be used to mispredict an uncontrolled branch towards a disclosure gadget.

Open Sourcing. All the source code of our detector, including our LLVM instrumentation for Verilator (BFSan) and a taint-tracking wrapper for BOOM, is available at `https://zenodo.org/records/14726711`, along with all transient leak experiments and fuzzing infrastructure.

2 Background

In this section, we briefly recap the nature of transient execution attacks and existing detection models.

2.1 Transient Execution Attacks

Phases. End-to-end transient execution attacks consist of three main phases: ① a *priming* step, in which an attacker massages the microarchitecture to a vulnerable state, ② a *secret access* step, in which the CPU transiently accesses some secret data as a result of the attacker's priming, and ③ a *transmission* step, in which the secret is first encoded into a non-transient microarchitectural state (e.g., the cache) and then recovered into an architectural value by the attacker.

Classification. In Meltdown-like attacks, the attacker accesses data belonging to a *different* security domain through a faulting instruction. In particular, in Meltdown a faulty attacker load brings victim data from the L1 cache into the pipeline, while the faulting load in an MDS attack accesses *in-flight* data belonging to the victim. In contrast, in Spectre-like attacks the access occurs in the *same* security domain, by a victim acting as a confused deputy, while the transient window is generated through speculation. Figure 1 shows the phases of a typical Spectre attack employing Flush+Reload [73].

Covert Channels. Step ③ transmits the secret via a timing covert channel. Many such covert channels have been discovered over the years, including those based on caches [73], translation structures [29, 62], prefetchers [30], predictors [3, 20], contention on execution units [8], etc.

2.2 Existing Detection Models

Previous work has proposed different models for detecting transient execution vulnerabilities at the pre-silicon stage.

Templating. IntroSpectre [25] and SpecDoctor [33] are pre-silicon *fuzzers* that are aimed at transient execution vulnerabilities. Due to the complexity of the DUT, they both employ strategies to restrict the search space. IntroSpectre defines a set of "gadgets", i.e., snippets of code taken from known vulnerabilities (such as "M5 – Generate store and load instructions with overlapping addresses.") and combines them.

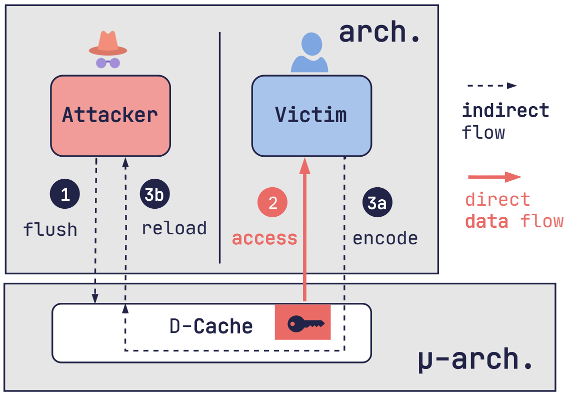


Figure 1: Phases of a Spectre attack employing Flush+Reload. The attacker first primes the microarchitecture by flushing the cache ①, then forces the victim to transiently access a secret ②, the value of which gets encoded in the microarchitectural state (here the cache) which leaks to the attacker by means of subsequent timed loads ③.

SpecDoctor uses multi-phased fuzzing starting from a predefined template that mimics the different phases of known attacks and tries to fill them until an end-to-end leakage is found, including the transmission and recovery through a covert channel.

Secret Tracking. Detection mechanisms for transient vulnerabilities generally involve tracking a *secret* through the microarchitecture. In particular, IntroSpectre uses a secret value generator to populate secret memory with specific values, and triggers detection if such values are found in a microarchitectural buffer (e.g. the Line-Fill Buffer). SpecDoctor uses differential testing by changing the values of secret memory between two different runs of the same program and checking if the hash of the microarchitectural state differs. STT [74] and CellIFT [60] propose a different approach for detection (but not in the context of fuzzing) that uses hardware Information Flow Tracking, or *taint tracking*, to precisely follow the flow of secret data during its manipulation.

Secret Definition. All existing detection approaches rely on defining *secrets*. STT [74] is a microarchitectural defense that considers all speculatively-accessed data as secret, until the corresponding instruction is past a Point-of-No-Return in the RoB, by which time the data is considered architectural. This approach is not suitable for fuzzing, as *any* speculative window would trigger a violation, even those where the speculation turns out to be correct. All other approaches use *explicitly defined secrets*. CellIFT and SpecDoctor start from a predefined secret memory region, which is isolated using hardware primitives (PMP or page flags). IntroSpectre also uses page flags to identify secrets, but allows them to evolve based on the permissions changes operated by the gadgets.

3 Challenges and Observations for Fuzzing

In this section, we highlight some of the obstacles that existing pre-silicon detectors and fuzzers for transient vulnerabilities face, as well as key insights to overcome them.

Sources of Entropy. To generate a program that uncovers a transient vulnerability, fuzzers need to beat a variety of entropy sources. First, the fuzzer needs to generate a set of valid instructions, and a program that exhibits some non-trivial control and data flow. Next, the program needs to open a speculative window. On top of this, the program needs to access a memory location containing a secret during speculation. Finally, in the case of SpecDoctor, the program also needs to encode the secret into the microarchitecture, and the fuzzer needs to generate the receiver code that extracts the secret. Creating programs that follow all of these steps is a considerable effort for a fuzzer, and makes efficient fuzzing impractical. Existing fuzzers tackle this complexity through *templating*, which aims at reducing the entropy of program generation. While this approach speeds up fuzzing, it risks overfitting on known vulnerabilities. We observe that, by concentrating on other sources of entropy, we might be able to significantly speed up fuzzing without the need for templates.

Observation #1: To generate samples of transient execution attacks, fuzzers must beat a variety of **entropy sources**. By focusing on sources other than program generation, we can eliminate the need for templates.

Indirect Flows. Our second observation stems from analyzing the different steps of transient execution attacks in Figure 1. We observe that in the priming step (step ①) the attacker massages the microarchitecture indirectly, i.e., performs actions that modify the content of prediction structures, *without* directly accessing their content (which is not available architecturally). Similarly, in step ③, the victim modifies the microarchitectural state in a secret-dependent way, but there is no direct flow of information between victim and attacker. In contrast, in step ② (secret access) there is a direct data flow between secret data and some microarchitectural buffer, e.g., the *Register File* or the *Line-Fill Buffer*. A key observation is that, while indirect flows are a known issue for taint tracking frameworks and can often lead to overtainting, direct data flows can be precisely tracked—making the secret access step an ideal place to catch speculative attacks. Moreover, as the secret access happens *independently* of the transmission and recovery step, it is *orthogonal* to the side-channel being used. Focusing on the secret access step therefore targets the core of the attack, reducing fuzzer entropy.

Observation #2: We can remove the **entropy of the side-channel** by focusing on the secret access phase, where we have a direct data flow of the secret.

Secret Model. A core challenge of defining transient execution attacks at the RTL level is modelling secrets. Existing techniques rely on *explicitly defined secrets*—for instance, by marking some pages as secret [33, 60]. Explicit secret models restrict the number of speculative accesses that trigger detection, making it harder for the fuzzer to find a violation. Moreover, they require the detector to commit to *specific threat models*. For example, attacks can leak data across hardware-defined boundaries (e.g., user code reading supervisor memory) or software-only boundaries (e.g., JavaScript programs breaking website isolation). Similarly, the secret may be read directly from within the attacker context (Meltdown), or through the victim (via a gadget in the victim code), and then exfiltrated by the attacker (Spectre). Approaches based on explicit secrets protected with PMP/Page Flags must explicitly pick an attacker model before fuzzing [33], and cannot account for leakage across software-only boundaries.

> **Observation #3:** By avoiding explicit secrets we can greatly reduce the **entropy of the secret** address (and possibly catch same-domain leaks).

4 Phantom Trails

We now discuss how Phantom Trails addresses these fuzzing challenges, based on our observations.

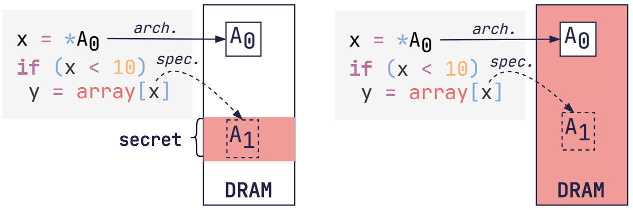
4.1 Transient Data Leaks

While detecting end-to-end leaks is a challenging task and represents a considerable obstacle for fuzzing, detecting secret accesses, which happen before and independently of the side-channel transmission, can be achieved with precise taint-tracking, and catches vulnerabilities at their root. In particular, given a program to test, we track all data flows through the DUT by accessing the RTL-level design and applying taint tracking to the cycle-accurate simulation of the CPU. This includes *speculative* data flows, e.g., speculative loads, which are visibile in the microarchitecture for a restricted period of time (until the speculation is squashed) but not from the architectural execution. Such data flows can move secret data from the memory subsystem to an exposed buffer inside the CPU, i.e., a *taint sink*, such as the Register File. Once the secret has entered the RF, it can be leaked in a variety of ways, for example, by a subsequent load or a variable-time instruction.

We call direct leaks from secret memory to exposed buffers *transient data leaks*, and focus our detection method on them.

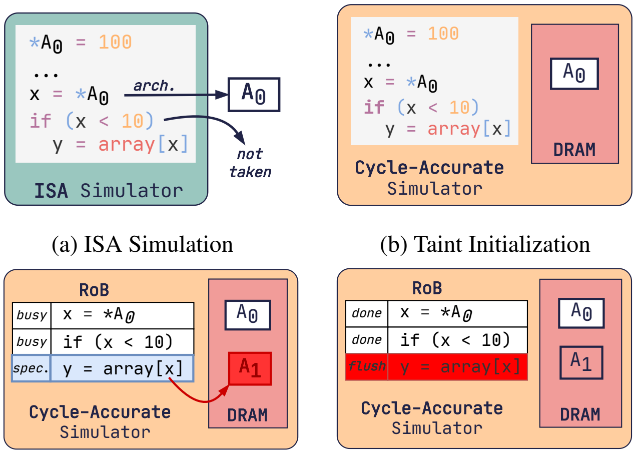
4.2 Implicit Secrets

Instead of relying on explicit secrets, which make it hard for the fuzzer to find vulnerabilities and risk missing attacks, we introduce the concept *implicit secrets*, depicted in Figure 2. Given a stream of instructions executed by the CPU, we can derive the set of all memory locations that should be accessed architecturally. We use this intuition to define our notion of *implicit secrets*: all data that is *not* accessed architecturally by a program is considered a secret. During fuzzing, we generate programs that start from the same initial state and can run for a maximum number of cycles proportional to the size of the binary (see Section 5.2). We infer secrets by first executing a generated program on an ISA simulator, such as Spike [2], which recovers the list of architectural accesses for a single run, and then taint *every other memory location* in the simulated DRAM before running the same program from the same initial state on the microarchitectural simulator.



(a) Explicit Secrets (b) Implicit Secrets

Figure 2: A visualization of the difference between explicit secret models ⓐ and implicit secret models ⓑ (red is secret).



(a) ISA Simulation (b) Taint Initialization

(c) Taint Detected (d) Pipeline Flush

Figure 3: Different phases of our detection model.

Example. Figure 3 represents an example of how a Speculative Bounds Check Bypass (Spectre-v1) can be detected by our model. First, we run a sequence of instructions with an ISA simulator (*Phase* ⓐ) and infer the set of architecturally-accessed locations ($\{A_0\}$). Then, we taint every other location in the simulated DRAM as "secret" (*Phase* ⓑ). Finally, on the cycle-accurate simulation of the CPU, we observe taint coming from A_1, which was never loaded architecturally, inside of the Register File (*Phase* ⓒ), which triggers detection.

4.3 Flush-Based Classification

The transient nature of the vulnerabilities we are looking for implies that the instruction accessing the secret is *speculative*, and therefore has to be squashed when the speculation is revealed to be incorrect. Microarchitectures typically have at least three ways to signal that an instruction has to be squashed: (1) *pipeline exceptions*, generated by faulty instructions (e.g., loads that cause a page fault), (2) *mispredictions* that indicate incorrect control-flow speculation, (3) and *rollbacks*, which might happen on value speculation, e.g., with store-to-load forwarding. We can use these signals for classification: whenever the microarchitecture brings a secret into a sink, instead of immediately crashing the execution, we wait until one of such signals is detected (*Phase* ⓓ in Figure 3) and perform a preliminary classification of the leak based on it. If no pipeline flush is detected before the end of the program, we report an unidentified leak. This might indicate either the presence of an additional, unidentified flush signal, or an architectural bug that leaks transiently-accessed data.

4.4 Tainting Software Simulations

The implementation of our detector has two major requirements: (1) we need a taint-tracking engine to track secrets in the microarchitecture (2) we need easy access to the microarchitectural state during simulation, in particular the simulated DRAM, the Physical Register File (i.e., our sink) and any relevant component for classification (Re-Order Buffer, and signals indicating a pipeline flush). Additionally, for fuzzing, we need to instrument the simulation to gather *feedback*.

To tackle these requirements, we adopt an approach similar to Trippel et al. [64] and instrument the *software* cycle-accurate simulation of the CPU generated by Verilator [58], an open-source cycle-accurate simulator. With such approach, we can benefit from the power and maturity of existing software such as LLVM [41] and AFL [22, 23, 75]. More specifically, a software-only approach has the following advantages:

1. Reusing mature infrastructure widely adopted in academia and industry (LLVM, AFL) makes this approach compatible with past and future research/tooling on software fuzzing and vulnerability discovery

2. Using LLVM instrumentation for both taint tracking and fuzzing means that the same taint infrastructure can be used for both *detection* and fuzzing *feedback*

3. A software-only infrastructure makes it easier to prototype new detection mechanisms and taint policies, and guarantees easier scalability (does not require FPGAs)

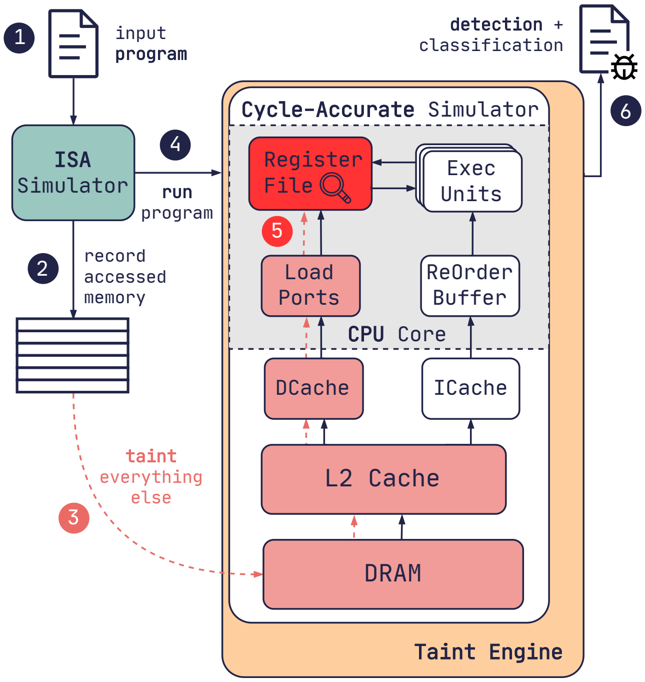


Figure 4: Structure of Phantom Trails' detector. Given an input binary ①, the detector runs an ISA Simulator to obtain a list of **architectural accesses** ②. Every address that is not in this list is tainted in the simulated DRAM ③. Then, the program is run through the **cycle-accurate** simulator ④, where taint is allowed to propagate until it reaches a **sink** ⑤. On the next pipeline flush, the detector will abort the execution with an error code and produce a detection report ⑥ that marks the input as problematic.

5 Detector Design

We now present our implementation of Phantom Trails' detection component, and evaluate its ability to correctly identify and classify PoCs of known vulnerabilities.

5.1 Components

Figure 4 represents an overview of the structure of Phantom Trails' detector. Similar to previous work, we use the BOOM RISC-V core [5] as the design-under-test for our prototype.

ISA simulator. To infer secret locations,we use a modified version of the RISC-V ISA simulator *Spike* [2] to architecturally simulate an input program. We modified Spike to log all memory locations (and instructions) accessed during simulation as well as the number of executed instructions. Additionally, we added the possibility of discarding test cases that hinder correct classification, such as self-modifying code (see Section 6.2). It is worth noting that, while our system is currently implemented for RISC-V architectures, ISA simulators exist also for other instruction sets.

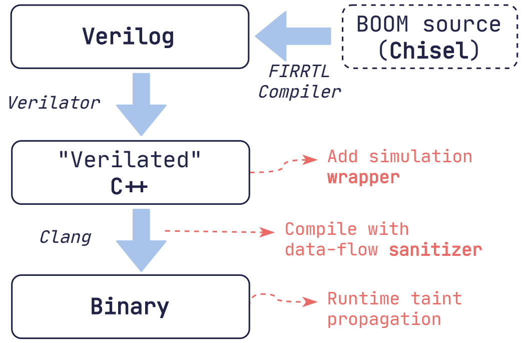


Figure 5: Compilation pipeline that transforms the BOOM design into an instrumented binary that can be used for fuzzing.

Taint tracking engine. Our prototype uses a custom taint tracking engine called *BFSan*, which supports bit-precise tracking of taint throughout a program. We only propagate taint through direct data flows. BFSan is built on top of the MemorySanitizer (MSan) error detector from the LLVM compiler infrastructure. Similar to MSan, it divides the memory space into two parts: normal program memory, and a *shadow map*, used to track the taint of each bit. BFSan follows the flow of taint by instrumenting a program during compilation. The instrumented code propagates the taint through the program by updating the contents of the shadow map on each executed instruction.

Cycle-accurate simulator. The cycle-accurate simulator used in our prototype is generated by the Chipyard [1] build system. The source code of BOOM is first lowered from Chisel [17] to Verilog, then translated by Verilator into a compilable C++ object that contains all simulation logic and whose members represent hardware registers and wires. Figure 5 shows an overview of the compilation process. Once the C++ object is generated, we identify the relevant components to monitor and execute the simulator through a software wrapper. The software wrapper applies the initial taint to the DRAM, advances the simulation clock, and monitors taint sinks for classification. The resulting C++ program is then compiled with BFSan, which adds logic for taint tracking.

5.2 Challenges

Termination. Since hardware is reactive, that is, it does not terminate as long as a clock signal is provided, we face the problem of deciding *when to stop* the simulation for a given program. We employ the following strategy:

1. During *architectural* simulation (Spike), we terminate on any faulty instruction, thus ensuring we only execute the loaded program. In particular, we initialize all DRAM locations outside the loaded program to 0 – an illegal instruction in RISC-V – and ensure that our trap handler also contains an illegal instruction. This means that when the program reaches its end, the CPU will encounter an illegal instruction, which we use as a termination signal. Since our program may contain unbounded loops, we further put a bound on the total number of execution steps. If no illegal instruction is found, the simulation terminates after the maximum number of steps, which is calculated depending on the size of the input program.

2. During *cycle-accurate* simulation, we monitor the Re-Order Buffer (RoB) to count the number of retired instructions. Whenever this number matches the number of instructions reported by Spike, we end the simulation.

Taint sources. Since all the input program's code and data are loaded in memory at the start of each execution, we use DRAM as our initial taint source. In particular, we leverage BOOM's option to provide a *black-box* implementation for the simulated DRAM. We use the list of memory accesses generated by the ISA simulator to initialize taint. In particular, we apply taint to all DRAM locations that have *not* been accessed architecturally. This includes locations whose initial value is overwritten by a subsequent store before being read.

Taint sink(s). The simulation wrapper monitors the presence of taint in a predefined sink after each clock cycle. For our prototype, we chose the Physical Register File (PRF) as sink for two main reasons: (1) non-architectural data reaching the PRF can be leaked through a variety of side-channels, e.g., port contention, cache, TLB; (2) if tainted data reaches a physical register, we can easily infer which instruction is responsible for it by inspecting entries in the RoB.

Taint washing. If a program speculatively *jumps* to a tainted value rather than loading it, taint might end up in the Register File. While this correctly implies that speculative code is being executed, we only care about speculative code that *brings new data into the Register File*, like Spectre gadgets for example. To avoid marking speculative code that does not directly leak values as a vulnerability, we make sure that taint is washed for instructions passing through the instruction cache, which prevents taint from spreading to the RF.

Self-modifying code. Differently from x86, RISC-V architectures do not guarantee that the instruction cache is invalidated if code is modified during execution, and instead require explicit synchronization from software through FENCE.I. Programs that modify cached instructions without flushing the I-Cache are expected to produce a different behavior than the ISA simulation. For our use-case, this means that any program that modifies a load (e.g., by turning it into a `nop`), will still observe the microarchitectural effects of that load, while the ISA simulation will not. To avoid reporting such cases, we detect programs that contain self-modifying code during the ISA simulation, and immediately discard the program without wasting time on the slow cycle-accurate simulation.

5.3 Classification

To aid the analysis of the reported leaks, we perform an initial classification of the bug using the *pipeline flush* signal. In particular, instead of aborting the simulation immediately when taint reaches an exposed sink, the simulation continues executing the program and records:

1. The *Taint Event*, i.e., when taint is first observed in the Register File. We refer to the instruction responsible for this event as the *tainting instruction*, which can be derived by observing the Re-Order Buffer.

2. The *Flush Event*, i.e., a flush signal that squashes the taint instruction. We refer to the instruction that triggers the flush event as the *flushing instruction*.

The simulator finally crashes whenever it detects that a pipeline flush is about to "remove" (squash) the tainting instruction from the pipeline. If taint is found in a sink but the corresponding instruction is never squashed, the crash is generated at the end of the test-case execution, i.e., when all the architecturally-executed instructions have retired from the pipeline, and the test case is marked as *Unknown Flush*.

We use this information to perform a preliminary classification of the violation found. In particular, by observing the flush signal we can distinguish between Spectre violations (mispredictions), Meltdown violations (pipeline exceptions), and memory ordering faults (**Spectre-v4**). For Spectre variants other than Spectre-v4, we observe the flush instruction to determine if the misprediction was caused by a branch (**Spectre-v1**), indirect jump (**Spectre-v2**) or return (**Spectre-RSB**). For pipeline exceptions, we check if taint was introduced by the flush instruction itself (**Meltdown**) or by a younger instruction in the pipeline (**OOO - Out-Of-Order**).

Finally, for branch mispredictions, we further report if the branch was predicted taken or not-taken, and if the tainting instruction was architecturally executed at least once before the taint detection. This allows us to differentiate between Spectre-v1-static (predicted not-taken, new instruction), Spectre-v1-training (predicted taken, previously executed instruction), and Spectre-v1-new (predicted taken, new instruction).

5.4 Extensions

MDS Detection. MDS attacks, such as RIDL [66] and Fallout [13] and derivative attacks such as LVI [65] and CrossTalk [53], showed that, on Intel microarchitectures, an attacker can leak in-flight data from Line-Fill Buffers, Load Ports, and Store Buffers. Differently from traditional Spectre and Meltdown attacks, these vulnerabilities incorrectly access values in *internal CPU buffers*, as opposed to secrets in memory. These vulnerabilities can be modeled in Phantom Trails by adding such internal buffers as *taint sources*. In particular, we extended our prototype with a simple user-space initialization snippet (a set of loads and stores) that runs right before the start of the program under test, without any fence. Once the last instruction of the initialization snippet retires, Phantom Trails taints the initial values of all internal buffers, while the user program is ready to start. If the program is able to leak such stale values, e.g., through a faulty load, their taint will be observed in the Register File. Note that the user program is not allowed to directly access such values, so, whenever they are leaked, we are sure that there is a violation. As BOOM is not vulnerable to MDS, to test this setup we added a simple MDS-StoreBuffer vulnerability, as described by the Fallout [13] paper, to the BOOM core design, and verified that the leakage is detected. For the benefit of future research, we open-source the patch for adding the vulnerability to BOOM.

Secure Speculation. Phantom Trails can be extended to incorporate knowledge of both software and hardware defenses. For instance, the instruction generator can be constrained to always emit an `LFENCE` [37] after each branch, to mimic cases where this mitigation is deployed. For secure speculation defenses such as STT [74], finer-grained detection policies can be added for taint sinks, e.g., discarding tainted entries that are read by instructions deemed "safe" by STT.

Other Taint Sources. Similarly to MDS, other data sampling attacks such as Gather Data Sampling [46], AEPIC Leak [10] and ZenBleed [51] have been shown to be possible on x86 cores. In particular, Downfall [46] shows that the `gather` instruction can transiently leak stale data from a temporal buffer called the *SIMD register buffer*, confirmed by Intel. AEPIC Leak and ZenBleed instead can read stale data *architecturally* from the superqueue (buffer between L2 and LLC) and XMM registers, respectively. Similarly to the MDS case, Phantom Trails can be extended to handle more taint sources by making sure such internal buffers are initialized and tainted right before the start of the program. For initial taint residing in the Register File, more fine-grained taint sink policies can be applied to ignore specific initially-tainted locations until they are overwritten by another operation.

6 Fuzzing

This section describes how we integrated Phantom Trails' detector into a pre-silicon fuzzer, and highlights the benefits of our detection model to the fuzzing use-case.

6.1 Overview

Phantom Trails resembles a traditional greybox fuzzer that exercises a software representation of the hardware as the DUT [64]. Figure 6 presents a high-level overview of its components. In particular we used the setup described in Section 7.1 as the DUT in our fuzzing campaigns, which includes

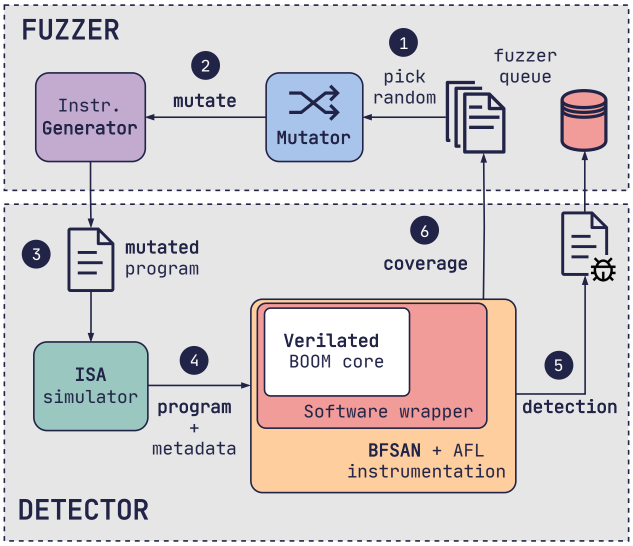


Figure 6: Phantom Trails's fuzzing cycle. A sequence of instructions is picked randomly from the **fuzzer queue** ① and modified by the **mutator** ②. The resulting sequence is then translated into a RISCV program ③ and executed by the **ISA simulator**. If the program is not discarded, the metadata is passed to a **cycle-accurate simulator** ④ where, in case of detection, the program will be saved ⑤. If the program execution produced new coverage in the simulator, it is also added to the fuzzer queue ⑥.

a minimal setup for the BOOM core in its MEDIUMBOOM configuration and a black-box DRAM module.

Fuzzing infrastructure. We build our fuzzer on top of the state-of-the-art `libafl` [23] fuzzing framework and run it in *fork mode*—forking after the completion of a hardware reset to avoid the cost of restarting the simulation on each input. To adapt the `libafl` software-based infrastructure to hardware designs, we developed a set of components suitable for generalized hardware fuzzing. Our entire infrastructure is open-source and available at `https://github.com/vusec/phantom-trails`.

DUT warmup. Before any code is run, the hardware simulation is reset through the default Verilator wrapper for BOOM by asserting the `reset` signal for 100 cycles.

Boot phase. Upon reset, execution starts from the content of the boot ROM, which we modify to simply jump to the first DRAM address. At the beginning of DRAM we place our initialization code, which is responsible for:

1. Setting up the *trap handler*, which in our case simply contains an illegal instruction to terminate simulation;
2. Configuring the Physical Memory Protection (PMP) unit to permit access to all memory;
3. Setting up *page tables* and enabling virtual memory management; in particular, we map a contiguous set of pages starting from the beginning of DRAM with different page flags;
4. Optionally, initializing register values (optimization **D2**);
5. Jumping to U-mode (code with user privileges), where the input program is located;

Predictors initialization. By default, the BOOM processor initializes all entries of the Bi-Modal Table (`BIM`) to 2 on reset, which corresponds to the "weakly taken" state. While this does not prevent detection, it can cause the classifier to mistake cases of static branch prediction for cases where the branch was trained, and incorrectly label Spectre-v1 samples. To ensure a correct fine-grained classification, we modify the BIM initialization procedure to instead set entries to the "not-taken" state.

Taint initialization. While in supervisor-mode, the initialization code reads from the supervisor data region, which fills the D-Cache with tainted (supervisor) data. As discussed in Section 5.4, and optional user-mode initialization can be performed for MDS to fill internal CPU buffers, such as the Load Queue or the Store Buffer, with tainted data as well, right before the start of the generated program.

6.2 Program Generation

As stated in section 3, one of the entropy sources that the fuzzer has to beat is program generation. While sophisticated approaches [59] can be added on top of our fuzzer, in this paper we want to demonstrate that our detector already helps even with a minimal program generator. In particular, in our prototype we make sure to generate and mutate syntactically valid RISC-V instructions, and we adopt a set of minimal optimizations to increase the chance of generating complex control and data flow. Such optimizations differ from templates, as they are aimed at maximizing the odds of generated well-formed, complex programs rather than following the blueprint of a specific vulnerability.

6.2.1 Instructions

Mutator. Phantom Trails' custom program mutator is aware of what constitutes syntactically valid RISC-V instructions, but possesses no further (semantic) information about them. In its basic form, it generates instructions by choosing a random RISC-V instruction type and applying a random mutation operation. In particular, the current prototype supports inserting a new instruction, replacing an instruction with a new one, replacing the argument of an existing instruction, repeating an existing instruction, deleting an existing instruction, replacing an instruction with a nop, and swapping two instructions. Optionally, the mutator can be biased towards emitting jalr and ret instructions, and towards reusing previous values when choosing arguments, as we will discuss in Section 6.2.

Program generator backend. Since applying random mutations like bit-flips at the assembly level has a high chance of generating invalid programs which would waste precious simulation time, the fuzzer instead uses a structured internal representation to apply mutations. Programs stored in this internal representation are then translated into valid RISC-V programs by the instruction generator, before entering the detector component.

6.2.2 Optimizations

To avoid wasting simulation cycles on uninteresting inputs and to maximize the likelihood of finding bugs quickly, we develop a set of optimizations that bias our program generation towards valid programs. Unlike templates [33], the optimizations are general so as to avoid overfitting. We group our optimizations into: **Basic (B)**, biasing the generator towards reusing arguments, **Control-Flow (C)**, maximizing the probability of generating well-formed function calls, and **Data-Flow (D)** optimizations that increase the likelihood of using valid pointers (code and data).

B1 - Register reuse. With this optimization, when deciding on the argument of an instruction, the mutator has a bias towards selecting the registers used by previous instructions (e.g., a probability of 50% in the current prototype). The idea is to improve the chance of generating data flow between instruction sequences, as well as that of creating race conditions in the microarchitecture through aliasing.

B2 - Power-of-two constants. When picking immediate values, this optimization adds a bias towards powers of 2, which reduces the amount of entropy for constants and helps with alignment.

C1 - Indirect calls. To help the fuzzer reduce the entropy for indirect calls, this optimization adds the possibility of inserting one of two code snippets shown in Listing 1. Snippet 1 performs a well-formed indirect jump to a nearby address (i.e., a small offset from the current program counter); Snippet 2 emits a valid ret instruction, which, in RISC-V, is a pseudonym for an indirect jump to ra. While this does not guarantee the generation of a valid indirect call, it improves the chances of executing valid calls and returns. Note that more sophisticated approaches [59, 72] to program generation can build on top of such a basic optimization—albeit at the cost of additional overhead.

C2 - Discard invalid jumps. To further reduce the likelihood of wasting simulation cycles on programs with invalid control-flow, the ISA simulator terminates the execution immediately whenever a program jumps outside the range of valid memory locations, and discards the corresponding input program. This guarantees that, at least for jump and call instructions, Phantom Trails runs the expensive cycle-accurate simulation only if the program jumps to valid targets.

```
# Load current PC
auipc x2, 0
# Jump to PC + offset
jalr ra, rand_offset(x2)
```

(a) **Snippet 1:** Indirect call

```
# Return
jalr zero, 0(ra)
```

(b) **Snippet 2:** Return

Listing 1: Control-flow snippets inserted by the mutator.

D1 - Map address 0. Since most of the memory is filled with 0s at startup, and it is not uncommon for predictors to default to address 0 [36] on empty prediction structures, this optimization makes sure virtual address 0 is mapped to a valid memory page before the input program starts execution.

D2 - Initialize registers. This optimization ensures that some registers are filled with valid pointers before jumping to the input program's code. We fill half of the logical Register File with addresses of both code and data pages that have an associated page table entry. This means that, whenever an instruction uses a register for the first time there is a 50% chance that it will use one of the initialized pointers.

6.3 Feedback

Currently, there is no consensus on the best feedback metric for hardware fuzzing [59], nor is there is a "standard" strategy for transient execution fuzzers. As we want to show the advantages of our detection model on a simple fuzzer, we use as baseline feedback the standard coverage metric provided by the AFL++ software fuzzer, which we call '*SW Feedback*'. To evaluate additionally Phantom Trails's sensitivity to feedback metrics, we additionally implemented an alternative, taint-based feedback mechanism which is inserted into the cycle-accurate simulation via an LLVM pass, to explore the possibility of using taint as feedback.

SW Feedback. In this case, the metric is an approximation of the edge coverage of the system-under-test as described in previous work [64]. We adapted this metric by only counting whether an edge in the simulator has been executed at all, and not *how often* it was executed. Doing so avoids labelling mutations that merely traverse the same edges as interesting, while adding little relevance to the program.

Taint Feedback. Instead of tracking the edge coverage of the simulator during the input program's execution, this metric tries to measure how much taint has spread through the design—across all the CPU's wires. Since most Verilog-specific information, including the list of wires, is lost during Verilator's translation process, we identify the code for each wire from within the compiler pass indirectly, based on the observation that Verilator stores the contents of registers that persist between cycles in memory. We therefore approximate wires of the simulated CPU by considering every instruction that writes to non-stack memory. For each store, our compiler pass assigns a unique slot in the fuzzer's coverage map. The coverage map is scaled according to the number of identified wires and the bit size of each slot is at least as large as the number of bits in the respective wire. After allocating a slot in the coverage map, the pass injects code into the compiled simulation that copies and merges the current taint status of each wire with the taint data stored in the coverage map.

7 Evaluation

7.1 Detector Evaluation

Instrumentation Overhead. To measure the overhead introduced by our instrumentation, we generated a simulation of the MEDIUMBOOM core (2-wide, 64 RoB entries, 80 physical registers) using the CHIPYARD toolchain [1] (version 1.8.1) and Verilator 5.006. For instrumentation we used clang-15 with our own instrumentation pass (BFSan). We ran the DHRYSTONE benchmark provided by the chipyard infrastructure on both the "stock" MediumBoomCore and the same version compiled with our toolchain. Our experiments show a mean runtime of 26s ± 0.04s for the uninstrumented core, and a mean runtime of 34.9s ± 0.09s for the instrumented version, resulting in a ≈34% slowdown.

Scalability. To measure the impact of our instrumentation on bigger cores, we also ran the DHRYSTONE benchmark on the LARGEBOOM configuration (3-wide, 96 RoB entries, 100 physical registers). Our experiments show a runtime of 34.6s ± 0.01s on the uninstrumented simulation and 110.2s ± 0.2s on the instrumented simulation over 10 runs. When applied to the programs generated by Phantom Trails' fuzzer, we observed a fuzzing throughput degradation of around 85%.

ISA Simulation Overhead. For the ISA simulator, we build a software wrapper around Spike and include it as a library in the simulation wrapper. We ran our modified version of the Spike simulator on the DHRYSTONE benchmark. The observed runtime is 63.1 ms ± 7.5 ms, which constitutes a ≈0.2% overhead on top of the instrumented simulation.

Input Discarding. During architectural simulation, Phantom Trails' detector can optionally be configured to discard programs that hinder correct classification, i.e., programs that jump to the middle of an instruction, modify the code region (SMC), jump outside of the code region, or jump to the next instruction. This is particularly useful during fuzzing, since early discarding prevents from wasting precious cycle-accurate simulation cycles. When running Phantom Trails' program generator without any feedback we observed that ≈60% of the randomly-generated programs are discarded.

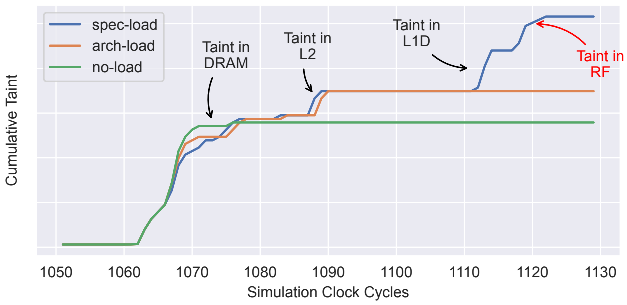


Figure 7: Cumulative amount of tainted locations for different test cases: a program with 0 loads (**no-load**), a program with 1 architectural load (**arch-load**), and the same program with an additional speculative load (**spec-load**).

PoC Detection. We created a testsuite of minimal PoCs, available at `https://github.com/vusec/riscv-transient-attacks`, for all known speculative vulnerabilities on the BOOM core (Spectre-v1, Spectre-v2, Spectre-RSB, Spectre-SSB and Meltdown) consisting of at most 60 lines of RISC-V Assembly. Our experiments show that Phantom Trails is able to detect the secret leakage in all PoCs before the secret is encoded into a covert channel. Each program accesses only a limited amount of memory, while the vast majority of locations is never accessed architecturally. To confirm the effectiveness of taint propagation, we produced a taint profile similar to the one in Figure 7 for each PoC and analyzed which components get tainted during simulation.

False Negatives. While in its basic configuration Phantom Trails' detector can already catch all speculative vulnerabilities known on BOOM, it can be extended to catch other x86-specific variants, such as MDS, where the leaked value is generated *architecturally*, by making sure that internal buffers such as the Store Buffer are also properly initialized and tainted (see Section 5.4).

False Positives. To verify the leaks reported by Phantom Trails, we employ the flush-based classification strategy described in Section 5.3, which is able to confirm that non-architectural data is loaded from a speculative instruction. Phantom Trails also allows further manual inspection by producing two reports: a json file representing the microarchitectural state at the moment of detection and a log of which new components are tainted after each clock cycle. We used such inspection tools to check for cases of overtainting, which could be a source of false positives. We note that, after adding taint washing of the I-Cache, we did not encounter any overtainting samples. We attribute this to our bit-precise taint tracker *BFSan*, as well as the fact that we only track direct data flows.

7.2 Fuzzing Evaluation

We evaluate the Phantom Trails fuzzer along several dimensions to answer the following questions:

Q1 Can we find all known transient execution vulnerabilities in reasonable time, despite the lack of templates?

Q2 What is the impact of replacing the default SW feedback with more advanced taint-based feedback?

Q3 What are the individual contributions of the various optimizations?

Q4 How does Phantom Trails perform compared to templated approaches?

In particular, we develop two copies of the same setup with the two different types of feedback (*SW* and *taint*, Section 6.3), to allow the effects of the feedback mechanism to be measured in isolation. We run 10 fuzzing campaigns for each feedback on AMD Ryzen Threadripper PRO 5995WX machine with 128 cores and 500 GiB of RAM and report the time-to-exposure (TTE) for various vulnerabilities on BOOM in Table 1.

Each campaign is given a single program consisting of one `nop` instruction as a starting seed. Our fuzzing campaigns have variable duration and only stop when the fuzzer has found each of the selected vulnerabilities. This way, we are not only able to answer the first question listed above, but also account for outliers in the data set with respect to TTE.

Additionally, we run a sequence of 24-hour campaigns for different configurations of the fuzzer, to evaluate the contributions of the different optimizations described in Section 6.2.

Total Runtime. Figure 8 shows a comparison of the *total runtime* of each campaign for the two different types of feedback, i.e., the time needed to find all of the reported vulnerabilities. In both cases, the total runtime is on average well below 24 hours, with a geomean of ≈9h for the software feedback and ≈14.5h for the taint feedback, as opposed to weeks [33]. Our first conclusion, in answer to question **Q1**, is that the problem of finding speculative vulnerabilities with fuzzing is tractable, even without smart seeds or templates. In addition, with respect to **Q2**, we see that Phantom Trails works well with both types of feedback, but on average the simple SW feedback performs slightly better than the more advanced taint-based feedback.

Time/Iterations-To-Exposure. Table 1 reports the time-to-exposure measurements for each vulnerability. The table also provides the number of programs that were executed by the fuzzer until each vulnerability was found (*iterations-to-exposure*). We provide both units of measurement as they reflect different properties of the feedback mechanisms. Time-to-exposure accounts for the usefulness of the feedback mechanism as well as the overhead required to measure the relevant metrics. Iterations-to-exposure ignores the overhead and therefore provides insight into how useful the feedback is to the fuzzing process. Figure 9 shows a visual comparison of the TTEs of each vulnerability for the two different feedbacks. Regarding question **Q2**, our experiments show that the two feedback metrics are mostly comparable. The only statistically significant difference was observed for Spectre v1, for which the SW feedback yielded a smaller TTE.

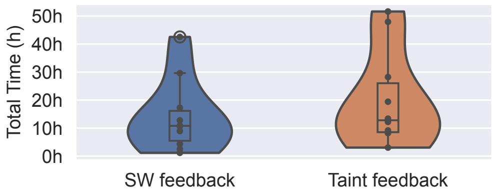


Figure 8: Total TTE for different types of feedback. Each point represents the runtime of a separate fuzzing campaign that found all the relevant vulnerabilities.

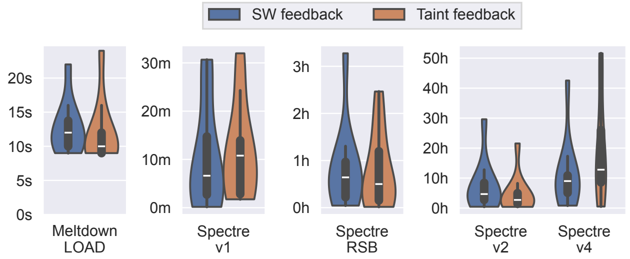


Figure 9: Time-To-Exposure for different vulnerabilities (smaller is better).

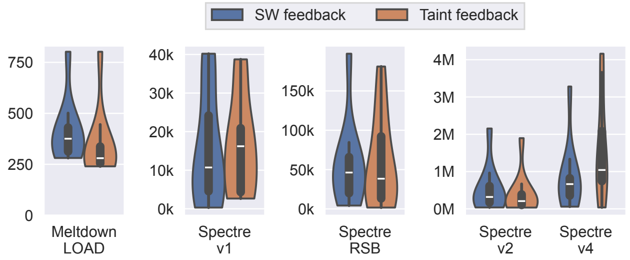


Figure 10: Iterations-To-Exposure for different vulnerabilities (smaller is better).

Optimizations Breakdown. In answer to question **Q3**, concerning the contribution of the optimizations described in Section 6.2, we run a 24-h campaign 5 times for each set of optimizations, and measure how many runs found each of the bugs, which estimates the probability of finding each bug in a 24-h run of the fuzzer. Table 2 reports the measured probability of exposure of different bugs for the different sets of optimizations. The results show that our simple data-flow and control-flow optimizations play a key role in finding Spectre-v4 and Spectre-RSB, respectively, and both are needed to find

Vulnerability	SW Feedback					Taint Feedback				
	Iterations		Time			Iterations		Time		
	Mean	Stdev	Mean	Stdev	Geomean	Mean	Stdev	Mean	Stdev	Geomean
Meltdown	410	±154	12s	±3s	**12s**	348	±170	12s	±4s	**11s**
Spectre v1	15k	±14k	10m 28s	±11m 7s	**4m 31s**	16k	±13k	11m 27s	±10m 4s	**7m 33s**
Spectre RSB	56k	±55k	50m 34s	±56m 49s	**27m 40s**	59k	±60k	46m 54s	±49m 27s	**21m 28s**
Spectre v2	528k	±637k	7h 22m	±8h 40m	**4h 9m**	394k	±562k	4h 45m	±6h 22m	**2h 28m**
Spectre v4	853k	±933k	11h 18m	±12h 1m	**6h 47m**	1.59M	±1.36M	19h 52m	±17h 25m	**12h 7m**
All	1.06M	±991k	14h 4m	±12h 56m	**9h 3m**	1.62M	±1.33M	20h 9m	±17h 6m	**14h 32m**

Table 1: Statistics for the Time-To-Exposure and Iterations-to-Exposure of different vulnerabilities across 10 runs.

Vulnerability	All	Data-Flow	Control-Flow	Basic	None
Meltdown	10/10	5/5	5/5	5/5	0/1*
Spectre v1	10/10	5/5	5/5	5/5	0/1*
Spectre RSB	10/10	2/5	5/5	0/5	0/1*
Spectre v2	9/10	1/5	1/5	0/5	0/1*
Spectre v4	9/10	5/5	1/5	2/5	0/1*

*7-day run on Intel Xeon Silver

Table 2: Probability of exposing each bug in a 24h run for different fuzzer configurations, expressed as number of runs in which the bug was found / total number of runs.

Spectre-v2 samples in reasonable time. In contrast, we ran a version of the fuzzer without *any* optimization on an Intel Xeon Silver 4310 machine with 48 cores and 126 GiB of RAM for 7 days. This version was only able to uncover Out-Of-Order loads in the allotted amount of time, showing the importance of simple program generation optimizations in making the problem tractable.

Comparison with Templating. To answer question Q4, we provide a TTE comparison between Phantom Trails and SpecDoctor [33], a state-of-the-art pre-silicon fuzzer for transient execution vulnerabilities. SpecDoctor employs a variety of templating techniques. For example, programs generated by the fuzzer are divided into a *prefix* code section and a *transient* code section. The prefix contains 3 to 5 basic blocks, and each block has a 50% chance of containing a predefined snippet that loads 64 secret bits into the L1 cache. Moreover, when mutating the prefix code, there is a 20% chance of introducing a snippet that is meant to slowdown a specific operation. In the transient section, each load has a 1 in 8 chances of targeting the predefined secret memory section.

We ported Phantom Trails to the same BOOM configuration as used in the SpecDoctor paper (1-wide, 32 RoB entries, 52 physical registers) and ran a fuzzing campaign on a comparable setup (Intel Xeon Silver 4310, 40 CPUs, 128GB RAM). Table 3 shows a comparison between the TTEs reported in the SpecDoctor paper (lefmost column) and the ones measured for Phantom Trails (rightmost column) for the same set of vulnerabilities. We also ran SpecDoctor in two additional configurations: `no-sidechan` and `no-templ`. In the former, we disabled side-channel generation and reported the intermediate seeds. While this only reports *potential* Spectre-v1 and Meltdown candidates, it provides insight on the overhead of program generation and the impact of templates. In the latter (`no-templ`), we also removed part of the templating primitives employed by the fuzzer, i.e., the secret-preloading snippet and the slowdown snippet. We observe that Phantom Trails greatly outperforms SpecDoctor in its default configuration and, when compared to the `no-sidechan` configuration, is still able to perform similarly to templated approaches for Spectre v1 while maintaining a generic instruction generator, even outperforming SpecDoctor on Meltdown. Moreover, by just removing the L1-prefetching gadget and the delay gadget from the generator, we observe that SpecDoctor's TTEs significantly increases for Spectre v1, and saturates the timeout of 8h for our experiments for Meltdown. We also notice that seeds have much more impact in the `no-templ` configuration, as several runs (which we not included in the TTE computation) timed out after 24h without finding any vulnerability.

Vuln.	SpecDoctor*	no-sidechan†	no-templ†	PT†
Meltdown	34.7h	2h ± 3.2h	> 8h ‡	**9s** ± 14s
Spectre v1	26.9h	3m ± 5m	11.3m ± 6m	**5.3m** ± 5m

*Total CPU time reported in [33] †TTE geomean across 10 runs
‡All runs timed out

Table 3: TTE comparison between Phantom Trails and SpecDoctor.

8 Spectre-LoopPredictor

During our fuzzing campaigns, Phantom Trails was able to uncover a new Spectre-v1 variant on BOOM caused by the interaction between the PHT and BOOM's LoopPredictor.

```
1 li t0, 1
2 loop:
3 auipc ra, 0  # <-- ret lands here
4     # ... slow down t0 ...
5     beqz t0, speculative  # always false
6 ret  #  jump to 3 (always mispredicts)
7
8 speculative: # <-- speculation ends up here!
9     ld t0, 0(spec_data)
```

Listing 2: Simplified version of the Spectre-LP sample found by Phantom Trails.

The LoopPredictor. The LoopPredictor is responsible of identifying loops and predicting how many iterations of a loop will be executed before exiting. Each predictor entry holds a *confidence* value, indicating how "sure" the predictor is about that entry. When the confidence is low, the LoopPredictor simply forwards the predictions of the PHT. Once the confidence value reaches a threshold, the LoopPredictor takes over by *flipping* the prediction output.

Spectre-LP. Listing 2 shows the simplest example of Spectre-LP. On line 3, the current PC is saved into `ra` (return address register). Then, on line 5, a branch is executed, which is always architecturally *not-taken* (the condition is always false). Finally, the `ret` on line 6 reads `ra` and jumps back to line 3, creating a loop. The direct assignment of `ra` on line 3 bypasses the Return Address Stack (RAS), which is used by the CPU to speculate on returns. As a consequence, the CPU mis-speculates the `ret` at each loop iteration. An unexpected side-effect of such repeated misprediction is that, after 7 iterations of the loop, the branch on line 5, which is *never taken*, unexpectedly starts to be predicted "taken", causing the CPU to speculatively jump to the gadget on line 9.

Root Cause Analysis. Our analysis shows that the root cause of the reported leak can be traced back to an aliasing of the two instructions (the mispredicted branch and the `ret`) in the *LoopPredictor* of BOOM. In particular, the repeated `ret` mispredictions cause the LoopPredictor to increase its confidence until the threshold (7) is reached, indicating that it is sure it has found a loop. On the next iteration, the correct prediction for the branch (not taken) coming from the BIM is flipped by the LoopPredictor, generating the behavior we observed.

Practical Exploitation. We were able to reproduce this behavior on an unmodified BOOM simulation in its MediumBoom configuration by substituting the misprediction at line 9 with different types of misspeculations including *branch mispredictions*, which confirms that this behavior is not specific to `rets`. Indeed, causing repeated mispredictions on *any* control-flow instruction at the end of a loop causes nearby branches to be mispredicted. Contrary to traditional Spectre bugs, where the attacker is in control of the mispredicted branch, here the attacker controls a *nearby* branch whose misspeculation affects the target. This means that any scanner that checks for attacker-controlled branches would not cover this case, and an attacker would be able to attack branches even in the presence of software mitigations for typical Spectre v1 gadgets.

9 Related Work

Scope. Our work focuses on finding transient execution vulnerabilities in CPU designs. Rather than attempting to detect the ever-growing set of possible covert channels, which include array-based variations [27, 70], direct [8, 24, 55] or indirect [55] branches, AVX instructions [56], Rowhammer [18, 63], TLB [13, 43], noncanonical translation via Intel LAM [31], resource contention [7, 8] and many others, we take a different approach by detecting transient leaks, which happen *before* any of the aforementioned covert-channel transmissions. We also specifically target *transient execution* vulnerabilities, which do not include other microarchitectural vulnerabilities *not* based on transient execution such as PortSmash [4] and GoFetch [14].

Hardware verification. Formal verification methods such as UPEC [21], Iodine [26], Conjunct [19], and LeaVe [67] have been applied to side-channel leakages in hardware. These methods prove security properties via inductive invariants. While such methods can provide strong formal guarantees for the absence of vulnerabilities, they are difficult to scale to larger designs like BOOM, or require prohibitive manual effort [21]. On the other hand, fuzzing techniques such as the one adopted by Phantom Trails are not able to provide completeness guarantees, but they can scale to realistic designs.

Black-box fuzzing. Frameworks such as Transynther [47], Osiris [68], SMoTherSpectre [8] and Revizor [49, 50] aim to automatically generate side-channel attacks by either observing the timing or cache behavior of a program or by inspecting the CPU's performance counters in a black-box setting, *i.e.*, without access to the RTL. While this approach is useful for fuzzing closed-source commercial products, it is inherently approximate and cannot leverage deep inspection of the RTL for fuzzing feedback and root-cause analysis.

Hardware-Software contracts. Revizor [49] proposes an approach based on hardware-software contracts: given a program, Revizor finds *pairs* of inputs that lead to the same contract trace but have different hardware traces. By contrast, our implicit secret generation uses a *single* ISA run to infer an invariant that is applicable immediately to the hardware simulation. Generating proper differential inputs adds yet another dimension of entropy to the problem. To tackle this, the original Revizor paper generates programs with only four registers, confines the memory sandbox to one or two 4K memory pages, and lowers the entropy of the PRNG. Such additional entropy can be constrained using e.g., symbolic execution [48], which however does not scale to big inputs, or contract-based generation [50], which identifies which part of the input should *not* change, however it cannot be completely eliminated due to the differential nature of the approach.

Hardware fuzzers. Prior work explored the concept fuzzing hardware designs to find *architectural* bugs. PSOFuzz [15] gathers data during fuzzing to dynamically adjust the selection of weights for each mutation. TheHuzz [38] tries to estimate the optimal mutations for a given processor. HyPFuzz [16] proposes using static analysis to generate inputs that reach certain parts of the DUT. MorFuzz [72] uses run-time morphing of instructions to guide the execution and achieve higher coverage. Cascade [59] uses intricate program generation to create complex programs with varied control- and data-flow dependencies. Each of these optimization strategies comes with different overheads and is complementary to our work, as they mainly concentrate on finding implementation bugs.

Other prior work proposes various hardware coverage metrics that could be used as feedback mechanisms when fuzzing hardware. Trippel et al. [64] suggested using edge coverage of the hardware simulation, which is the basis for the 'SW feedback' metric we use in Section 7.2. DiFuzzRTL [35] uses the value-transitions of finite-state-machine registers as an alternative coverage metric. ProcessorFuzz [12] proposes using the value-transitions of manually selected CSR registers as fuzzing feedback. TaintFuzzer [32] uses a user-provided cost function and a database of vulnerabilities to find analyst-defined security violations, and uses taint inference to assess the impact of inputs on the hardware system. All such metrics can be added on top of our system, and are orthogonal to our work. Finally, Whisperfuzz [9] and SIGFuzz [54] are hardware fuzzers that search for timing side-channels in RTL designs, which is orthogonal to finding transient data leaks, as explained in Section 4.1.

Finding transient execution vulnerabilities. Previous work has proposed the idea of using taint tracking in relation to transient execution vulnerabilities. **Speculative Taint Tracking** [74] (STT) proposes a hardware defense mechanism for speculative execution attacks that taints speculative values inside the pipeline. In particular, all data coming from speculative loads is tainted until the instruction passes a point of no-return, after which it is guaranteed to retire. This model is ill-suited for fuzzing, since every transient load (regardless of secret/nonsecret access) would trigger detection. Moreover, STT is designed to be an invasive solution, which requires changes to the Front-End, Execution Units, Branch Unit, and Load-Store Unit. It also requires the architect to identify all instructions that can potentially leak a secret value. Our tool instead uses taint tracking on the whole CPU (DRAM to Register File), employs a taint policy which is suitable for greybox fuzzing, and requires minimal knowledge of the DUT (and no hardware modifications).

CellIFT [60] proposes a new taint tracking mechanism in hardware that can (among other things) be applied to detecting transient execution attacks. As there is no fuzzing component, the user manually specifies where secret data is located in the design, and manually chooses a program to execute in order to demonstrate a transient execution attack. Similar to Phantom Trails, the user chooses a set of taint sinks and the technique indicates a leak, when such a sink becomes tainted. However, Phantom Trails can automatically generate the leaking program and automatically infer the secret memory regions from the ISA simulator. Moreover, CellIFT instruments the hardware design at the RTL level, while Phantom Trails uses software taint tracking.

IntroSpectre [25] and SpecDoctor [33] are RTL fuzzers that specifically target transient execution vulnerabilities. **IntroSpectre** uses a set of predefined code gadgets to generate inputs, which include snippets such as "Create contention on execution units with the same write port" to mimic the PoCs of known vulnerabilities. We take a radically different (and vulnerability-agnostic) approach by mutating random (valid) programs starting from an empty seed and letting the coverage feedback guide the fuzzer. Moreover, IntroSpectre is limited to Meltdown-type vulnerabilities. Finally, IntroSpectre populates memory with specific values which are then searched in the execution log, while we use taint tracking (so even values that are computed from a secret are tracked). **SpecDoctor** uses multi-phased fuzzing based on templates to find snippets that display end-to-end attacks. The fuzzer first finds speculation windows, then tries to find covert channels that can leak a secret with differential testing. Secrets are explicitly defined by marking specific pages as protected. Both methods rely on predefined program parts that are composed and modified to generate the attack, which limits generality. In contrast, Phantom Trails finds transient data leaks starting from an empty seed without the use of templates or smart seeds. Moreover, Phantom Trails automatically infers secrets by computing the set of non-architectural addresses using an ISA simulator.

10 Conclusions

We presented a new approach to pre-silicon transient execution vulnerability discovery that concentrates on microarchitectural data leaks and uses an implicit secret model to create a fuzzer-friendly detector. We apply taint tracking to the software simulation of a CPU and monitor the flow of secrets towards exposed buffers, using pipeline flush signals for classification. Without templating or smart seeds, our fuzzer finds samples of speculative vulnerabilities on the BOOM processor from an empty seed in hours, instead of weeks, and finds Spectre-LP, a new speculation primitive on the BOOM processor that can be used to cause mispredictions on uncontrolled branches. In future work, we plan to explore the possibility of using multiple taint colors for finer-grained secret modeling and work towards finding optimal feedback mechanisms across different classes of microarchitectural attacks.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback, and Nassim Corteggiani for his early feedback on the project. This work was supported by Intel Corporation through the "Allocamelus" project, by NWO (through project "INTERSECT", the Gravitation "CiCS" project grant 024.006.037, and the Dutch Prize for ICT research), by the EU's Horizon Europe programme (under grant No. 101120962, "Rescale"), and the European Research Council (through ERC Starting Grant no. 101115046 "SecuStack", and ERC Advanced Grant no. 101141972 "Ghostbuster").

Ethics Considerations and Compliance with the Open Science Policy

Disclosure. Upon discovery of the first sample of Spectre-LP in our experiments, we immediately disclosed the PoC to the maintainers of the BOOM project, asking if we could include it in a publication. In their response, they acknowledged the presence of the edge case in the LoopPredictor, and stated that they were "not aware of BOOM being used in any contexts where information security is at stake". Thus, no embargo on publication was imposed. Additionally, we publicly disclosed the vulnerability on the official BOOM repository.

We believe that by targeting the BOOM core, we were able to prove the capabilities of our fuzzer on a complex CPU without endangering users, as the public version of this core is used mainly for research purposes.

Open Science Policy. All the source code of Phantom Trails, including our LLVM instrumentation for Verilator, taint-tracking wrapper for BOOM, and modified Spike simulator is available at https://zenodo.org/records/14768707, along with all transient leak experiments on BOOM and Spectre-LP reproducers. All experiments reported in the paper can be reproduced following the instructions contained in the open-source repository, which is hosted at https://github.com/vusec/phantom-trails.

References

[1] Chipyard. https://chipyard.readthedocs.io/en/stable/.

[2] Spike risc-v isa simulator. https://github.com/riscv-software-src/riscv-isa-sim, 2023.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology – CT-RSA 2007*, 2006.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE S&P*, 2019.

[5] Krste Asanović, David A. Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. 2015.

[6] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*, 2022.

[7] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. Speculative interference attacks: Breaking invisible speculation schemes. In *ASPLOS*, 2021.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[9] 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. *USENIX Security*, 2024.

[10] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security*, 2022.

[11] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *DAC*, 2021.

[12] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Processorfuzz: Processor fuzzing with control and status registers guidance. In *HOST*, 2023.

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, 2019.

[14] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *USENIX Security*, 2024.

[15] Chen Chen, Vasudev Gohil, Rahul Kande, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Psofuzz: Fuzzing processors with particle swarm optimization. In *ICCAD*, 2023.

[16] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. {HyPFuzz}:{Formal-Assisted} processor fuzzing. In *USENIX Security*, 2023.

[17] ChipsAlliance. Chisel. https://www.chisel-lang.org/.

[18] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D Keromytis, Yossi Oren, and Yuval Yarom. Hammerscope: observing DRAM power consumption using Rowhammer. In *CCS*, 2022.

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

[20] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 2018.

[21] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. An exhaustive approach to detecting transient execution side channels in rtl designs of processors. *IEEE Transactions on Computers*, 2022.

[22] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *WOOT*, 2020.

[23] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *CCS*, 2022.

[24] Jacob Fustos, Michael Bechtel, and Heechul Yun. SpectreRewind: Leaking secrets to past instructions. In *ASHES*, 2020.

[25] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *ISCA*, 2021.

[26] Klaus v Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. {IODINE}: Verifying {Constant-Time} execution of hardware. In *USENIX Security*, 2019.

[27] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *CCS*, 2020.

[28] Google. Retpoline: a software construct for preventing branch-targetinjection. https://support.google.com/faqs/answer/7625886.

[29] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. TLBleed: When Protecting Your CPU Caches is not Enough. In *Black Hat USA*, 2018.

[30] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *CCS*, 2016.

[31] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation. In *IEEE S&P*, 2024.

[32] Muhammad Monir Hossain, Nusrat Farzana Dipu, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. Taintfuzzer: Soc security verification using taint inference-enabled fuzzing. In *ICCAD*, 2023.

[33] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *CCS*, 2022.

[34] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE S&P*, 2021.

[35] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE S&P*, 2021.

[36] Intel. Bhi disclosure documentation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[37] Intel. Intel analysis of speculative execution side channels. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/

analysis-speculative-execution-side-channels.html.

[38] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *USENIX Security*, 2022.

[39] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE S&P*, 2019.

[40] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 2018.

[41] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[43] Kevin Loughlin, Ian Neal, and Jiacheng Ma. DOLMA: Securing speculation with the principle of transient non-observability. In *USENIX Security*, 2021.

[44] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, 2018.

[45] Alyssa Milburn, Ke Sun, and Henrique Kawakami. You cannot always win the race: Analyzing mitigations for branch target prediction attacks. In *EuroS&P*, 2023.

[46] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *USENIX Security*, 2023.

[47] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX Security*, 2020.

[48] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In *CAV*, 2020.

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

[50] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *IEEE S&P*, 2023.

[51] Tavis Ormandy. Zenbleed. https://lock.cmpxchg8b.com/zenbleed.html.

[52] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

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

[54] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. Sigfuzz: A framework for discovering microarchitectural timing side channels. In *DATE*, 2023.

[55] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead micro-ops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, 2021.

[56] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[57] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*, 2012.

[58] Wilson Snyder. Verilator. https://www.veripool.org/verilator/.

[59] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: Cpu fuzzing via intricate program generation. In *USENIX Security*, 2024.

[60] Flavien Solt, Ben Gras, and Kaveh Razavi. CellIFT: Leveraging cells for scalable and precise dynamic information flow tracking in RTL. In *USENIX Security*, 2022.

[61] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*, 2015.

[62] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. {TLB; DR}: Enhancing {TLB-based} attacks with {TLB} desynchronized reverse engineering. In *USENIX Security*, 2022.

[63] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. SpecHammer: Combining Spectre and Rowhammer for new speculative attacks. In *IEEE S&P*, 2022.

[64] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *USENIX Security*, 2022.

[65] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE S&P*, 2020.

[66] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue inflight data load. In *IEEE S&P*, 2019.

[67] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and verification of side-channel security for open-source processors via leakage contracts. In *CCS*, 2023.

[68] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *USENIX Security*, 2021.

[69] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. InSpectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2. In *USENIX Security*, 2024.

[70] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *USENIX Security*, 2022.

[71] Johannes Wikner and Kaveh Razavi. Breaking the Barrier: Post-Barrier Spectre Attacks. In *IEEE S&P*, 2025.

[72] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. {MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *USENIC Security*, 2023.

[73] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *USENIX Security*, 2014.

[74] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *MICRO*, 2019.

[75] Michal Zalewski. American fuzzy lop. https://github.com/google/AFL.

USENIX Security ’25 Artifact Appendix: Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks

Alvise de Faveri Tron
Vrije Universiteit Amsterdam

Raphael Isemann
Vrije Universiteit Amsterdam

Hany Ragab
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Klaus von Gleissenthall
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

A Artifact Appendix

A.1 Abstract

We present a prototype of our pre-silicon fuzzer that uses a novel detection mechanism based on implicit secrets to find transient data leaks in CPU designs. Our artifact includes all the components of our fuzzing infrastructure, namely (1) an LLVM pass that adds bit-precise taint-tracking to the CPU simulation, (2) an AFL++ fork that adds the option to use taint as feedback, and (3) a LibAFL fuzzing driver, along with our RISC-V instruction generator and mutator, to drive the fuzzing loop. The artifact also includes the detector infrastructure we built for BOOM, namely (1) an architectural simulator that infers all architecturally-accessed locations, (2) the BOOM configuration used for fuzzing and (3) the C++ wrapper that drives the Verilated simulation and detects crashes. The artifact also includes a testsuite of minimal detectable PoCs for both known vulnerabilities (Spectre-v1, Spectre-v2, Meltdown, Spectre-v4, Spectre-RSB) and the new Spectre-LP vulnerability. Information about each component can be found in the README and in the paper.

A.2 Description & Requirements

The scripts provided in the artifact can be used to setup Phantom Trails on the BOOM core and evaluate its results.

A.2.1 Security, privacy, and ethical concerns

Experiments are completely local, and no destructive steps are taken during evaluation.

A.2.2 How to access

The main entrypoint of our artifact is available at https://github.com/vusec/phantom-trails at the tag `ae-initial`. The final artifact version will be available on Zenodo at https://zenodo.org/records/14726711.

A.2.3 Hardware dependencies

No specific hardware feature is required, however to reproduce the TTE measurements of our fuzzing campaigns it is required to run on comparable machines. Our experiments have been conducted on two machines:

1. (**M1**) AMD Ryzen Threadripper PRO 5995WX machine with 128 cores and 500 GiB of RAM
2. (**M2**) Intel Xeon Silver 4310 machine with 48 cores and 126 GiB of RAM

A.2.4 Software dependencies

No specific OS is required in principle. Our experiments have been tested on Ubuntu Linux 22.04.

A.2.5 Benchmarks

None.

A.3 Set-up

The only prerequisites are `git` and `docker`. Scripts require `bash` and `python3`.

A.3.1 Installation

Clone the repository and all dependencies:

```
git clone https://github.com/vusec/phantom-trails
cd phantom-trails
git submodule update --init --recursive
```

Build the instrumented simulation in a container (takes ≈40 minutes on a 48 cores/126GB RAM machine):

```
cd BOOM
./start.sh
```

By default, this script will build Phantom Trails with basic block coverage as feedback and the BOOM core simulation in its MediumBoom configuration inside of a container named `boom-fuzz-manual-started`. Once the script has terminated, you should get a shell inside of the docker container.

If the build script exits before completion, it is very likely that the process ran out memory during the building and linking phases of LLVM. In that case, you can modify (decrease) the number of parallel jobs by substituting `nproc` at line 21 of `BOOM/start.sh`.

A.3.2 Basic Test

After `start.sh` the shell is located in the `/chipyard/sims/verilator` folder of the Docker container. From there you can run some simple tests.

To verify that the detector works as intended, you can run:

```
phantom-trails run \
    /Samples/build/bins/pocs/meltdown-us.bin
```

This loads the selected RISC-V flat binary in the instrumented simulation's memory and runs it. The source code of the test can be found in `Samples/src/pocs/meltdown-us.S`.

The expected results is that the command will fail, and will indicate "Meltdown PF" as cause before exiting.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Phantom Trails is able to detect all known speculative vulnerabilities on the BOOM core (Spectre-v1, Spectre-v2, Spectre-RSB, Spectre-SSB and Meltdown) at the secret-access step. Experiment **E1** confirms that each PoC of our testsuite is correctly detected and classified, as reported in Section 7.1, paragraph "Poc Detection".

(C2): Phantom Trails was able to uncover a new Spectre-v1 variant, Spectre-LP, described in Section 8 of the paper. Experiment **E2** confirms that different variants of Spectre-LP, including a return-based variant and a branch-based variant, are detected by Phantom Trails.

(C3): Phantom Trails can be extended to handle MDS. Experiment **E3** reproduces the detection behavior as reported in Section 5.4 (MDS Detection).

(C4): Phantom Trails is able to find all known transient vulnerabilities on the BOOM core in reasonable time despite the lack of templates. Experiment **E4** reproduces one of our fuzzing campaigns as described in Section 7.2 (Question Q1), whose results are reported in Figure 8-10 (blue) and in Table 1 (left).

(C5): Phantom Trails is still able to perform similarly to templated approaches on SmallBoomCore. Experiment **E5** reproduces the TTE measurements reported Table 3 (rightmost column).

(C6): Phantom Trails uses simple control-flow and data-flow optimizations. Experiment **E6** described in Section 7.2 (Question Q3) shows the impact of each optimization. Our results are reported in Table 2.

(C7): Phantom Trails provides the possibility of using Taint as a fuzzing feedback. Experiment **E7** described in Section 7.2 (Question Q2) can be used to measure the impact of the taint feedback. Our results are reported in Figure 8-10 (orange) and in Table 1 (right).

A.4.2 Experiments

(E1): PoC Detection [1 compute-minutes]: Run Phantom Trails on PoCs of known vulnerabilities.
Preparation: After running `BOOM/start.sh` you should be running in a `fish` shell inside of the Docker container, in the `/chipyard/sims/verilator` folder.
Execution: Run the following (`fish` syntax)

```
for poc in /Samples/build/bins/pocs/*;
    echo "===== Running $poc ======"
    phantom-trails run $poc;
end;
```

Results: For each sample, Phantom Trails should abort and print "Found issue: " alongside its classification. Samples should include 1 Meltdown sample (reported as `Meltdown_LOAD_PFAULT`), 2 Spectre-RSB samples, 2 Spectre-v1 samples, 1 Spectre-v2 sample, and 2 Spectre-v4 samples.

(E2): Spectre-LP [1 compute-minutes]: Run Phantom Trails on Spectre-LP snippets.
Preparation: You should be running in a `fish` shell inside of the Docker container, in the `/chipyard/sims/verilator` folder.
Execution: Run the following (`fish` syntax)

```
for poc in /Samples/build/bins/spectre-lp/*;
    echo "===== Running $poc ======"
    phantom-trails run --init=/dev/null $poc;
end;
```

Results: The folder includes 3 Spectre-LP PoCs. For each of them, Phantom Trails should abort and print "Found issue: Spectre_v1_new".

(E3): MDS [15 compute-minutes]: Build BOOM with MDS and run Phantom Trails to verify that MDS is detected.
Preparation: Inside of the Docker container, build BOOM with MDS-SB:

```
phantom-trails build --config MDSConfig -j<JOBS>
```

Execution: Run the following

```
phantom-trails run --mds --config=MDSConfig \
     /Samples/build/bins/mds-tests/mds.bin
```

Results: The command is expected to abort after printing "Found issue: MDS_STORE_BUFFER".

(E4): Fuzzing Campaign with SW Feedback [24 to 48 compute-hours]: Run a fuzzing campaign on MediumBoomCore with the Software Feedback.

Preparation: You should be inside of the Docker container. TTEs reported in Table 1 of the paper (left part) have been observed on machine **M1** described in Section A.2.3.

Execution: Run the fuzzer

```
phantom-trails fuzz
```

By default, this will run the fuzzer until all samples in `expected_findings` are found. You can kill the fuzzer at any time with:

```
sudo killall sim-fuzzer && sudo killall run-FuzzConfig
```

Results: While the fuzzer is running, you should see TTEs being updated live in the TUI (top-right panel). Once the fuzzer has found all expected crashes, the results of the campaign are available in `/chipyard/sims/verilator/out`. The same folder is also available outside of the container, under `BOOM/results/manually-started/<timestamp>/`. Crashing inputs after classification can be found in the `out/causes` folder, in flat-binary format. You can retrieve time-to-exposure and iterations-to-exposure from within the container with:

```
# Print TTEs.
/external/BOOM/eval-results-folder.py -z out/
# Printe ITEs.
/external/BOOM/eval-results-folder.py -t out/
```

(E5): Fuzzing Campaign on SmallBoom [< 1 compute-hour]: Run a fuzzing campaign on SmallBoomCore for Meltdown and Spectre-v1 (used to compare against SpecDoctor).

Preparation: Inside of the Docker container, build BOOM in the SmallBoom configuration:

```
phantom-trails build --config SmallFuzzConfig -j<JOBS>
```

Modify `expected_findings` to match SpecDoctor.

```
# Copy old list.
cp expected_findings expected_findings_all
# Create new list.
cat expected_findings_all | head -n 2 > expected_findings
# Only Spectre v1 and Meltdown should be remaining.
```

Finally, remove old results and fuzzing queue.

```
rm -rf out/*
rm .cur_input*
```

Execution: Run the fuzzer

```
phantom-trails fuzz --config SmallFuzzConfig
```

To kill the fuzzer, same instructions as Experiment E4 apply.

Results: TTEs reported in Table 3 of the paper (right-most column) have been observed on machine **M2** described in Section A.2.3. Statistics of the campaign can be printed as described in Experiment E4.

(E6): Impact of Fuzzing Optimizations [40 minutes + 24 compute-hours]: Remove optimizations from the fuzzer and verify their effect on vulnerability discovery.

Preparation: To run a fuzzing campaign without optimizations you will need to apply the patches located in `phantom-trails/eval-patches`. In principle it is possible to apply one patch at a time and re-run the fuzzing campaign. The most evident effect should be visible by removing both control- and data-flow optimizations ("Basic" column in Table 2).

```
# Exit from the container.
(container) exit
cd <PHANTOM_TRAILS_TOP_FOLDER>
# Remove dataflow opts.
git apply eval-patches/no-dataflow-*
# Remove control flow opts.
git apply eval-patches/no-cfg-*
cd Fuzzer
git apply ../eval-patches/fuzzer/no-cfg-*
```

Finally, re-build the container. This can take up to 40 minutes.

```
cd BOOM
./start.sh
```

Execution: From within the newly built container, run the fuzzer

```
phantom-trails fuzz
```

To kill the fuzzer, same instructions as Experiment E4 apply.

Results: After a 24-hours run it is highly probable that only Spectre-v1 and Meltdown will be found by the tool, as reported in Table 2 of the paper ("Basic" column).

(E7): Fuzzing Campaign with Taint Feedback [40 minutes + 24h to 48 compute-hours]: Run a full fuzzing campaign on MediumBoomCore with the Taint Feedback and verify TTEs.

Preparation: To run a fuzzing campaign with the taint feedback you will need to re-build the instrumented simulation. Exit from the container and run (from the phantom-trails top folder):

```
# Discard all previous modifications.
git checkout .
# Start container with Taint feedback.
cd BOOM
./start.sh "Taint"
```

This can take up to 40 minutes.

Execution: Run the fuzzer

```
phantom-trails fuzz
```

To kill the fuzzer, same instructions as Experiment E4 apply.

Results: TTEs reported in Table 1 of the paper (right part) have been observed on machine **M1** described in Section A.2.3. Statistics of the campaign can be printed as described in Experiment E4.

A.5 Notes on Reusability

Phantom Trails can be used beyond the scope of this paper to introspect the microarchitectural behavior of RISC-V programs and evaluate different feedback metrics.

- Phantom Trails help

  ```
  phantom-trails -h
  ```

  This should show a summary of the three main commands provided by the `phantom-trails` script: `build`, which can be used to build a specific configuration of the BOOM simulation (default one is `FuzzConfig`), `run`, which can be used to run the detector on a single program, and `fuzz`, which can be used to start a fuzzing campaign.

  `phantom-trails` is a simple python script that wraps the invocation of `make`, `run-FuzzConfig` (verilated simulation entrypoint) and `sim-fuzzer` (LibAFL fuzzer) for *build*, *run* and *fuzz* respectively. The content of the script can be inspected at `BOOM/scripts/phantom-trails` (or in the `/scripts/` folder within the container).

- Compile your own sample: once inside the container, you can run the following

  ```
  # Add RISC-V toolchain to env.
  bash
  source /chipyard/env.sh
  fish

  # Add new ASM file to src.
  cd /Samples/src
  echo "nop" > test.S
  cd /Samples && make

  # Run it with Phantom trails.
  cd /sims/chipyard/verilator
  phantom-trails run /Samples/build/bins/test.bin
  ```

- Debugging and Introspection: all configurations can be built with an optional debug flag

  ```
  # Clean previous sims.
  make clean

  # Build in debug mode.
  phantom-trails build -j<PROCS> --debug
  ```

  This enables the possibility of adding additional printing flags at runtime which can be used to inspect the cycle-by-cyle state of the microarchitecture

  ```
  # Run in verbose mode.
  phantom-trails run --verbose <SAMPLE>
  # Add --logfile out.log to output to a file.

  # Print cycle-by-cycle taint and RoB info.
  phantom-trails run --report <SAMPLE>

  # Inspect specific buffers.
  PRINT_REGFILE=1 PRINT_LSQ=1 phantom-trails run <SAMPLE>
  # See boom-wrapper/src/main/resources/csrc/args.h
  # for other print flags.
  ```

- Initialization: you can change or disable the initialization snippet by providing the `-init` flag to `phantom-trails run` (`-init=/dev/null` to disable)

  ```
  phantom-trails --init=/dev/null <SAMPLE>
  ```

  This will avoid prepending the init snippet to the sample under test. To inspect the state of memory right before execution (e.g., if you want to check the exact addresses of the initialization snippet and the sample-under-test in DRAM) you can run:

  ```
  # This generates <SAMPLE>.bin.memdump
  MEMDUMP=1 phantom-trails <SAMPLE>.bin

  # Disassemble the flat binary.
  riscv64-unknown-elf-objdump                           \
              -b binary -m riscv:rv64 -M no-aliases -D \
              <SAMPLE>.bin.memdump
  ```

- Other metrics: Our Dockerfile by default builds LLVM with MSAN (more precisely, our version of MSAN – BFSAN) but in principle different sanitizers can be provided at this stage. Moreover, we build the fuzzing instrumentation using AFL++, and in principle more coverage maps can be added to the codebase we already provide.

- Other buffers: While we use the Physical Register File as a taint sink, it is possible to move the sink to other buffers as well. For this purpose, you can modify `checkTaintSinks()` in `boom-wrapper/src/main/resources/csrc/Simulator.h`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.