Crucible: Retrofitting Commodity CPUs with Vulnerabilities via Transparent Software Emulation

Tristan Hornetz, Lukas Gerlach and Michael Schwarz
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

***Abstract*—Transient-execution attacks such as Meltdown, Foreshadow, and MDS expose fundamental flaws in modern CPUs, yet reproducing and comparing them today is increasingly complex: vulnerable CPUs are scarce, lab setups cannot be shared easily, and results are hard to compare. These challenges make it difficult to evaluate detection tools, study exploits, or integrate attacks into teaching environments. As vulnerable CPUs become rarer, hands-on experimentation and consistent benchmarking gradually become infeasible, complicating both research and education in microarchitectural security.**

In this paper, we introduce Crucible, a software-only framework that transparently simulates Meltdown-type transient execution vulnerabilities on any x86 CPU. Crucible simulates transient execution after a fault by shadowing the instruction stream in a different process to emulate key microarchitectural effects, such as cache leakage, transient windows, and fence behavior. Crucible runs unmodified public proofs-of-concept and even complete exploits with leakage patterns that match real hardware. We reproduce 3 full end-to-end exploits for well-known vulnerabilities, such as key extraction from VeraCrypt with Meltdown, on unaffected hardware. Crucible supports testing of binary-only applications and integrates with state-of-the-art fuzzers, which detect simulated vulnerabilities with results comparable to real CPUs. We further simulate two artificial vulnerabilities to evaluate generalization in fuzzer behavior. Our work enables systematic, repeatable experiments, preserves legacy vulnerabilities for future use, allows comparison of vulnerability detection approaches, and provides an accessible platform for teaching and training in CPU security.

1. Introduction

CPU security vulnerabilities have evolved into one of the most pressing challenges in computer security research. While still an emerging topic, recent years have seen a surge in attacks that target subtle implementation bugs, including Meltdown [1], Foreshadow [2], MDS attacks [3], [4], [5], [6], [7], and several architectural CPU vulnerabilities [8], [9], [10], [11]. These attacks shift the focus from software to hardware, revealing that even modern CPUs can leak sensitive information in unexpected ways. While CPU vulnerabilities may resemble software vulnerabilities in principle [8], they pose unique challenges to defenders. In contrast to software, where applying patches and updates is straightforward, CPUs are costly and time-consuming to replace or fundamentally re-engineer, and firmware updates are limited in what they can mitigate [8], [12]. Hardware bugs are ”baked in” post-manufacturing, leaving only imperfect mitigations that sacrifice performance [13], [14] or functionality [3], [4], necessitating research into CPU vulnerability detection and prevention.

Researching CPU vulnerabilities is challenging because achieving reproducibility and comparability is inherently difficult. Existing vulnerabilities are often only found on specific CPU generations, sometimes even requiring CPU models with specific stepping [5], microcode [11], or instruction-set extensions, such as Intel TSX [4]. Without a consistent research environment, researchers cannot systematically evaluate attacks across hardware platforms or benchmark vulnerability-discovery methods like fuzzers. Moreover, discontinued vulnerable CPUs become increasingly scarce, often requiring second-hand acquisition, which hampers their use in educational labs and complicates training future researchers.

In this paper, we propose a novel, software-only method to retrofit x86 CPUs with simulated hardware vulnerabilities. Our technique creates “artificial”, yet faithful, replicas of CPU vulnerabilities, allowing existing proof-of-concept implementations to run unmodified and yield behavior closely mirroring that of the original attacks. A central building block of our approach are *shadow streams*, which emulate the core mechanisms driving many high-impact CPU vulnerabilities. Whereas transient execution in hardware runs an instruction stream with effects invisible at the architectural level, we duplicate the instruction stream *architecturally*, executing the simulated transient instructions in a different process. Consequently, microarchitectural effects, such as changes to the cache, are triggered without altering the architectural state of the original program. Additionally, we ensure a simulation of the transient window, the effect of fences, and the impact on performance counters that is functionally equivalent to what attackers observe on affected CPUs. Our technique operates in user space, relying only on well-supported and stable interfaces of existing operating systems. However, it can be augmented with kernel data if desired, e.g., to simulate the leakage of real kernel memory with attacks like Meltdown.

We implement our approach as a configurable framework, Crucible, that acts as a transparent wrapper for arbitrary x86_64 ELF applications, without the need to modify them. With a single wrapper script, Crucible is also easy to apply to a wide range of existing applications, even if they are only available as binaries. Thus, in contrast to related approaches on RISC-V that require re-implementing vulnerabilities and PoCs [15], Crucible can directly execute existing x86_64 PoCs and exploits. Crucible accurately recreates multiple transient execution attacks, such as Meltdown [1], Foreshadow [2], ZombieLoad [4], RIDL [3], and Downfall [7] on unaffected Intel and AMD x86 machines. Researchers can thus examine these vulnerabilities on commodity hardware with unprecedented accessibility.

We validate our simulations by running unmodified[1] PoCs from original authors and the community [1], [3], [4], [7], [16], comparing results against known ground truths. Importantly, our evaluation is conducted on hardware that is unaffected by these underlying vulnerabilities, demonstrating that Crucible accurately simulates their effects. The results confirm that our simulations reproduce leakage characteristics and operational patterns, enabling meaningful, controlled, and repeatable experiments. In addition to PoCs, we also evaluate end-to-end exploits: leakage of AES keys with Downfall [7], a KASLR break with EchoLoad [17], and extracting the hard disk encryption key from VeraCrypt via Meltdown [18]. Despite relying on diverse underlying vulnerabilities and exploitation techniques, these complex attacks work with Crucible out of the box.

We also evaluate the suitability of Crucible for testing and comparing black-box CPU vulnerability fuzzers. For this purpose, we combine it with Transynther [6], a state-of-the-art CPU fuzzer that originally targets vulnerable hardware. The results show that it can identify and characterize our simulated vulnerabilities with results closely mirroring their performance on real systems. Moreover, our framework allows us to simulate artificial vulnerabilities, helping to judge whether fuzzers overfit to known vulnerabilities or are generic enough to find previously unknown vulnerabilities as well. We inject two artificial transient execution attacks, only one of which is successfully discovered by Transynther [6]. We thus show that Crucible can help evaluate the strengths of microarchitectural fuzzers, as well as their limits.

Overall, our platform makes transient execution vulnerabilities reproducible on any recent x86_64 CPU, enabling comparative research and rapid experimentation [19]. This environment also preserves attack techniques in a working state, ensuring a reproducible repository of CPU flaws that can guide and inspire the development of more robust, next-generation microarchitectures, and also ease the learning process for newcomers in this field. Moreover, the framework's modular design supports an extensive set of use cases–ranging from safe archives of bug behavior to advanced educational tools–ultimately democratizing CPU vulnerability research.

1. We preserve leakage code but adjust Flush+Reload thresholds as documented.

Contributions. The main contributions of this work are:

1) We devise a generic method to transparently simulate the effects of any Meltdown-type hardware vulnerability on commodity x86_64 CPUs for unmodified applications.
2) We create Crucible, an implementation of our approach that works with dynamically linked ELF binaries. Crucible can accurately reproduce the leakage profiles of many hardware vulnerabilities, including Meltdown, RIDL, ZombieLoad, and Downfall.
3) We extensively evaluate Crucible on proof-of-concept implementations, end-to-end exploits, and black-box CPU fuzzers, using CPUs unaffected by Meltdown-type leakage. In all experiments, the behavior closely aligns with what is observed on vulnerable hardware.
4) We simulate vulnerabilities to showcase Crucible's utility in evaluating and improving CPU fuzzers.

Structure. The paper is organized as follows. Section 2 provides the background required for the remainder of the paper. Section 3 introduces our simulation approach for transient execution vulnerabilities. Section 4 presents Crucible, our proof-of-concept implementation of this approach. In Section 5, we evaluate Crucible against benchmarks, proof-of-concept implementations, and real-world exploits. Section 6 discusses limitations and related work. Section 7 concludes.

Availability. We release our code as open source upon paper acceptance.

2. Background

In this section, we provide the background necessary for the rest of the paper.

2.1. Out-of-order and Speculative Execution

Out-of-order execution is a performance optimization that reorders instruction execution in the CPU while committing results in order. This allows instructions to be executed as soon as their input operands and required resources are available, maximizing utilization of the processor's hardware facilities and avoiding unnecessary pipeline stalls. However, an instruction's effects only become visible to the architectural state after earlier instructions have retired, preserving the semantics of in-order execution. *Speculative execution* extends this mechanism by executing instructions along a predicted control or data path, hence executing instructions before their input is known with certainty. This can occur, for instance, while the CPU waits for the results of a high-latency operation. Instead of stalling, it heuristically predicts the expected result, potentially influencing the direction of a branch or the target of an indirect jump. If the prediction turns out to be correct, the speculative results are committed to the program state and become visible; if not, the state is rolled back to a consistent point, discarding any effects of the misspeculated instructions. Speculative execution thus allows the CPU to avoid stalling while waiting for unresolved control or data dependencies. *Transient execution* exclusively refers to the execution of instructions that are

eventually squashed–either because they were executed out-of-order after a faulting instruction or speculatively after a misprediction. While their architectural results are discarded, the microarchitectural effects of such instructions may persist. For instance, transient execution is known to influence the state of caches [1], [20], branch predictors [21], and execution units [22]. These effects are invisible at the architectural level but can be indirectly observed using side channels [16]. Transient execution occurs in a variety of situations [23], including branch mispredictions, memory disambiguation errors, and delayed exception handling.

2.2. Transient Execution Attacks

Transient execution attacks exploit the brief window during which the CPU executes transient instructions, i.e., instructions that never retire architecturally. The attacks are broadly categorized based on the source of transient execution. *Spectre-type* attacks exploit speculative execution after control or data flow mispredictions. By selectively mistraining the branch predictors governing speculative execution, they cause the victim to transiently execute instructions that access sensitive data [20]. The effects of this, e.g., on the cache state, are observable with side channels, thus revealing this sensitive data to the attacker. Importantly, the accessed data must be architecturally valid for the victim, but security checks are bypassed transiently [20]. In contrast, *Meltdown-type* attacks exploit out-of-order execution that continues past a faulting instruction. Even though the instruction should trigger an exception (e.g., a page fault), subsequent instructions may be executed transiently before the fault is architecturally raised. In this window, the CPU may compute on architecturally inaccessible data, which can be exposed through microarchitectural side effects. Thus, Meltdown-type attacks break hardware-enforced privilege boundaries and can be used to leak kernel or hypervisor memory from unprivileged user code [16]. The microarchitectural source of this transiently used data differs between attacks. Foreshadow [2] and the original Meltdown attack [1] work on transient leakage from the CPU's Level 1 data cache (L1d). Microarchitectural Data Sampling (MDS) attacks [3], [4], [5], [6], [7] exploit leakage from other buffers, like the load and store ports or the line-fill buffer. A lesser-known variant, called Meltdown 3a or Meltdown-CPL-REG [16], [24], leaks values from privileged system registers.

2.3. Intel TSX

Intel TSX [25], introduced in 2013 as an ISA extension, enables speculative execution of atomic memory transactions to minimize synchronization costs in multithreaded workloads. Transactions are defined by dedicated instructions and offer atomicity by deferring updates until successful completion. If an interruption or conflict occurs, e.g., a concurrent modification or exception, the CPU discards all intermediate changes and reverts to the prior state. Originally a performance feature, TSX became a valuable tool for attackers, as it suppresses faults during transactional execution. Thus, it has been used as a building block in transient execution attacks such as Meltdown [1], RIDL [3], and ZombieLoad [4]. Moreover, with TSX Asynchronous Abort (TAA) [26], TSX also introduced a transient-execution vulnerability. Due to these security risks, Intel deployed a series of microcode patches disabling TSX [27], and removed the feature entirely in its 10th-generation Core processors [25].

3. Simulating Transient Execution

In this section, we introduce the core contribution of our paper: a *software-only* technique that (from an attacker's perspective) makes an unaffected x86_64 processor behave as if it were vulnerable to Meltdown-type vulnerabilities. This technique enables researchers to simulate Meltdown-type attacks on any x86_64 CPU. We outline the design goals in Section 3.1, introduce our shadow-stream mechanism in Section 3.2, and describe how we reproduce microarchitectural side effects in a configurable way in Section 3.3.

3.1. Design Goals and Overview

Our primary objective is to let existing PoCs and full exploits run *unchanged*. To that end, the simulation must (i) leave the attacker's architectural state intact, (ii) manifest *matching* cache and buffer footprints that genuine transient execution would leave [16], and (iii) terminate after around n instructions, approxmating a reorder buffer [23], [28]. Moreover, the approach should be implementable without kernel modifications to make it easily usable and distributable. Note that we do *not* aim for a generic emulation that handles all corner cases, such as loops or nested speculation during out-of-order execution, as long as Crucible works reliably for PoCs. Section 4 shows our proof-of-concept implementation, Crucible, that fulfills these goals.

We focus on Meltdown-type attacks, i.e., attacks where the root cause is a fault [16]. Hardware behavior specific to Meltdown-type vulnerabilities manifests in the transient path, which is invisible to the attacker program. The attacker only observes the side effects of transient execution on the microarchitecture. Hence, by artificially applying these side effects to the microarchitectural state, we can replicate the effects of a Meltdown-type vulnerability on unaffected hardware, even if no transient execution occurs. Thus, our primary challenge is to devise a method for modifying the microarchitectural state that is both *accurate*, i.e., it changes the state in the same way as a vulnerability would, and *generic*, i.e., applicable to all Meltdown-type vulnerabilities.

We illustrate a high-level overview of our approach in Figure 1. When a fault occurs, we create an *architectural shadow stream* by duplicating the instruction stream and executing it in a separate process. To induce Meltdown-type leakage, we then emulate the faulting instruction in the child, and continue executing the dependent instructions. We resume the original process on the same CPU core without modifying its state. The simulated transient path, like a real attack, leaves no architectural effects but alters the microarchitectural state that the original process can recover.

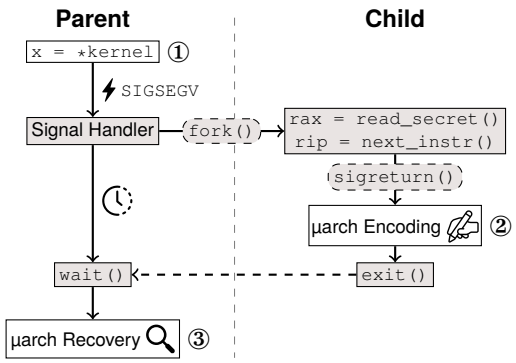

Figure 1: A high-level overview of our simulation approach. Newly introduced components are highlighted in gray. When a fault occurs ①, we spawn a child process that emulates leakage (`read_secret()`). The child encodes this data into the microarchitecture ②, allowing the parent to recover it ③.

```
1  char test_buf[256 << 12] = {0, };
2
3  void leak(size_t addr) {
4    for (int i = 0; i < 256; i++)
5      flush(&test_buf[i<<12]);
6
7    int x = kernel[addr] & 255;  ① Illegal Access
8    access(&test_buf[x << 12]);  ② Cache Encoding
9  }
10
11 void handle_fault() {  ③ Cache Recovery
12   for (int i = 0; i < 256; i++) {
13     int time = access_time(&test_buf[i << 12]);
14     if (time < THRESHOLD) {
15       printf("Leaked %d\n", i);
16     }
17   }
18 }
```

Listing 1: Typical structure of a Meltdown exploit with signal handlers and Flush+Reload. The attacker loads a value from kernel memory ①, and transiently encodes it into the cache ②. When handling the fault, they recover the value with a timing side channel ③.

3.2. Shadow-Stream Execution

In Meltdown-type attacks, the vulnerable CPU first raises an exception (e.g., a page fault) yet continues to execute a *transient* path that consumes the inaccessible data and encodes it into a microarchitectural element, such as the cache. We emulate this behavior by *forking* the process at the moment a fault is delivered (Figure 1). We install a high-priority signal handler to override any handler the application may have registered. On receiving the signal, the handler[2] spawns a child process and pauses itself. The child "replays" the faulting instruction but substitutes its result according to the selected vulnerability profile (e.g., kernel data for Meltdown, line-fill buffer entries for RIDL/-ZombieLoad). Additionally, it replaces fences and known serializing instructions (e.g., `cpuid`) in memory with a trapping instruction to stop the shadow stream when reaching them. The child process executes up to n instructions after the faulting instruction, where n is configurable to mimic the reorder-buffer depth of different microarchitectures [28]. When the instruction budget is exhausted, or execution reaches a serialising instruction, the child process destroys itself. Loops can be handled similarly using PMCs (cf. Section 6.2). No architectural changes flow back to the original process, but any microarchitectural state, such as the cache state, *does* persist because we ensure that both processes are scheduled on the same physical core. The original process resumes, runs the program's ordinary exception handler (or TSX fallback path), and finally probes the microarchitectural state just as it would on real hardware.

Illustrative Example. To illustrate the approach, consider its effects on the code in Listing 1. The initial fault occurs at ①, splitting execution into a parent and a child process. In the child, we resolve a value to leak, and place it into the register corresponding to `x`. The child then executes the cache encoding step in ② and terminates. Finally, we pass control to the parent's fault handler, allowing it to recover the value in ③.

The primary advantage of this method is that it enables the leakage of arbitrary data for a wide range of Meltdown-type vulnerabilities. Furthermore, it is highly accurate, as the instruction sequence executed in the child matches that of a real transient execution path. Finally, the attacker code can remain completely unchanged. Our approach thus requires no binary-level code modification, enabling our method to work without access to source code.

3.3. Buffer-based Leakage Simulation

Another challenge in our approach is choosing the correct values to leak when a fault occurs. For real-world vulnerabilities, this depends on which values are present in specific microarchitectural buffers, such as the L1d cache or line-fill buffers. Hence, accurately simulating these vulnerabilities also requires keeping track of the state of these buffers. For the L1d cache, as exploited in, e.g., Meltdown and Foreshadow, it is possible to probe the cache state directly through memory access timings. Values can be obtained directly via a kernel module, as these attacks leak from a virtual or physical address, respectively. However, the buffers involved in MDS attacks cannot be easily probed by software and are poorly documented. Therefore, to simulate MDS attacks, we *simulate* these buffers and the conditions under which victim data may reside in them. By dynamically instrumenting a victim binary, we track all instructions that interact with the memory hierarchy, including memory accesses and cache-maintenance instructions such as `clflush`. We feed this information into a model of the buffers we want to simulate, which may include, e.g., the L1d cache, load and store ports, a line-fill buffer, and the SIMD buffer exploited by Downfall. During shadow-stream execution, we query this model to obtain information on the

2. Implementation details are contained in Section 4.1.

buffer's contents and states, thus creating a highly accurate leakage profile.

3.4. Instantiating Vulnerability Configurations

In this section, we explain how the above mechanisms are specialized to reproduce the behavior of concrete transient execution vulnerabilities. At a high level, each *vulnerability configuration* consists of (i) trigger preconditions (what instruction/fault must occur), and (ii) a microarchitectural data source (what buffer provides the transient value). Across configurations, the shadow stream mirrors the instructions an attacker would rely on under genuine transient execution. At the same time, configuration-specific preconditions and data sources determine *what* is leaked and *when*. Thus, with such vulnerability configurations, it is possible to simulate a wide range of existing and also artificial vulnerabilities.

3.4.1. Defining Trigger Preconditions. On every signal, we inspect the faulting instruction and its cause to decide whether the event qualifies for a given configuration. Examples include: a user-mode load from a kernel virtual address (Meltdown), a present/reserved fault with a valid physical address in the page-table entry (Foreshadow/L1TF), a user-mode execution of privileged instructions such as `rdmsr`/`rdpmc`/`mov` to/from control or debug registers (Meltdown 3a), or a fault during a `gather` operation (Downfall). Only if the preconditions match, we continue the simulation of the transient execution and enter the shadow stream. Otherwise, there is a fallback to the default signal handler (if one is installed).

3.4.2. Selecting Microarchitectural Data Sources. The second aspect of the vulnerability configuration is the data source, which determines the values leaked by the shadow stream. Data sources can be genuine data sources, such as memory content, simulated content, e.g., from the buffer simulation, or programmatically generated values.

Genuine Data Sources. For attacks that leak from the L1d (Meltdown, Foreshadow), we can directly read the data, e.g., via a kernel module, and use this data in the shadow stream. Additionally, for a leakage simulation that is closer to the real vulnerabilities, we can verify whether the target cache line is cached by timing the memory access first. We thus only leak value if their addresses are cached in the CPU's cache. While this can lead to false negatives, i.e., a target is wrongly classified as not being cached, this still leads to leakage that resembles leakage from real attacks, which are not always successful either [4].

Simulated Data Sources. For MDS and Downfall, where leakage originates from line-fill buffers, load/store ports, or the SIMD buffer, we rely on the buffer model described in Section 3.3. The model exposes victim values consistent with published attacks, and constrains attacker control to the documented degrees of freedom (e.g., byte/offset control but not higher address bits).

Programmatic Data Sources. Meltdown-type vulnerabilities can also exhibit data leakage not directly tied to architectural values. For example, Hofmann et al. [29] show transient leakage of an intermediate divider state when dividing by zero, and Canella et al. [17] show that the first hardware-based Meltdown fixes still leak '0' if a physical page backs the target virtual address. We support programmatic data sources that return a configuration-defined value based on architectural and microarchitectural conditions, or alternatively "stall" by aborting the transient execution.

3.4.3. Vulnerability Configuration Examples. To illustrate that our vulnerability configurations are sufficiently generic to cover a wide range of vulnerabilities, we discuss concrete examples of such configurations.

Meltdown, Foreshadow-OS. The original Meltdown attack [1] exploits faults due to the U/S bit and transiently forwards kernel data from the L1d. Similarly, Foreshadow [2] transiently ignores the absence of the present (P) bit and loads the referenced physical address from the L1d. On such a faulting load, the configuration checks whether the cause of the fault matches with Meltdown (U/S bit not set) or Foreshadow (P bit not set). The faulting load is then replayed in the child, substituting the result with a dummy secret (user-space-only simulation) or the actual value obtained via a simple kernel module. For Meltdown, which targets virtual kernel addresses, this kernel module can directly dereference the target address. For Foreshadow, the it uses the direct-physical map in Linux to read from the physical address provided by the page-table entry.

Measuring the access time to the value can optionally be enabled to detect whether the target line is cached. Additionally, mitigations can be simulated by masking the value before returning it or "stalling" (i.e., aborting the shadow stream) [17].

Meltdown 3a (Privileged Instructions). Meltdown 3a (a.k.a. Meltdown-CPL-REG) transiently uses results of privileged instructions executed in user mode [16], [24]. Upon a fault from instructions such as `rdmsr` or `rdpmc`, the configuration defines that the instruction requires reexecution in the kernel context via a small kernel module (or consults OS interfaces for MSR/PMC values) and injects the obtained value into the child's registers, which then performs the usual microarchitectural encoding. Thus, this simulation can use a genuine data source. Alternatively, values, especially control and debug-register values, can also be supplied via pre-configured rules using a programmatic data source.

MDS (RIDL/ZombieLoad/Fallout). MDS-class attacks [3], [4], [5], [6] leak from memory-subsystem queues rather than the L1d. This requires a simulated data source based on the buffer model: the model is queried and injects a matching value into the shadow stream. As on real hardware, the attacker can often steer the line offset but not higher address bits, a constraint we also consider in our model.

Downfall (SIMD Buffer + `gather`). The Downfall attack [7], also referred to as Gather Data Sampling (GDS), leaks from the SIMD buffer used by SSE/AVX memory operations when a faulting `gather` transiently forwards

stale vector data. For this vulnerability configuration, we track vector loads in the buffer model and, on a qualifying `gather` fault, inject the buffered SIMD data into the shadow stream. In Section 5, we demonstrate that a simulated data source successfully reproduces the behavior of vulnerable hardware in proof-of-concept implementations and even end-to-end exploits.

Divider State Sampling (DSS). Hofmann et al. report transient data leakage following faults caused by divisions by zero on Intel CPUs [29]. While the exact mechanisms behind this leakage are unknown, we can approximate it by injecting leakage from a programmatic data source after a zero-division.

4. The Crucible Framework

In this section, we present Crucible, a proof-of-concept implementation of the transient execution-simulation approach introduced in Section 3. In Section 4.1, we provide a high-level implementation overview. Section 4.2 presents the unique execution environment of Crucible's transient window, Section 4.3 explains the buffer model for simulating MDS attacks. Section 4.4 explains how Crucible emulates the functionality of Intel TSX on any hardware, allowing for the simulation of TSX Asynchronous Abort (TAA) vulnerabilities. Finally, Section 4.5 describes how we influence performance counters to behave similarly to real attacks.

4.1. Implementation Overview

The primary design goal of Crucible is to allow for the transparent simulation of Meltdown-type transient execution vulnerabilities without modifying the attacker code. All simulation parts are implemented in a shared user-mode library, meaning that Crucible can run entirely in user space. However, Crucible also ships with an optional kernel module that allows access to kernel addresses and physical addresses if a vulnerability configuration defines a genuine data source. Crucible is highly configurable, allowing users to provide multiple vulnerability configurations at once. This makes it possible to simulate multiple vulnerabilities at the same time, including combinations that do not exist in real hardware. Additionally, Crucible supports both signal handlers and TSX (even if the CPU has no support) for handling faults in target applications, and can provide simulation of performance-counter events.

We implement Crucible as a shared user-space library for Linux on x86_64. Using the preloading functionality of the GNU C library's dynamic linker, we can insert this library into the address space of any dynamically-linked ELF executable. Crucible intercepts functions that register signal handlers and ensures that its own signal handler for segmentation faults (`SIGSEGV`), illegal instructions (`SIGILL`), and floating-point exceptions (`SIGFPE`) are registered first. Inside each fault handler, we first decode the fault context, including address, opcode, and exception information, then compare it against the active vulnerability configurations. If a configuration matches the fault, we start the shadow stream to simulate transient execution (Section 4.2) and wait in the parent until the simulation is finished. Otherwise, control continues directly to any handler installed by the target program.

All simulator parameters, including reorder-buffer depth (Section 4.2), buffer sizes (Section 4.3), supported fault handling (Section 4.4), or performance-counter simulation (Section 4.5), are configurable via a YAML file parsed at start-up. This configuration may also reference external plug-ins, such as drop-in replacement buffer models or a custom PMC simulation model. Crucible loads these plug-ins with standard dynamic-loading facilities.

4.2. Shadow-Stream Runtime

On each fault, Crucible's handler first consults the active vulnerability configurations. If a configuration matches the fault, it then disassembles the faulting instruction using Capstone [30], forks, and pauses the parent process. Next, the child resolves any data that the original instruction would have provided transiently. To that end, the handler invokes the active leakage source, which returns a blob of up to 64 bytes (i.e., one cache line). The handler injects the blob directly into architectural state, choosing the correct general-purpose or SIMD register and, for sub-register destinations, masking only the relevant part to avoid collateral damage. This process leaves the remaining program state intact, so subsequent instructions in the window see precisely the values that a real vulnerability would have supplied.

Before resuming execution, the child also installs *termination patches*. These termination patches are single-byte trap instructions (`ud2`) at points where the transient execution should stop. The maximum length of the transient execution is configured in Crucible's configuration file, and defaults to 192 instructions. Additionally, depending on the configuration, it replaces every serializing instruction in the window with a termination patch. In line with Intel's documentation [25], our implementation currently supports `lfence`, `mfence`, `sfence`, and `cpuid` as serializing instructions. Crucible utilizes its fault handler to handle the execution of termination patches correctly.

These measures guarantee that the total number of architecturally executed instructions never exceeds the configured budget while still allowing stacked faults just as on real hardware. When the child hits a trap, it exits, prompting the parent to wake up and resume the application's regular exception path. As parent and child share the entire cache hierarchy, all microarchitectural modifications created by the child persist, allowing attacker code to observe them.

To support the nested faults sometimes found in Meltdown-type attacks, Crucible also handles faults inside the child. For instance, some implementations first access a `NULL`-pointer before accessing protected memory, as this can extend the transient window length on some CPUs [1], [28]. Hence, Crucible's child process cannot simply terminate when a fault occurs. Instead, it handles nested faults in the same way as faults in the parent process, leaking data as

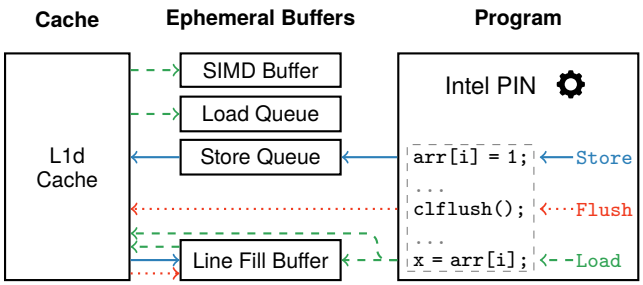

Figure 2: High-level illustration of Pin's interaction with our default buffer model. Victim programs can trigger a store (solid line), load (dashed line), or flush (dotted line) path. The arrows indicate which buffers are affected by a specific path, and in which order. The load path may skip the line-fill buffer in case of an L1d hit.

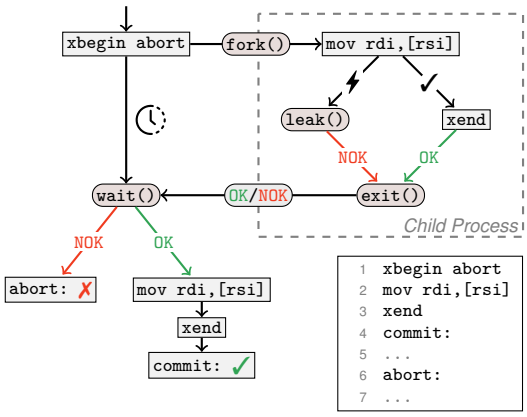

Figure 3: An illustration of Crucible's TSX emulation on a simple code example. Crucible first executes the transaction in a child process, simulating leakage when encountering faults. Based on whether the child transaction succeeds, it then aborts (NOK) or commits (OK) the transaction in the parent process.

configured by the user. However, nested faults do not create another child process.

As the shadow stream executes the instructions architecturally, we additionally implement measures to avoid (accidental) architectural exposure of the leaked values, e.g., by system calls. For this, we rely on seccomp-bpf [31] to block all syscalls, matching the behavior of transient execution on real hardware.

4.3. Pluggable Buffer Model

Some vulnerabilities leak from structures that software cannot observe directly [3], [4], [5]. To cover such cases, the library can load an external buffer-simulation module via `dlopen`. Our default module maintains an 8-way, 64-set L1d cache, a line-fill buffer, separate load and store queues, and one SIMD buffer per core. The simulated replacement is pseudo-LRU, and lifetimes match reverse-engineered characterisations of Intel's client cores [4]. Users can individually enable or disable leakage from each of these buffers, making it possible to simulate a variety of vulnerabilities. For example, users can enable leakage from the line-fill buffer for simulating RIDL while keeping leakage from the L1d cache disabled. With Downfall, they would instead enable the SIMD register buffer and disable leakage from other buffers.

To populate the simulated buffer with values, we employ Intel Pin [32] for binary instrumentation on the victim. With every memory access, the instrumented victim feeds data into a dedicated *store*, *load*, or *flush* path of a buffer model, triggering a series of updates (see Figure 2). For instance, when a load misses the L1d cache, our default model's load path creates an entry in the line-fill buffer, the L1d cache, and the load queue. With loads from vector instructions, we additionally update the SIMD buffer. In contrast, flushing or evicting a cache line only updates the line-fill buffer. To approximate the leakage behavior observed on real hardware, we model all buffers except for the L1d cache as *ephemeral* buffers, which only keep their entries until the next memory access. Thus, the attacker has only a short time window to leak them. To inspect the model's state, Crucible uses a separate interface to query individual buffers for candidate values when the shadow stream requires data from a particular buffer class.

While this implementation is only an approximation, it is sufficiently accurate to simulate MDS vulnerabilities (cf. Section 5.2). Implementing a more accurate buffer simulation model is only possible with full knowledge of these details, which is out of scope for Crucible. However, we design Crucible's interface to make the buffer simulation model easy to replace, thus enabling users to insert their own buffer model as a shared library. Hence, it is still possible to run Crucible with a fully accurate model if the implementation details become publicly known, or a hardware manufacturer wishes to utilize Crucible internally.

4.4. TSX Support

For attacker code relying on the Intel TSX extensions, Crucible contains an optional mechanism to transparently emulate the behavior of Intel TSX. Crucible thus works with code using TSX for fault suppression, even without TSX support in the underlying hardware. Furthermore, this allows it to simulate TAA vulnerabilities, such as the TAA variant of RIDL and ZombieLoad v2.

Crucible's TSX emulation, which is illustrated in Figure 3, also relies on user-mode signal handlers. If the underlying CPU does not support TSX, either because it is unavailable or disabled, `xbegin` triggers `SIGILL`. Crucible intercepts this signal and treats the fault like other transient-window entry points, creating a child process. Crucible then treats this fork in the same way as a speculative window, simulating leakage for instructions that would otherwise cause a TSX abort during regular execution.

When the child process terminates after simulating a TSX abort, we jump to the fallback address specified in the initial `xbegin` instruction in the parent process, resuming execution from there. If the child instead reaches an `xend` instruction without aborting, we simulate a TSX commit. Crucible does this by resuming parent execution at the instruction immediately following the initial `xbegin`. This way, the parent process executes the same instruction sequence as the child, causing the same changes to the program state. When the parent reaches the `xend` instruction, Crucible's signal handler skips ahead to the next instruction, ignoring the `xend` instruction.

Naturally, however, this mode of emulation does not fully capture the microarchitectural effects of TSX. For instance, a TSX abort caused by a cache conflict is not emulated. As a workaround, Crucible can be configured to abort TSX transactions after a given number of memory accesses or when a specific address is accessed. Given the knowledge of where a transaction *should* abort, users can therefore still use it to prototype TAA attacks. In Section 5.2, we show that this emulation strategy can successfully reproduce proof-of-concept attacks for ZombieLoad v2 and RIDL's TAA variant.

4.5. Performance-Counter Interface

Crucible can optionally be configured to simulate performance-counter behavior during shadow-stream execution. The motivation is that several tools for discovering or observing transient execution attacks rely on performance counters for guidance, or for inspecting the effects of transient execution [6], [23], [33]. Users can configure a *PMC model*, which is implemented as a shared library that is easily replaceable. When a fault occurs, Crucible stops all active performance counters and informs the PMC model about the fault type, leakage mode, program state, and the instructions executed in the transient window. Based on the expected behavior given this information, the model then updates the PMC values in hardware. Finally, before execution resumes in the parent process, Crucible re-enables the performance counters it previously disabled, resuming profiling with the updated values. The program can then recover the PMC values, allowing it to make inferences about Crucible's simulated transient behavior. To make this generic, Crucible itself only provides an interface for PMC simulation, leaving the PMC model to be implemented by the user. We demonstrate the accuracy of a PMC model for Intel's divider units in Section 5.1.

5. Evaluation

In this section, we evaluate Crucible along four axes: (i) microarchitectural fidelity, i.e., does the simulated leakage match the measurable artefacts of genuine hardware flaws? (Section 5.1) (ii) functional coverage, i.e., can Crucible run the existing corpus of public proof-of-concept (PoC) programs and full exploits unmodified? (Sections 5.2 and 5.3) (iii) research utility, i.e., does the framework enable new experiments, such as injecting hypothetical bugs or benchmarking fuzzers, that are difficult or impossible on physical CPUs? (Sections 5.4 and 5.5) and (iv) overhead, i.e., what is the performance impact on benign workloads (Section 5.6). All experiments are performed under Ubuntu 22.04 unless stated otherwise. Table 1 lists the test platforms.

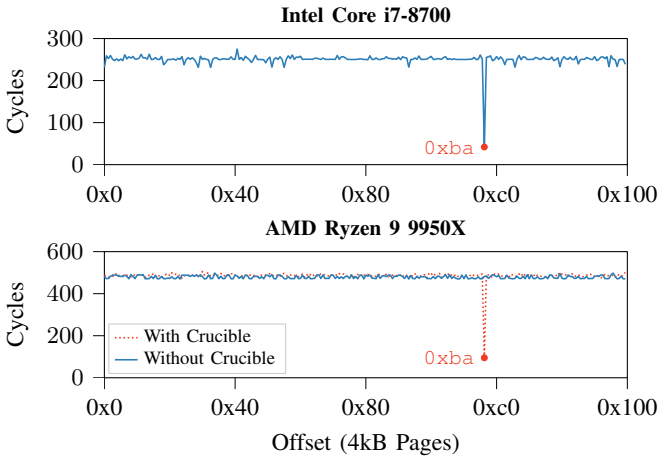

Figure 4: Access timings for buffer indexed with leaked kernel byte `0xba` (n=2000). Native leakage on Meltdown-affected CPU (top) is successfully replicated by Crucible on unaffected CPU (bottom).

Experimental Setup To ensure comparability across experiments, we fix the following parameters unless an experiment requires otherwise. We set the reorder-buffer budget to 192 instructions, and the buffer model mirrors an 8-way 64-set L1d with a 16-entry line-fill buffer, which is a good approximation of a wide range of (Intel) CPUs. To ensure that the Flush+Reload primitive in the original PoCs is reliable, we disable hardware prefetchers, turbo boost, and microarchitectural attack mitigations, fix the CPU frequency, and pin the benchmark to a dedicated core.

5.1. Microarchitectural Fidelity Benchmarks

In this section, we evaluate how faithful the simulation of Crucible is. We evaluate the effect of loads in the transient execution on the cache state, measure the leakage rate, and compare PMC effects to real attacks.

5.1.1. Transient Cache Effects. An essential part of Crucible's simulated transient execution is the influence of the shadow-stream execution on the microarchitectural state. As with real vulnerabilities, this allows data to be exfiltrated through, e.g., timing-based cache side channels. Hence, we verify that these modifications are accurate. We implement a simple test program that exploits Meltdown with Flush+Reload to leak a single byte from kernel memory. Using a kernel module, we ensure that a single kernel address is always cached, which is a prerequisite for Meltdown leakage. Our test program reads from this kernel address and transiently encodes its value into the cache by using it

as an index for accessing a buffer, as described by Lipp et al. [1]. After handling the fault, we measure the memory access timings for this buffer, recovering the index at which the value is cached.

Figure 4 compares Flush+Reload access times on an Intel Core i7-8700 (Coffee Lake, Meltdown-vulnerable) and an AMD Ryzen 9 9950X (Zen 5, Meltdown-unaffected) with and without Crucible. As expected, we observe that the access time for the offset corresponding to the leaked byte is significantly lower on the Core i7-8700, but not on the Ryzen 9 9950X. However, if we run the same program using Crucible's Meltdown simulation, the access times on the Ryzen 9 9950X show the same characteristic dip at the expected offset. This shows that the effects introduced by Crucible closely match the behavior of the real vulnerability on an affected CPU. While Crucible necessarily touches a small, stable set of lines in handler code, the leakage signal is unaffected by that. Crucible does not cache any lines used in the probe buffer, but potentially only evicts them. For Flush+Reload, the probe buffer has to be fully evicted before the leakage. Thus, there is no negative effect.

5.1.2. Leakage Rates. In this experiment, we measure the data leakage rate achieved by Crucible's simulation. We use the same methodology as in the previous experiment and run the exploit until it has leaked 64 kB of kernel memory. As leaking a single byte with Meltdown may require multiple attempts [1], we repeat the leakage procedure for each byte until we observe at least one cache hit in our index buffer. On the affected Intel Core i7-8700, we measure a leakage rate of 14.6 kB/s with an average of 2.5 leakage attempts required per byte and an error rate of 5.5 %.

Using Crucible's Meltdown simulation on an unaffected Ryzen 9 9950X (Zen 5), we observe a leakage rate of 0.25 kB/s. Our exploit only repeats 357 leakage attempts, resulting in a total number of 65 893 leakage attempts for leaking 65 536 bytes. Furthermore, there is no error in any of the leaked bytes. We observe similar numbers using Crucible on an Intel Core i9-13900K (Raptor Lake), leaking at 0.26 kB/s with 0.1 % error rate, and all except for 61 out of 65 555 leakage attempts succeeding.

While leaking a byte with Crucible is significantly slower than the actual Meltdown leakage rates, it is still sufficiently fast to run end-to-end attacks at comparable leakage rates (cf. Section 5.3). The slowdown is unsurprising given that Crucible spawns a new process on every fault and waits for its execution to complete. However, as most Meltdown-type attacks target small secrets, such as cryptographic keys [18], KASLR offsets [1], or password hashes [3], this is not a practical limitation. Additionally, the leakage induced by Crucible is more stable, i.e., leaking fewer incorrect bytes and requiring fewer leakage attempts. This is to be expected, as we configure Crucible to inject only kernel memory, without buffer simulation for MDS. However, in Meltdown-affected hardware, these vulnerabilities coexist. If realism is desired, Crucible can replicate this effect with its buffer simulation. Thus, Crucible can be configured to require a higher number of leakage attempts from the exploit by adjusting its leakage profile, intentionally making the leakage less stable but even more realistic.

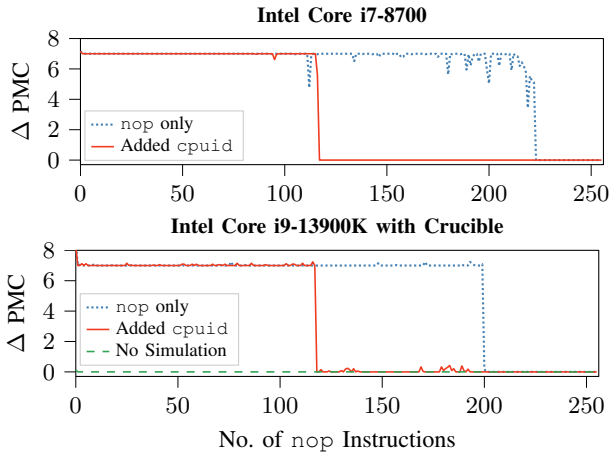

Figure 5: Average difference of the ARITH.DIVIDER_ACTIVE PMC after transiently executing a nop-slide of a specific length, terminated by vsqrtpd (n=100). In one series, we insert a cpuid instruction after 120 instructions. Crucible correctly simulates the PMC update in transient execution and also respects the serializing cpuid instruction.

5.1.3. PMC Effects. In this experiment, we demonstrate that Crucible's performance-monitoring counter (PMC) simulation can accurately inform a program about its transient behavior. We create a program that uses the Intel performance-monitoring event ARITH.DIVIDER_ACTIVE to measure the length of a transient window as shown by Weber et al. [34]. This program induces a fault by dereferencing a NULL-pointer, before transiently executing a given number of instructions, e.g., , nops or xors. The next instruction is a vsqrtpd instruction, which increments the ARITH.DIVIDER_ACTIVE counter even when executed transiently. Hence, by reading the PMC after handling the fault, we can determine whether the instruction was transiently executed. Thus, we can increment the number of instructions until we no longer see active cycles recorded by the PMC to determine the length of a transient window. To simulate ARITH.DIVIDER_ACTIVE with Crucible, we create a PMC model that increments this PMC by a fixed value whenever it encounters an instruction triggering the floating point or integer division unit. The fixed value is determined by profiling on actual hardware. For example, we profile that the division unit on Intel Coffee Lake CPUs is active for 7 cycles when executing a vsqrtpd instruction, thus Crucible increments the counter by 7 every time this instruction is encountered in the shadow stream.

We run our program on a Meltdown-affected Intel Core i7-8700 (Coffee Lake), as well as an unaffected Intel Core i9-13900K (Raptor Lake) CPU with Crucible. The results are shown in Figure 5. As expected, the Core i7-8700

TABLE 1: Crucible's effect on PoC implementations. We reproduce (●) the behavior of affected CPUs on both Intel and AMD processors, except when missing required hardware features (-).

Class	Program	Core i7-10710U	Core i9-12900HK	Core i9-13900K	Ryzen 7 5700G	Ryzen 7 PRO 7840U	Ryzen 9 9950X
Meltdown + L1TF	memdump [1]	●*	●	●*	●	●	●
	US/poc_x86 [16]	●	●	●	●	●	●
	P/poc_x86 [16]	●	●	●	●	●	●
	PK/poc_x86 [16]	-	●	●	●	●	●
MDS	pocs/ridl_basic [3]	●	●	●*	●	●	●
	attacker/leak_v1 [4]	●	●	●*	●	●	●*
	attacker/leak_v2 [4]	●	●	●*	●	●	●*
Downfall	gds_test/gds [7]	-	-	-	-	●	●

* With re-tuned timing thresholds

transiently executes past the NULL-pointer access. When iteratively adding one nop instruction at a time, we observe that the PMC value stops changing after 223 nop instructions have been added, thus revealing the length of the transient window. We further use this experiment to verify the serializing behavior of the cpuid instruction. When inserting a cpuid instruction after 120 nop instructions, we observe no further increase in PMC values beyond this point. On Core i9-13900K, we observe no effect on the PMC, as expected, indicating that the CPU does not transiently execute past the NULL-pointer access. When enabling Crucible's simulated transient window, configured to a maximum of 200 instructions, the PMC differences we observe are near-identical to the ones observed on the Core i7-8700. The only observable difference is that on real hardware, the transient window's cutoff point is less strict, with transient execution terminating early in some instances. However, extending Crucible to simulate this behavior as well is merely a minor engineering task.

5.2. Proof of Concept Implementations

To validate the correctness of Crucible's simulation, we test it on 8 unmodified proof-of-concept (PoC) implementations of various transient execution attacks. This includes demos from the publicly available artifacts of Meltdown [1], RIDL [3], ZombieLoad [4], and Downfall [7], as well as a series of demos implemented by Canella et al. [16]. Furthermore, to demonstrate that Crucible's effects are consistent across hardware platforms, we run these demos on 6 CPUs by both Intel and AMD, none of which are affected by any of the simulated vulnerabilities. For some PoCs, we retune the covert-channel part, i.e., the part that recovers the encoded value from the microarchitecture. For example, the RIDL PoC has a hard-coded threshold to determine whether a memory access was a cache hit. When CPUs do not align with that threshold, it causes the demo never to record a cache hit, thus breaking its functionality. The only other modification we make is to the Downfall demo, which runs the attacker and victim programs in two threads of the same process. To allow for instrumentation using Intel Pin, we run this demo as two processes, without any further changes, thus also without any changes to the exploit itself. Each demo is executed 10 times and considered reproduced if its output indicates vulnerable hardware behavior at least once. This is in line with PoCs, as such attacks can typically be repeated until they are successful.

The results of this experiment are shown in Table 1. For nearly all PoC implementations and hardware platforms, Crucible successfully simulates the behavior of a vulnerable CPU with entirely unmodified code. Although some demos only succeed with minimal adjustments to the Flush+Reload threshold, we stress that this is not an issue in Crucible's simulation, but rather due to exploits being built without portability in mind and thus implementing unportable cache covert channels. Furthermore, this is also documented for some of the PoCs, which state in their documentation that the cache miss threshold might have to be set manually [16]. In all cases where the expected result could not be reproduced, this was due to limitations of the PoC rather than Crucible. For instance, we find that the Downfall PoC requires the AVX-512 ISA extension to work correctly, which is not supported on some platforms. On all platforms with AVX-512 support, this simulated demo works reliably. Similarly, the Core i7-10710U lacks support for Protection Keys, which are the mechanism exploited by PK/poc_x86.

5.3. End-to-end Attacks

We evaluate Crucible on 3 end-to-end attacks on a CPU not affected by any of the respective exploited transient-execution vulnerabilities. First, we test a VeraCrypt hard disk-encryption key recovery attack via Meltdown [1], as shown by Gruss et al. [18]. Second, we demonstrate that partially-patched Meltdown variants can be simulated by running the EchoLoad KASLR break [17]. Finally, we reproduce an existing Downfall exploit [7] on the AES implementation of OpenSSL. Reproducing these exploits on an AMD Ryzen 7 PRO 7840U unaffected by any of the simulated vulnerabilities demonstrates Crucible's portability.

5.3.1. Meltdown on VeraCrypt. We target the VeraCrypt [35] Meltdown exploit[3] as presented by Gruss et al. [18], in which an attacker leverages Meltdown to read the disk-encryption key. On a high level, this exploit consists of the following stages: First, it uses Meltdown to locate the task struct of the VeraCrypt process in kernel memory, and leaks the root address of its paging structures. The exploit leverages Meltdown to traverse the page tables of the VeraCrypt process to get the physical address of the variable containing the secret key. Eventually, the key is

3. The exploit's source code was kindly provided by its authors.

leaked by mounting Meltdown on the previously recovered physical address through the kernel's direct physical map.

Setup. The exploit implements logic in Python and Meltdown attacks in C, with Python invoking the C binary. Crucible injection requires only setting LD_PRELOAD in Python and switching fault suppression from speculation to signal handlers—facilitated by preexisting implementation.

Evaluation. We run the exploit in an 8-core virtual machine running Ubuntu 16.04 with Linux 4.13.0 on an AMD Ryzen 7 PRO 7840U. Over 20 end-to-end experiments, the simulated attack recovers the key, on average, in 381 s, with a 100 % success rate. The successful extraction of the correct key after a series of Meltdown usages (kernel struct leakage, page-table iteration, reading via direct-physical map) confirms that our method reproduces the behavior of Meltdown in realistic use cases. The performance is also comparable, as Gruss et al. [18] showed a runtime of 163 s. This case study further demonstrates that researchers can leverage our tool to rapidly prototype such complex attacks.

5.3.2. Breaking KASLR with EchoLoad. The first hardware patches for Meltdown did not fix the root cause but only zeroed the leaked values, as analyzed by Canella et al. [17]. While this prevents data leakage, it still leaks metadata about the kernel address range: a virtual address pointing to kernel memory transiently returns '0', whereas a virtual address not backed by a physical address results in a stall. This behavior is exploited in the EchoLoad KASLR break [17].

Setup. We configure Crucible with a programmatic data source to emulate Meltdown with only '0' leakage. As EchoLoad uses signal handling, we do not require TSX emulation or simulated buffers. Instead, we run the EchoLoad binary with Crucible without any modifications.

Evaluation. We run the exploit 1000 times on an AMD Ryzen 7 PRO 7840U. We successfully break KASLR with EchoLoad in every run. On average, EchoLoad under Crucible takes 26.5 ms. While this is three orders of magnitude slower than the native implementation, the absolute time difference is still in the millisecond range. Thus, this shows that researchers can also use Crucible for evaluating real-world attacks that are otherwise hard to test, as only a small number of CPUs have this specific Meltdown vulnerability.

5.3.3. Leaking AES Keys with Downfall. Using Crucible, we reproduce a Downfall attack [7] to leak AES keys from OpenSSL's AES-128-CBC implementation. When loading the AES key from memory, this implementation utilizes the AES-NI instruction set extensions, and the key thus enters the SIMD buffer. The attack assumes that OpenSSL is continuously running in a parallel thread, which can occur, e.g., when encrypting large files. In this scenario, leaking the SIMD buffer often yields key material.

Since the cache side-channel employed by the original attack only allows for leaking 8 key bytes per leakage attempt, it consists of two parts. First, it collects a series of candidate values for the upper and lower halves of a 128-bit AES key by continuously leaking from the upper and lower 8 bytes of the SIMD buffer. For each half of the key, the exploit spends 5 s in this phase. Next, it combines the candidates' values into candidates for the full key. If both halves of the key are in the original candidate sets, this yields the correct key. To confirm which full key candidate is correct, the attack decrypts a known ciphertext.

Setup. The original exploit's shell scripts invoke C binaries for attack primitives, allowing Crucible injection through minimal shell modifications while leaving primitives unmodified. We instrument an unmodified OpenSSL 3.5.1 (Debian package) with Intel Pin. We increase leakage time to 30 s per key half to compensate for Pin overhead, resulting in total runtime of 60 s.

Evaluation. Running the attack 100 times with random keys on a Ryzen 7 PRO 7840U yields 74 % success, demonstrating Crucible's ability to reproduce intricate Downfall exploits and scale to complex applications like OpenSSL.

5.4. Artificial Vulnerabilities

To highlight how generic Crucible is, we simulate two additional vulnerabilities, which are not known to exist on any real-world hardware. First, we add a configuration that extends the ability of `gather` instructions to leak the SIMD buffer to all vector instructions. Hence, we can simulate a Downfall variant utilizing regular vector loads as a trigger point, eliminating the need for a `gather` instruction. Second, we add a Meltdown-type vulnerability that does not require any cache encoding. For this vulnerability, we simulate a secret-dependent runtime of the division instruction when the divisor is stored in (inaccessible) kernel memory. This vulnerability is inspired by the TSX-based timing side channel on kernel memory used to break KASLR [36]. Specifically, the runtime for our division instruction correlates with the Hamming distance of the attacker-provided dividend and the transiently read divisor. By measuring how long it takes to handle the exception or abort a TSX transaction, an attacker can leak information about the inaccessible secret. By incrementally adapting the dividend, an attacker can leak the secret bit-by-bit, similar to power-based side-channel attacks [37]. For both these artificial vulnerabilities, we implement PoC exploits to show that they can be exploited by leveraging Crucible.

5.5. Fuzzers

We demonstrate Crucible's fuzzer evaluation capabilities by integrating it into Transynther [6], a black-box CPU fuzzer for Meltdown-type attacks. We show that Transynther successfully discovers Crucible's MDS simulation on an AMD Ryzen 7 5700G, which is not affected by any known Meltdown or MDS variant. As expected, it does not discover Downfall and Meltdown 3a, despite the simulation of these vulnerabilities being enabled, as Transynther does not generate the required instructions. Furthermore, we test Transynther against the artificial vulnerabilities described in Section 5.4. It detects the modified SIMD buffer leakage, demonstrating Crucible's ability to test fuzzer capabilities through novel vulnerability emulation.

5.5.1. Test Setup. We integrate Crucible into Transynther's evaluation phase with minimal changes—modifying only the environment variable for synthesized code invocation. We configure Crucible for MDS-like vulnerabilities, enabling buffer simulation and leakage from all buffers except L1d cache, including the SIMD buffer (Downfall), Meltdown 3a, and artificial vulnerabilities from Section 5.4 Our only additional modification separates victim and attacker into distinct processes for Intel Pin instrumentation, replacing Transynther's original two-thread approach. All other components remain unmodified.

5.5.2. Results. We run Transynther for $10\,000$ test cases on an AMD Ryzen 7 5700G (Zen 3) with Ubuntu 22.04, Linux v5.15.0, and GCC v11.4.0, with hardware prefetchers disabled for stability. Each test runs for $5\,\mathrm{s}$, totaling approximately 14 CPU hours.

In 2586 test cases, Transynther's attacker code triggers Crucible's simulation. 1839 of these test cases trigger MDS, whereas the remaining 747 trigger leakage of the SIMD buffer due to a vector load (see Section 5.4). Transynther did not trigger our second artificial vulnerability, i.e., the timing leak of the division instruction, which is expected given how different it is from other transient execution vulnerabilities. As expected, it also fails to trigger Downfall or any Meltdown 3a variant, as they cannot be synthesized from the corpus that Transynther uses.

In 57 cases where leakage occurred, Transynther proceeds to leak the injected values in a noise-free way, confirming its ability to discover Crucible's artificial leakage automatically. Although this success rate is low, it is consistent with the original results of Moghimi et al. on vulnerable hardware [6], where 100 out of $46\,500$ test cases are reported to show interesting leakage patterns. Furthermore, 17 test cases successfully leak values from the SIMD buffer, showing that Transynther would likely have discovered Downfall if it had the `gather` instruction in its code generation.

Consistent with the original results, we observe that Transynther's evaluation phase has a high false-positive rate. In 4488 instances, Transynther reports leakage where none was artificially injected. While some of these cases can be attributed to the inevitable noise of the Flush+Reload side channel, we also observe many instances where Transynther reports trivial or benign leakage. For instance, 896 test cases with leakage do not even cause a fault, encoding values into the cache state during architectural instead of transient execution. This bypasses Crucible entirely and trivially causes leakage without any simulation being required. Furthermore, we observe test cases that cause a fault but encode values into the cache that are architecturally accessible. In such cases, Crucible still duplicates the instruction stream, but depending on the type of initial fault, does not necessarily inject artificial leakage. In this state, the child process can also encode architecturally accessible values into the cache state, allowing Transynther to leak them without requiring artificial leakage. While this mirrors the behavior expected on real hardware, it also increases Transynther's false positive rate. We observe similar behavior in test cases where Crucible triggered leakage that Transynther did not correctly recover. Here, the artificially injected values are discarded, with other values being encoded into the cache instead. The child process may also terminate before values can be encoded, leading to 856 instances where artificial leakage was injected, but Transynther recovered nothing.

These results lead us to two conclusions: First, they align with the results observed on vulnerable hardware, demonstrating that Crucible's artificial leakage closely resembles real vulnerabilities. Second, evaluating Transynther's performance on an artificial leakage model allows us to gain unparalleled insights into its strengths and limitations. For instance, we show that even though Transynther triggers leakage in roughly a quarter of its test cases, it only retrieves the leaked values correctly in less than $0.6\,\%$. We argue these insights are uniquely enabled by Crucible, proving its value for building future fuzzers.

5.6. Average Performance Overhead

Crucible uses process forking to emulate transient execution, which is slower than hardware-level fault suppression. These timing differences could cause exploits and programs with strict timing requirements to fail. However, overhead only occurs when signals like `SIGSEGV` trigger which is rare in benign software. SPEC CPU 2017 benchmarks [38] confirm negligible overhead: intspeed 7.43 (baseline: 7.42), intrate 68.30 (baseline: 68.14), fpspeed 52.75 (baseline: 52.60), and fprate 75.59 (baseline: 75.40).

6. Discussion

6.1. Related Work

Several recent efforts have explored CPU-level bug injection and emulation to evaluate vulnerabilities, but with different goals and types of vulnerabilities.

Emulating CPU Bugs. Swierzy et al. [15] built an RISC-V-based emulator that targets educational demonstration of transient-execution attacks. It provides a simplified, software-only CPU model that illustrates the steps of Meltdown and Spectre-style exploits in a controlled environment. However, in contrast to Crucible, it is fundamentally incompatible with real-world exploits because it uses a different instruction set architecture. Hammulator [39] is a DRAM-centric rapid prototyping simulator that injects Rowhammer bit flips into timing-accurate memory models.

Bug Injection. Dessouky et al. [40] introduce artificial bugs into RISC-V CPUs at the RTL level. These vulnerabilities form the basis of the Hack@DAC hackathon, where participants aim to identify and exploit them [41]. In contrast to our approach, the injected bugs are used to evaluate pre-silicon bug-finding techniques. Bölcskei et al. [42] demonstrate a similar approach by automatically injecting artificial bugs into CPUs, then assessing the coverage of CPU fuzzers. As with the former method, their approach requires CPU modification, which restricts their evaluation to RISC-V CPUs and prevents the evaluation of fuzzers targeting x86.

CPU Simulation. There exists a large variety of tools that simulate or emulate CPU behavior. At the architectural level, emulators such as QEMU [43] and Unicorn [44] enable one architecture to execute binaries compiled for another, with a variety of use cases in research, e.g., cross-architecture fuzzing [45]. To analyze performance and security properties of microarchitectural properties, frameworks such as gem5 [46] are typically used. By modeling pipelines, reorder buffers, and multilevel caches, gem5 can evaluate the impact of proposed CPU design changes, such as mitigations for transient-execution attacks. However, these tools typically do not emulate CPU vulnerabilities such as Meltdown-type attacks. More specialized tools, including uICA [47], allow fine-grained studies of instruction latency and throughput on tiny code fragments.

To optimize programs for performance, prior works also introduced cache simulators, such as Dinero IV [48] and Valgrind's Cachegrind [49]. These cache simulators are used to instrument target applications to detect performance bottlenecks. However, they focus on hit/miss behaviour instead of the cache content. Such cache simulators have also been developed for benchmarking new cache designs [50], [51]. However, these works are limited to the cache and do not focus on the content that we require in our approach.

Execution of Transient Instructions. Several existing tools explicitly execute transient code paths to simulate or model the effects of speculative execution. For instance, SpecFuzz [52] inverts the direction of direct branches in a target program, forcing the architectural execution of code paths that a program would otherwise only take speculatively. This enables, e.g., the use of fuzzers to detect speculative out-of-bounds accesses. Tools like Scam-V [53], [54] and Revizor [29], [55] search for violations in a CPU's expected speculation behavior, which is specified, e.g., by a leakage contract [56]. Scam-V symbolically executes transient paths for guiding the input generation of test cases, while Revizor may execute transient paths in an ISA simulator to obtain contract traces. Finally, approaches like Spectector [57], KLEESpectre [58], and Haunted RelSE [59] leverage symbolic execution of transient paths to detect speculative leakage.

6.2. Limitations

While Crucible offers a practical and accessible way to emulate transient execution vulnerabilities, it inevitably introduces trade-offs that limit its scope.

Scope. Crucible, by design, is constrained to user-space applications. Though ideal for prototyping, it cannot simulate privileged transient execution. Furthermore, it restricts the types of victim programs we can instrument for the simulation of MDS and Downfall. For instance, our implementation does not support victim programs in kernel mode or SGX enclaves. We argue that given a user-mode harness for kernel or enclave-based victim code, Crucible could be used to prototype attacks against such targets. Though implemented for x86, Crucible's ISA-agnostic design enables ARMv8/RISC-V ports. To date, only Meltdown has been shown on some ARMv8 processors [1].

Crucible targets Meltdown-type attacks because faults provide a precise architectural trigger for initiating shadow stream execution. Such a trigger is not available for branch speculation, and as a result, Spectre-type attacks are out of scope for Crucible. For the same reason, Crucible only targets attacker programs that use fault handlers and TSX for fault suppression, excluding programs that suppress faults by speculation [1]. However, our approach could be extended to programs where a trigger for branches is retrofitted, e.g., via binary instrumentation or OS-controlled hardware breakpoints. We leave this for future research, as open challenges, such as simulating the behavior of branch predictors, make this endeavor non-trivial.

Performance. Performance is dominated by process creation and context setup. Three drop-in variants can reduce cost. First, a lightweight fork path using `clone3` with `CLONE_VM|CLONE_THREAD` and a private stack plus `userfaultfd`-based rollback avoids page-table duplication and reduces page faults. Second, a per-core child pool with ring-buffer handoff removes creation cost entirely. Third, a KVM-based hypervisor variant could reduce overhead by avoiding libc and scheduler paths. We prioritized deployment simplicity over implementation, though neither approach requires modifying attacker code or syscall ABIs.

Cache Effects. Most exploits and PoC implementations for Meltdown-type vulnerabilities utilize Flush+Reload to probe the CPU cache. We extensively demonstrate that Crucible creates a microarchitectural state in which this side channel yields the same results as with a real-world vulnerability. However, side channels like Prime+Probe may yield inaccurate results under Crucible, as its fault handlers evict additional cache lines. Note that for reproducing Meltdown-type attacks, this is typically not an issue. We are unaware of any existing Meltdown-type attack that utilizes Prime+Probe, as other side channels are usually more practical.

Loop Handling. Static patching is insufficient when the shadow stream enters loops or when the transient window aborts early. However, this is purely an engineering problem that can be solved. The idea is to use an performance-counter-based terminator: the child arms a per-core fixed counter on `INST_RETIRED.ANY` with a sample period equal to the configured reorder buffer capacity. The corresponding performance monitoring interrupt delivers a signal that our handler can convert into a controlled trap. This bounds the transient window even inside loops.

Mitigations. While Crucible emulates mitigations like masking of transiently used values (see Section 5.3.2), we currently do not implement other OS or microcode-level mitigations. However, Crucible could be extended to support a large variety of mitigations, such as the flushing of microarchitectural buffers with the `verw` instruction or clearing the L1d on context switch [4].

7. Conclusion

We presented Crucible, a software-only framework that enables simulation of Meltdown-type transient execution vulnerabilities on unmodified x86 systems. Crucible reproduces key microarchitectural behaviors, including cache effects, transient windows, and fencing, by duplicating instruction streams across isolated processes, allowing unmodified proof-of-concept exploits to execute with realistic outcomes. We demonstrated its effectiveness by reproducing real attacks, including VeraCrypt key extraction, with results that closely align with real hardware. Crucible supports binary-only targets, integrates seamlessly with modern fuzzers, and enables detection of both known and synthetic vulnerabilities. By making research on Meltdown-type vulnerabilities more accessible and repeatable, Crucible bridges a critical gap in CPU security experimentation, legacy exploit preservation, and hands-on education. We hope the framework accelerates progress on both offensive insight and robust mitigation, ultimately resulting in more secure CPUs.

Acknowledgements

We would like to thank our anonymous reviewers for their strong support and insightful feedback. Additionally, we would like to thank Joshua Heinemann, Fabian Thomas, Daniel Weber and Ruiyi Zhang for constructive discussions contributing to the success of this work.

Ethics Considerations

Our work retrofits *already–public* transient–execution vulnerabilities onto otherwise unaffected x86 CPUs. No new hardware bugs were discovered, and no proprietary data or human subjects were involved, so formal IRB approval was not required. We believe the benefits of a reproducible, sharable testbed for transient-execution research outweigh the limited residual risks. Nevertheless, we reflected on three areas of potential risk:

Dual Use. Crucible could, in principle, help adversaries practise or refine attacks. We judged this risk to be low because the framework only reproduces vulnerabilities that have been public for years (e.g., Meltdown, RIDL, Downfall), all of which have mature mitigations.

Impact on Vendor Ecosystems. Because Crucible does not expose new vulnerability classes, we determined that coordinated disclosure was unnecessary.

Misinterpretation of Synthetic Bugs. The two artificial vulnerabilities (vector-load leakage and divisor timing) are clearly labelled as hypothetical in code and documentation, so they cannot be mistaken for real CVEs. They are intended solely for fuzzer benchmarking and are off by default.

LLM Usage Considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security*, 2018.

[2] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security*, 2018.

[3] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *S&P*, 2019.

[4] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[5] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *CCS*. ACM, 2019.

[6] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis," in *USENIX Security Symposium*, 2020.

[7] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *USENIX Security*, 2023.

[8] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture," in *USENIX Security*, 2022.

[9] T. Ormandy, "Reptar," 2023. [Online]. Available: https://lock.cmpxchg8b.com/reptar.html

[10] ——, "Zenbleed," 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

[11] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *USENIX Security*, 2024.

[12] P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, and M. Schwarz, "CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode," in *WOOT*, 2023.

[13] D. Gruss, D. Hansen, and B. Gregg, "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer," *USENIX ;login*, 2018.

[14] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *S&P*, 2020.

[15] B. Swierzy, M. Hoffmann, F. Boes, F. Betke, L. Hein, M. Shevchishin, J.-N. Sohn, and M. Meier, "Teem: A cpu emulator for teaching transient execution attacks," in *Sicherheit 2024*. Gesellschaft für Informatik eV, 2024.

[16] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security*, 2019, extended classification tree and PoCs at https://transient.fail/.

[17] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *AsiaCCS*, 2020.

[18] D. Gruss, M. Schwarz, and M. Lipp, "Meltdown: Basics, details, consequences," in *BlackHat USA*, 2018.

[19] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," in *USENIX Security*, 2022.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[21] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *USENIX Security*, 2023.

[22] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[23] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *USENIX Security*, 2021.

[24] D. Weber, F. Thomas, L. Gerlach, R. Zhang, and M. Schwarz, "Reviving Meltdown 3a," in *ESORICS*, 2023.

[25] Intel, "Intel 64 and IA-32 Architectures Software Developer′s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," 2024.

[26] ——, "Deep Dive: Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort," November 2019. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort

[27] ——, "Performance Monitoring Impact of Intel Transactional Synchronization Extension Memory Ordering Issue," 2023, Document Number 604224.

[28] Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities," in *NDSS*, 2020.

[29] J. Hofmann, E. Vannacci, C. Fournet, B. Köpf, and O. Oleksenko, "Speculation at fault: Modeling and testing microarchitectural leakage of CPU exceptions," in *USENIX Security*, 2023.

[30] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, 2014.

[31] "Seccomp BPF (SECure COMPuting with filters)," 2025, accessed 2025-08-04. [Online]. Available: https://www.kernel.org/doc/html/v6.8/userspace-api/seccomp_filter.html

[32] Intel Corporation, "Pin - A Dynamic Binary Instrumentation Tool." [Online]. Available: https://www.intel.com/software/pintool

[33] A. Chakraborty, N. Mishra, and D. Mukhopadhyay, "Shesha: Multi-head microarchitectural leakage discovery in new-generation intel processors," *arXiv preprint*, 2024.

[34] D. Weber, L. Niemann, L. Gerlach, J. Reineke, and M. Schwarz, "No Leakage Without State Change: Repurposing Configurable CPU Exceptions to Prevent Microarchitectural Attacks," in *ACSAC*, 2024.

[35] IDRIX, "VeraCrypt," 2018. [Online]. Available: https://veracrypt.fr

[36] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *CCS*, 2016.

[37] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels," in *USENIX Security*, 2023.

[38] S. P. E. Corporation, "SPEC CPU 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[39] F. Thomas, L. Gerlach, and M. Schwarz, "Hammulator: Simulate Now – Exploit Later," in *DRAMSec*, 2023.

[40] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *USENIX Security Symposium*, 2019.

[41] A.-R. Sadeghi, J. Rajendran, and R. Kande, "Organizing the world's largest hardware security competition: challenges, opportunities, and lessons learned," in *Great Lakes Symposium on VLSI*, 2021.

[42] M. Bölcskei, F. Solt, K. Ceesay-Seitz, and K. Razavi, "Encarsia: Evaluating cpu fuzzers via automatic bug injection," in *USENIX Security*, 2025.

[43] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Usenix ATEC*, 2005.

[44] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," *BlackHat USA*, 2015.

[45] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *USENIX Security*, 2022.

[46] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, 2011.

[47] A. Abel and J. Reineke, "uiCA: Accurate throughput prediction of basic blocks on recent intel microarchitectures," in *ACM International Conference on Supercomputing*, 2022.

[48] J. Edler, "Dinero iv: Trace-driven uniprocessor cache simulator," 1994. [Online]. Available: http://www.cs.wisc.edu/\~markhill/DineroIV

[49] N. Nethercote, "Dynamic binary analysis and instrumentation," 2004.

[50] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *MICRO*, 2010.

[51] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security Symposium*, 2019.

[52] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing spectre-type vulnerabilities to the surface," in *USENIX Security*, 2020.

[53] H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, "Validation of abstract side-channel models for computer architectures," in *CAV*, 2020.

[54] P. Buiras, H. Nemati, A. Lindner, and R. Guanciale, "Validation of side-channel models via observation refinement," in *MICRO*, 2021.

[55] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box cpus against speculation contracts," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[56] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *S&P*, 2021.

[57] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled Detection of Speculative Information Flows," in *S&P*, 2020.

[58] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution," *TOSEM*, 2020.

[59] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse," in *NDSS*, 2021.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper introduces Crucible, a software-based framework that enables reproducible simulation of Meltdown-type transient execution vulnerabilities in x86 CPUs that are not actually vulnerable. Crucible employs a novel shadow-stream mechanism to duplicate execution after a fault, injecting microarchitectural side-effects that mimic the artifacts of transient execution. The authors prototyped several known and artificial vulnerabilities on Crucible and demonstrated the fidelity of reproducing intended leakage behavior and attack outcomes.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Addresses a Long-Known Issue.
- Provides a Valuable Step Forward in an Established Field.

A.3. Reasons for Acceptance

1) The paper creates a new tool to enable future science. The authors introduce a new framework that enables modeling future or emerging vulnerabilities and studying fuzzer behavior, facilitating more principled evaluation and vulnerability discovery.
2) The paper addresses a long-known issue. The authors introduce a framework that decouples vulnerability research from specific CPU models to address the challenges stemming from the lack of real hardware platforms that remain vulnerable to transient-execution leakage.
3) The paper provides a valuable step forward in an established field. The authors introduce a new framework that reproduces transient-execution behaviors, lowering the barriers for experimentation, benchmarking, and education. The proposed system enables systematic evaluation of exploits, detection tools, and fuzzers under controlled conditions.