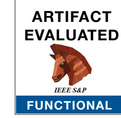


TREVEX: A Black-Box Detection Framework For Data-Flow Transient Execution Vulnerabilities

Daniel Weber*, Fabian Thomas*, Leon Trampert*, Ruiyi Zhang*, Michael Schwarz*
*CISPA Helmholtz Center for Information Security
Email: {daniel.weber, fabian.thomas, leon.trampert, ruiyi.zhang, michael.schwarz}@cispa.de

Abstract—**Transient execution attacks continue to pose serious security risks, even years after their initial discovery in the form of Meltdown and Spectre. Despite growing awareness and research, most attacks have been discovered manually. Existing automated approaches only focus on variants of known attacks or make strict assumptions, such as access to the CPU's RTL, a given leakage contract, or an ISA emulator. These limitations hinder broader and more generic detection, especially in post-silicon environments.**

In this paper, we present TREVEX, a post-silicon black-box detection framework designed to discover data-flow transient execution vulnerabilities automatically. TREVEX does not rely on RTL access or semantics of the ISA. Instead, TREVEX detects any unexpected transient data flow between different execution contexts by employing novel techniques informed by insights from prior transient execution attacks. We evaluate TREVEX across 20 microarchitectures from Intel, AMD, and Zhaoxin. Hereby, TREVEX discovers Floating Point Divider State Sampling (FP-DSS), a novel transient execution attack affecting AMD CPUs. We show that FP-DSS allows an unprivileged attacker and even a malicious website to leak data from different security domains, including the operating system. TREVEX further discovers a new variant of FPVI on AMD CPUs and that Zhaoxin CPUs are affected by FPVI. Our study shows that TREVEX reliably discovers known vulnerabilities on affected machines. Our study shows that TREVEX detects known vulnerabilities on affected CPUs, while also closing gaps in existing vendor documentation. Our findings highlight the urgent need for more diverse automated tools and demonstrate that TREVEX fills an important gap in current CPU vulnerability research.

1. Introduction

Despite being known for more than half a decade, transient execution attacks still pose a major threat to modern computer systems due to their severe security implications [1], [2], [3]. Shortly after the initial discovery of Meltdown [1] and Spectre [2] in 2018, it became apparent that these vulnerabilities were not isolated occurrences, but the first instances of a new class of vulnerabilities exploitable via so-called transient execution attacks (TEAs) [4], [5], [6], [7], [8], [9], [10]. Over the following years, researchers discovered a plethora of similar TEAs, such as ZombieLoad [5], RIDL [4], Fallout [6], GDS [9], and new Spectre variants [11], [12]. More recently, it was shown that the introduction of new CPU optimizations, such as load address or load value prediction, further increases the attack surface of CPUs [13], [14], consequently raising the need for automated vulnerability discovery.

Despite significant efforts by the research community to automate the discovery of transient execution vulnerabilities (TEVs), the vast majority were found manually. Automated approaches are typically constrained by their assumptions. As an example, a technique commonly employed for finding vulnerabilities is to mutate known vulnerabilities [15], [16]. While generating variants of known vulnerabilities, and thus projecting current assumptions onto future vulnerabilities, has yielded results in the past, it also significantly restricts the scope of findings [12], [15]. More formal approaches rely on the modelling of a leakage contract, including all known but excluding all unknown leakages, or on a correct emulator of the CPU's instruction set architecture (ISA) [17], [18], [19], [20]. Hence, these approaches require precise modelling of known leakage behaviors and base their scope on the correctness and completeness of the emulator. However, even state-of-the-art ISA emulators are often incomplete and may therefore not be able to correctly emulate all instructions of a CPU [17]. In addition, undocumented instructions, which are commonly found in commercial CPUs [21], [22], are naturally not supported by such emulators, which further limits the scope of these approaches. Alternative approaches that instead rely on access to a CPU's register-transfer level (RTL) [12], [23], [24], [25], [26], [27], [28], [29], [30] cannot be used to conduct independent research on most commercial CPUs, as their RTLs are not public. These limitations call for an *automated* but *generic* approach that treats the CPU and its ISA as a *black box* and is capable of finding transient execution vulnerabilities in commercial *post-silicon* designs, enabling applicability to a wide range of CPUs. Hence, we raise the following research question: *Can we develop a black-box vulnerability detection approach for discovering generic transient execution vulnerabilities on commercial CPUs?*

In this paper, we propose TREVEX (TRansient Execution Vulnerability EXplorer), a black-box detection framework

for data-flow transient execution vulnerabilities[1]. TREVEX does not assume knowledge of the semantics of the code it emits and minimizes the assumptions on what a vulnerability looks like. While our PoC implementation targets training-free data-flow transient execution vulnerabilities, TREVEX's design is capable of finding *generic* data-flow transient execution vulnerabilities, e.g., vulnerabilities that do not arise from the memory subsystem. TREVEX achieves this without relying on a leakage contract, the correctness of an ISA emulator, or access to the CPU's RTL. We focus our research on the discovery of data-flow transient execution vulnerabilities (mostly Meltdown-type attacks), as they typically reassemble the behavior of classic software bugs, i.e., undocumented behavior that gets fixed by a CPU vendor once discovered [3], [31]. By contrast, control-flow transient execution vulnerabilities (mostly Spectre-type attacks) typically misuse well-understood CPU features [2], [3], [32], [33], [34] rather than bugs. To find generic data-flow transient execution vulnerabilities, TREVEX combines 3 key innovations. First, TREVEX uses a novel *data-flow divergence testing algorithm* that is capable of detecting unexpected transient data flows, instead of only focusing on the mere existence of transient windows [35] or concrete leakage [15]. Second, TREVEX incorporates a technique we refer to as *instruction shadowing* to taint *arbitrary* microarchitectural elements, overcoming the previous limitation on the memory subsystem [15]. Third, TREVEX leverages a *cross-privilege domain data-dependency detection* algorithm for inferring whether the attack's leakage actually stems from the victim's privilege domain.

Using TREVEX, we conduct a CPU fuzzing campaign using 20 microarchitectures across x86 CPUs from Intel, AMD, and Zhaoxin. In the study, we find that TREVEX is effective at discovering novel vulnerabilities and that it reliably rediscovers known vulnerabilities, demonstrating its applicability to regression tests. Most prominently, TREVEX discovers Floating Point Divider State Sampling (FP-DSS), a novel TEA impacting AMD's Zen and Zen+ microarchitectures. FP-DSS allows an attacker to leak data across privilege boundaries via AMD's floating point division unit. To demonstrate the security impact of FP-DSS, we showcase that it allows an attacker to leak data from the Linux kernel. We further craft a proof-of-concept JavaScript exploit running in Chrome to leak data from a native process running on the same machine, thus demonstrating the large-scale security impact of FP-DSS. TREVEX discovers a new variant of Floating Point Value Injection (FPVI) on AMD CPUs that does not require denormal input values to trigger. TREVEX further discovers 3 instances of Zero-at-ret, a lesser-known variant of LVI-NULL. While Intel was aware of the existence of Zero-at-ret [36], the list of affected CPUs is not publicly documented [31], [36]. Furthermore, our disclosure led to Intel internally investigating and reconsidering their initial assessment of the vulnerability's impact. TREVEX shows that Zhaoxin CPUs are vulnerable to FPVI, a vulnerability previously only known to affect Intel and AMD CPUs. By discovering Gather Data Sampling (GDS) [9] and Floating Point Value Injection (FPVI) [8], [37], [38], TREVEX becomes the first automated approach to detect these vulnerabilities. Additionally, TREVEX successfully rediscovers all TEVs that we expect to find. We conclude that TREVEX is effective in discovering novel vulnerabilities across varying CPU designs and in discovering known TEVs for regression testing. Thus, TREVEX fills a crucial gap in the current state of CPU vulnerability research.

To summarize, we make the following contributions:

1) We design and implement TREVEX—a post-silicon black-box transient execution vulnerability detection framework that is capable of discovering generic data-flow transient execution vulnerabilities without reasoning about an ISA's semantics or requiring RTL access. Using TREVEX, we conduct a CPU fuzzing campaign, testing 20 x86 microarchitectures from Intel, AMD, and Zhaoxin.
2) We discover a novel TEA—Floating Point Divider State Sampling (FP-DSS)—affecting AMD CPUs. We demonstrate the exploitability of FP-DSS by leaking data from the Linux kernel and by crafting a Google Chrome exploit that leaks data from a native process.
3) Using TREVEX, we identify 3 instances of Zero-at-ret, a variant of LVI-NULL, for which the set of affected CPUs is undocumented. We discover a new variant of FPVI on AMD CPUs that does not require denormal input values to trigger. We further uncover that Zhaoxin CPUs are vulnerable to FPVI, a vulnerability previously only known to affect Intel and AMD CPUs.
4) TREVEX rediscovers all known data-flow transient execution attacks that we expected to find, including GDS [9] and FPVI [8], which previous tools did not support.

Availability. Our code and experiments are available at https://github.com/cispa/trevex.

Responsible Disclosure. We responsibly disclosed our findings to all affected CPU vendors and are currently engaged in ongoing discussions. We agreed with AMD on an embargo period until March 16, 2026, for FP-DSS and the new FPVI variant. After an embargo extension request by AMD, the embargo was extended until April 17, 2026. AMD informed us that FP-DSS will be assigned CVE-2025-54505 and will be discussed in a security notice. AMD further informed us that the FPVI variant will be discussed in an upcoming security brief.

Regarding our Zero-at-ret disclosure, Intel informed us about internal investigations to reconsider the vulnerability's impact.

2. Preliminaries

In this section, we provide the background required to understand the remainder of this paper, as well as the terminology used. We describe how covert channels can recover information from the transient domain, which enables reasoning about transient data from within the archi-

1. For a precise definition of *data-flow transient execution vulnerabilities*, we refer the reader to Section 2.

tectural domain. We introduce transient execution attacks and vulnerabilities along with a taxonomy that splits them into control-flow and data-flow transient execution vulnerabilities. We give a brief overview of CPU fuzzing and how it differs from traditional software testing.

2.1. Transient State Recovery via Covert Channels

Modern CPUs employ out-of-order execution—a feature that enables them to execute instructions without ever committing the result of their execution to the architectural state, thus enabling optimizations such as speculative execution. These instructions are commonly referred to as transiently executed instructions [3]. Although they do not influence the architectural domain, transiently executed instructions can influence microarchitectural structures, e.g., CPU caches [1], [2] or the states of execution units [39]. A *covert channel* translates such microarchitectural effects back into architecturally-observable behavior [1], [2], [3]. A commonly-used covert channel Flush+Reload [1], [2], [40], where the attacker (i) flushes a cache line, (ii) lets the transient instructions conditionally access an address mapped to that cache line, and (iii) measures the reload latency. A fast load time implies the cache line was brought back to the cache by a transiently executed load, thus leaking one bit of information. When a lookup table is used, multiple bits can be encoded at once. For example, a 256-byte lookup table allows encoding one byte of information per transient access by encoding the transiently accessed data as index in the lookup table, e.g., accessing the i-th entry of the lookup table to encode the value i [1], [2]. Other transmission primitives, e.g., Prime+Probe [41] on set-associative caches or contention on execution ports [42], follow the same 3-step pattern: prepare a microarchitectural state, let transient instructions encode a secret, and measure a property to recover the secret [43].

2.2. Transient Execution Attacks

A *transient execution vulnerability (TEV)* is a CPU vulnerability that affects the control or data flow of the ISA instruction stream during the transient execution of instructions, without leaving any directly observable effects in the architectural domain. We refer to code that combines a TEV with further building blocks, e.g., a covert channel, to leak or inject otherwise inaccessible information as a *transient execution attack (TEA)*. Transient execution attacks (TEA) abuse the mismatch between microarchitectural behavior and architectural guarantees. The goal of a TEA is typically one of the following. Either an attacker wants to manipulate a victim application into transiently operating on incorrect code or data, thus altering its behavior [2], [7], [8], [33], [34], or the attacker wants to leak otherwise inaccessible data from another privilege domain, such as the kernel or another process [1], [4], [5], [6], [9], [44], [45]. Hereby, the actual data exfiltration from the transient domain is done via a covert channel as described in Section 2.1.

TEAs are commonly grouped into two broad classes [3], i.e., Spectre-type and Meltdown-type attacks. However, this definition does not cover all known variants of TEAs and TEVs. Thus, to clearly define the scope of our work, we introduce a taxonomy spanning the entirety of currently known TEAs.

Control-Flow Transient Execution Vulnerabilities (CF-TEVs) manipulate the transient *control flow* of an instruction stream, e.g., by manipulating branch prediction mechanisms so the processor executes instructions along a misspeculated control path. Thus, CF-TEVs are instances of attackers misusing well-understood CPU features [2], [32], [33], [34]. For example, a mispredicted branch (Spectre-PHT), poisoned indirect (Spectre-BTB) or return target (Spectre-RSB) allows the attacker to redirect *control flow* transiently.

Data-Flow Transient Execution Vulnerabilities (DF-TEVs) exploit changes in the transient *data flow*, e.g., incorrectly forwarded results, to either leak otherwise inaccessible data or transiently inject data into another process. We refer to these subclasses as *data-leakage* and *value-injection* TEVs, respectively. Examples for data-leakage DF-TEVs include Meltdown-US [1] and Microarchitectural Data Sampling (MDS) attacks [4], [5], [44], [45]. Value-injection DF-TEVs include Load Value Injection (LVI) [7], [46] and Floating Point Value Injection (FPVI) [8], [37], [38]. In contrast to CF-TEVs, the majority of DF-TEVs arise from undocumented and unintended CPU behavior, such as lazy exception handling, and thus are akin to bugs in a CPU's design rather than a feature, making them an ideal target for post-silicon testing. Note that DF-TEVs can also stem from data-flow predictors, e.g., Speculative Store Bypass (SSB) [47] or Predictive Store Forwarding (PSF) [48]. Recent research has shown that especially the Apple M-Series CPUs make use of several data-flow predictors [13], [14], [49].

We refer to attacks abusing DF-TEVs and CF-TEVs as *data-flow transient execution attacks (DF-TEAs)* and *control-flow transient execution attacks (CF-TEAs)*, respectively. Furthermore, we introduce the terms *training-based* and *training-free* TEVs and TEAs to refer to variants that require previous mistraining or variants that do not require any mistraining for reliable behavior, respectively. For example, Spectre-PHT and Spectre-BTB are instances of training-based CF-TEVs, whereas Spectre-RSB is a training-free CF-TEV. Furthermore, Meltdown-US and MDS are instances of training-free DF-TEVs, whereas PSF is a training-based DF-TEV.

We choose this classification for a clear distinction between existing attacks based on their exploitable behavior, which sets the boundaries for an attacker, while being observable for researchers discovering new vulnerabilities. Furthermore, it removes uncertainties about the classification of existing attacks. While our proposed classification is similar to the classification of Van Schaik et al. [4], their classification only discusses data-leakage TEVs, e.g., Meltdown and MDS, but it does not include value-injection TEVs, e.g., LVI or FPVI. Moreover, their classification does not specify on which abstraction level the distinction

between control- and data-flow transient behavior is made, as opposed to our explicit focus on the ISA abstraction level.

2.3. CPU Fuzzing

Fuzzing (or *fuzz testing*) refers to an automated test generation technique by generating random inputs for a program [50]. Note that while white-box fuzzers can use feedback loops to guide input generation [51], [52], black-box fuzzers can generate inputs randomly without any feedback from the target program [50]. In contrast to traditional software fuzzing [50], [53], *CPU fuzzing* mutates instruction sequences rather than data inputs and evaluates them on either a simulated or physical CPU.

CPU fuzzers can be split into *pre-silicon* or *post-silicon* types, based on whether they operate on register-transfer level (RTL) descriptions of a CPU or a production-ready and already fabricated CPU, respectively. *Pre-silicon* fuzzers benefit from more insight into the hardware and are capable of finding bugs before the CPU rollout. However, in practice, they are limited in their performance [54]. Furthermore, the designs of modern commercial CPUs are rarely publicly accessible, which hinders independent research on these CPUs. In contrast, *post-silicon* fuzzing is significantly faster [54] without requiring access to a manufactured CPU, which enables testing by independent researchers. However, several constraints make fuzzing CPUs post-silicon more difficult than traditional software fuzzing:

1) *Oracle construction*. The analogue of an "indicator of misbehavior" is subtle: a transient leak, deadlock, or performance counter anomaly.
2) *State explosion*. Each instruction interacts with multiple hidden and potentially unknown buffers and predictors. As exhaustive enumeration is infeasible, practical fuzzers compose known building blocks (e.g., faulting loads, fences, transactions) and rely on randomness to explore the remaining space [15], [18].
3) *Limited feedback*. Unlike instrumented binaries, a commodity CPU provides, at best, coarse coverage metrics (performance counters) and, at worst, only success/failure signals.

While multiple post-silicon CPU fuzzers have been proposed, they show severe limitations, such as relying on mutating known attack variants [15], [55], requiring a correct ISA emulator or a given leakage contract [17], [18], [19], or only finding transient windows without reasoning about their attack potential [35], [56].

3. Research Scope and Threat Model

In this section, we define the scope of our work based on the taxonomy introduced in Section 2. We further discuss the underlying threat model of attacks that we aim to discover.

3.1. Research Scope

Following the taxonomy of TEVs introduced in Section 2, it becomes evident that most TEAs discovered in recent years are, in fact, DF-TEAs, e.g., MDS [4], [5], [44], LVI [7], [46], FPVI [8], [37], [38], LVP [13], LAP [14], Spectre-BB [49], and GDS [9]. We hypothesize that this results from the fact that most known CF-TEAs exploit branch prediction. These prediction mechanisms are typically documented by the corresponding CPU vendor and thus well-understood, in contrast to the less-understood behaviors exploited by DF-TEAs. Consequently, most CF-TEAs were discovered shortly after the initial discovery of TEAs [2], [33], [34]. Furthermore, recent Apple M-series CPUs make strong use of data-flow predictors [13], [14], [49].

In this paper, we focus on the discovery of training-free data-flow transient execution vulnerabilities (training-free DF-TEVs). Note that while our proposed design (cf. Section 4) is unaffected by the training property and hence capable of discovering training-based DF-TEVs as well. However, developing code generation strategies to emit relevant training sequences is a challenging task that we leave as future work. The reason for our focus is twofold. First, DF-TEVs typically reassemble the behavior of classic software bugs, i.e., undocumented behavior that gets fixed by a CPU vendor once discovered [3], [31]. Thus, making it hard to enumerate potentially vulnerable behavior of a CPU, in contrast to CF-TEVs that typically build on top of well-understood CPU features [2], [3], [20], [32], [33], [34]. Second, DF-TEAs account for the majority of transient execution attacks [3], especially in recent years [8], [9], [13], [14], [18], [49]. In other words, we search for vulnerabilities that allow an attacker to leak sensitive information from another privilege domain, e.g., another privilege level or process, or maliciously inject incorrect data into a victim's computation. Note that for the latter, the victim can also take the form of the surrounding environment [8], [49].

3.2. Threat Model

If not explicitly stated otherwise, we assume the following threat model. The attacker can execute arbitrary native code on the target system. We assume the absence of classical software vulnerabilities, e.g., memory corruption vulnerabilities. We further assume that the attacker cannot mount traditional software-based side-channel attacks, e.g., cache attacks, against the victim. Note that this still allows the attacker to use microarchitectural covert channels in their own privilege domain to recover transiently leaked data. For attacks that require page-table modifications, we assume that the attacker can modify the bits either indirectly, e.g., by setting the Dirty bit when writing to a memory page, or directly via accessing privileged interfaces. Note that the latter is realistic for attacks against trusted execution environments, such as Intel SGX, where the attacker controls the page-table entries of the enclave.

4. TREVEX: Design

In this section, we answer our research question by proposing the design of a transient execution vulnerability

detection framework that does not require any knowledge about the semantics of the emitted code, a model of the CPU's microarchitecture, or access to the CPU's RTL. Furthermore, we ensure that the design is generic to prevent restricting it to a subset of DF-TEVs. We refer to this framework as TREVEX (TRansient Execution Vulnerability EXplorer). TREVEX's design is based on 3 key innovations. In the following, we give a high-level overview of TREVEX and describe each technique in detail. We discuss the implementation in Section 5.

4.1. Design Overview

At a high level, TREVEX operates in 3 stages: test case generation, execution, and classification. Figure 1 illustrates the workflow of TREVEX.

In the first stage, TREVEX generates a *test case* consisting of code for the attacker context, code for the victim context, and parameters for the environment. Hereby, TREVEX applies *instruction shadowing* ($\mathcal{T}1$) to taint *arbitrary* microarchitectural buffers, in contrast to a fixed set of buffers as in previous work [15]. The high-level idea is to create two contexts, one for a for a potential attacker and one for a potential victim, and let them use similar instructions, thus increasing the likelihood of influencing the same microarchitectural buffers.

In the second stage, TREVEX sets up the environment, initializes the attacker and victim contexts, and executes their code sequences in parallel. To model realistic attack scenarios and to enable reasoning about cross-privilege data flow, TREVEX executes the attacker and victim contexts in different privilege domains. For this, TREVEX exposes an interface that can be implemented by a diverse set of victim contexts. These victim contexts can, for example, be instantiated as another process (to model cross-process leakage), a kernel module (to model kernel to user leakage), or a trusted execution enclave (to model leakage from a trusted execution environment). This allows TREVEX to model realistic TEA scenarios and thus trigger TEVs that are easier to observe in the presence of an actual victim context. Furthermore, it enables TREVEX to reason about data flow between different privilege domains as required by data-leakage TEVs [1], [4], [5], [9], [18], [44], [45].

The attacker context contains code with instructions whose transient computations are converted into architectural state using a covert channel (cf. Section 2.1). While we leave the details of how transient computations are extracted open to the implementation, our implementation achieves this by following a template-based code generation approach that ends in the result of a specific instruction being encoded using a covert channel (cf. Section 5.1). These recovered transient values, together with the number of occurences for each value, are stored. This information is then used in TREVEX's leakage detection as the input for a *data-flow divergence testing algorithm* ($\mathcal{T}2$). The purpose of this algorithm is to detect *any* transient data flow that diverges from the architectural data flow, by comparing the transient data against expected architectural behavior. Using this technique, TREVEX can detect *any* transient data flow that deviates from the architectural behavior, without requiring knowledge about the semantics of the emitted code.

After identifying unexpected transient data flows, TREVEX reasons about whether the observed leakage depends on data originating from the victim context. To this end, TREVEX employs a *cross-privilege data-dependency detection* algorithm ($\mathcal{T}3$). This technique allows splitting test cases with observed leakage into data-leakage TEVs, when cross-privilege data flow is detected, and value-injection TEVs, when no such dependency is found.

In the third stage, the classification stage, TREVEX filters out false positives using a set of heuristics and further classifies the detected leakage based on its characteristics. This stage employs heuristics to determine whether the test case rediscovers known TEVs or whether it exhibits novel behavior. This classification eases manual inspection of the results.

In the following, we describe each of the 3 key techniques in detail.

4.2. $\mathcal{T}1$: Microarchitectural Tainting via Instruction Shadowing

Instructions that transiently return architecturally inaccessible data typically source that data from microarchitectural elements, e.g., the CPU cache [1], the line-fill buffer [4], [5], or the staging buffer [45]. The previous work Transynther [15] fills *known* buffers with pre-defined values using *manually crafted* routines to attribute leakage to microarchitectural elements. For the remainder of this paper, we refer to these marker values as *taints* and to the process of storing such taints in a microarchitecutral element as *tainting*. If a microarchitectural element is not tainted by these pre-defined routines, Transynther fails to detect leakage stemming from the element.

To solve this issue, we introduce a technique called *instruction shadowing*. The intuition behind it is that when an instruction can leak data from a specific microarchitectural buffer, the same instruction is likely also capable of influencing the content of that buffer. For example, memory loads that leak data from the line-fill buffer can also be used to taint the line-fill buffer [5]. Similar behavior has been observed with other TEAs where details about the responsible microarchitectural element are not publicly known [9], [18]. As an example, we experimentally verify that GDS [9] leakage is more stable, and thus easier to detect, with another process tainting the exploited microarchitectural state using the same vector instructions. Thus, TREVEX emits victim code that uses the *similar* instructions as the attacker code, which we refer to as *instruction shadowing*. By doing so, TREVEX also taints both *known and unknown* microarchitectural buffers that are shared between the attacker and victim contexts, allowing the discovery of previously unobservable TEVs (cf. Section 6.6). To pick a similar instruction, our implementation either picks the attacker code instruction whose transient result is encoded in a register or an instruction of the same category (cf. Section 5.1).

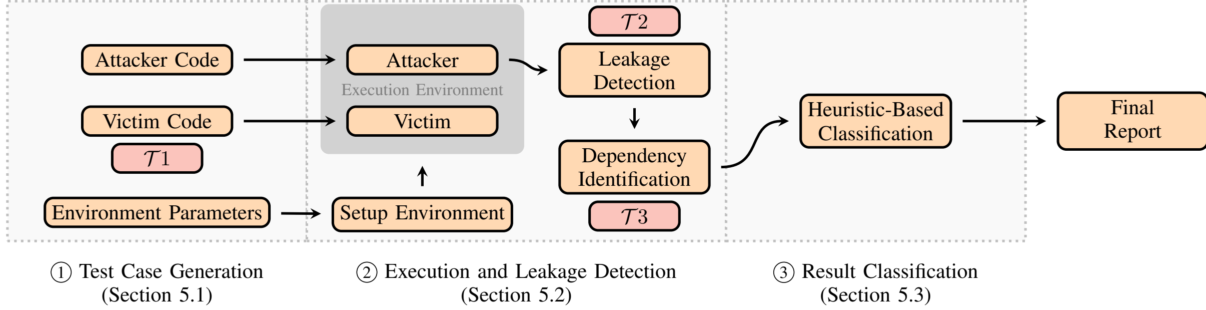
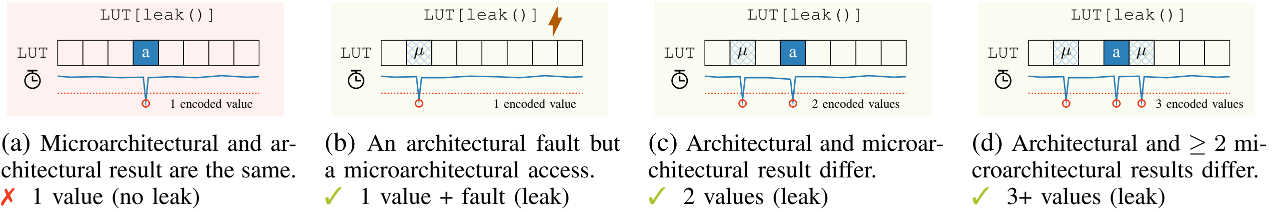


Figure 1: An overview of TREVEX's workflow and where the key techniques are applied. TREVEX generates an execution environment and code for the attacker and victim process ① using our instruction shadowing technique ($\mathcal{T}1$). Next ②, it executes the test case and detects whether the attacker can observe cross-privilege leakage via our data-dependency detection technique ($\mathcal{T}3$). Further, TREVEX uses our data-flow divergence testing algorithm ($\mathcal{T}2$) to accurately identify actual leakage. Such leakage is then classified in the last stage of TREVEX ③.



(a) Microarchitectural and architectural result are the same. ✗ 1 value (no leak)

(b) An architectural fault but a microarchitectural access. ✓ 1 value + fault (leak)

(c) Architectural and microarchitectural result differ. ✓ 2 values (leak)

(d) Architectural and ≥ 2 microarchitectural results differ. ✓ 3+ values (leak)

Figure 2: Illustration of TREVEX's data-flow divergence testing algorithm. If there is an architectural fault and still a microarchitectural access, or multiple different accesses, the architectural and microarchitectural values differ. Such an observation indicates unintended transient data flow and, consequently, a DF-TEV.

4.3. $\mathcal{T}2$: Data-Flow Divergence Testing

A crucial part of all fuzzers, including CPU fuzzers, is their ability to distinguish interesting test cases from uninteresting ones. Our detection is based on the following intuition: When the transient data flow, i.e., the microarchitectural state, diverges from the architectural data flow, i.e., the architectural state, there is unintended transient behavior that can potentially be exploited by an attacker. Note that data flow divergence alone can lead to exploitable behavior, even without direct data leakage, e.g., as demonstrated by LVI [7], [46], SCSB [8], [57], and FPVI [8], [37], [38]. Thus, TREVEX employs a data-flow divergence testing algorithm to detect whether the transient data flow of a test case diverges from the architectural data flow. This allows TREVEX to keep a negligible false positive rate while being able to detect arbitrary transient data flow divergence (cf. Section 6.2). Our data-flow divergence testing algorithm is illustrated in Figure 2. The algorithm operates on the list of values that were recovered using the covert channel in the execution of the attacker context. More precisely, the algorithm assumes that the i-th entry in the list (LUT) is marked if the value i was recovered from the transient domain during the execution of the attacker context for an arbitrary but fixed register. If more than one value was encoded, there were at least two different data flows leading to this encoding, as architecturally, a register can only contain one value at a time. Thus, for multiple recovered results, there is a transient execution path whose result differs from the architectural result, leading to transiently differing values in a register (cf. Figure 2c and Figure 2d). Two or more differing values always indicate at least one erroneous transient data flow, independent of whether the attacker process caught a fault or not. An example of such a case would be a microcode assist leading to MDS leakage [4], [5]. Moreover, if a fault has been raised and instructions depending on the result of the faulting instruction were able to execute and encode their result, this indicates unintended transient data flow (cf. Figure 2b). An example of such a case would be Meltdown-US [1].

This technique enables TREVEX to overcome key limitations of prior tools—namely, detecting transient windows without reasoning about exploitability [35], [56].

4.4. $\mathcal{T}3$: Cross-Privilege Domain Data-Dependency Detection

While our first technique allows for tainting arbitrary microarchitectural buffers, the following technique enables reasoning about potential cross-privilege leakage via these buffers. Since DF-TEAs typically target privilege boundaries [1], [4], [5], [9], [44], [45], TREVEX reasons about data dependencies between these domains to detect indirect data flows between them. For doing so, TREVEX follows the following *data-dependency detection algorithm*. First, TREVEX executes a given test case multiple times with varying taints

in the victim execution context collecting the respective leakage. To ensure stable results, TREVEX also repeats this step multiple times and aggregates the results. Next, TREVEX computes the symmetric difference of the leakages for each taint. Thus, the symmetric difference contains all leakage values only observed for a single taint. Thus, it represents the set of values that uniquely identify a given taint in the victim context. Eventually, TREVEX checks how many taints can be uniquely identified by such values. If the number of identifiable taints exceeds a configurable threshold, TREVEX flags the test case as having a data dependency between the execution contexts. Listing 1 in Appendix C shows the pseudo code of the data-dependency detection algorithm. This algorithm also ensures that TREVEX can detect TEVs that do not leak exact values, but all TEVs where there is a deterministic mapping between leakage and taint, i.e., the leakage depends on the victim's data. Such behavior is for example observed for MDS [4], [5], [44], [45], GDS [9], and Divider State Sampling (DSS) [18]. It is also observed for our newly discovered FP-DSS vulnerability (cf. Section 6.4). It is important to note that the algorithm in Appendix C checks for unique taints in the function `SymmetricDifference`. This is a conservative approach optimizing for a low false positive rate by sacrificing the ability to identify dependencies that only leak partial information about a taint. This conservative decision allows TREVEX to flag existing data dependencies with high confidence even for small sets of taints. To detect more fine-grained leakage, the function `SymmetricDifference` can easily be adopted to remove taints only when they exist in at least N other maps.

5. TREVEX: Implementation

In this section, we present TREVEX's implementation. TREVEX's implementation follows the design in Section 4 with the goal of discovering training-free DF-TEVs. In the following, we discuss each of its 3 stages—test case generation, execution, and classification—in detail.

5.1. Test Case Generation

In its first stage (1), TREVEX generates instruction sequences for the attacker and victim context, which we refer to as *attacker code* and *victim code*, respectively. Additionally, TREVEX generates the parameters for the runtime environment of the execution for each test case. The code generation step is essential, as it defines the scope of vulnerabilities that can be triggered. While all DF-TEAs share some common patterns, overfitting on these patterns can easily lead to missing vulnerabilities. Thus, TREVEX's code generation keeps the overall structure of a TEA while systematically mutating its components, similar to a grammar-based fuzzer. Note that TREVEX's current implementation does not target training-based DF-TEVs and hence does not aim to emit code containing training sequences.

In the following, we discuss in detail how TREVEX generates attacker and victim code and how it generates the environment parameters.

Attacker Code Generation. The attacker code consists of 3 parts: architectural state priming, exception generation, and cache encoding. Note that in the remainder of the paper, we use the term *exception* for any microarchitectural condition resulting in a pipeline flush, i.e., we do not restrict ourselves to actual faults. The first part of the code primes the architectural state. Registers are primed with valid memory pointers, values expected to trigger corner conditions, such as '0', '-1', varying floating point encodings, and random data. Memory pointers are randomly offset to potentially trigger unaligned memory accesses.

For the second part, TREVEX emits assembly instructions that are expected to trigger an exception, as exceptions are the most common way to trigger TEAs, e.g., through faulting memory loads or microcode assists [8]. Thus, the code is guided towards triggering exceptions, while still allowing the fuzzer to explore potentially vulnerable code patterns that do not trigger exceptions. The corpus of instructions that TREVEX emits is based on an initial list of instructions that can be generated independently. This is enabled by the fact that TREVEX does not assume *any* semantics about the instructions generated in this part of the attacker code. This makes TREVEX highly flexible and allows it to emit even undocumented instructions when provided with a list of suitable encodings. Such a list can be generated by automated tools that find undocumented instructions [21], [22].

After (potentially) triggering an exception, the third part of the generated code encodes the contents of a CPU register into the CPU cache using Flush+Reload [40] to transmit its content from the transient to the architectural domain. The code encodes a selected single byte of a selected single register for each iteration to keep the required transient window small, thus allowing TREVEX to find vulnerabilities restricted to limited-length transient windows. The selection of the exact byte and register is randomized as part of the attacker code generation.

Victim Code Generation. Besides generating code for the attacker context, TREVEX also generates code for a victim context. The purpose of the victim context is to taint the CPU's microarchitectural state from a foreign privilege domain to simulate a realistic attack environment. The victim code consists of a single tight loop that runs until stopped. Inside this loop, TREVEX picks one or several of the following operations: It either taints the CPU's memory subsystem with manually crafted routines, similar to prior work [15], or it performs our instruction shadowing technique (cf. Section 4.2). To pick similar instructions for the latter, TREVEX either uses the exact same instruction used in the second part of the attacker code, which we refer to as *strict similarity* or TREVEX picks instructions of the same category and ISA set, which we refer to as *lax similarity*. This metadata are retrieved from the initial instruction corpus.

Manually crafted routines using memory loads, stores, and flushes allow the efficient tainting of the memory subsystem [15]. In contrast, instruction shadowing allows for tainting arbitrary microarchitectural elements, including those that are shared with the attacker ($\mathcal{T}1$). For instruction shadowing, TREVEX emits the same instructions that are emitted for the exception generation part of the attacker code.

Environment Parameter Generation. In addition to the actual attacker and victim code, TREVEX also generates parameters for its execution environment. This involves mutating the memory permissions and configuration registers of the attacker code. For example, TREVEX may clear the User bit and set the Dirty bit in the page-table entry of the memory accessed by the attacker. Note that we do not tamper with the memory pages where the attacker code is stored, only with the memory pages potentially accessed by the attacker code. These mutations may trigger corner cases when fetching memory for the attacker code, such as specific microcode assists or faults. To trigger less common corner cases, TREVEX also mutates a CPU's configuration registers, such as the `MXCSR` register, which allows for unmasking faults for floating-point operations. For example, clearing bit 11 of `MXCSR` unmasks underflows in the floating point unit, thus introducing further exceptions for the attacker code to trigger. While most of these faults are typically not masked by commodity software, even an unprivileged attacker with native code execution can adjust some of these bits. In addition to mutating the page tables and configuration, TREVEX also modifies the cache state of the attacker's memory. For example, TREVEX decides to add or remove the accessed memory from the CPU's data caches or the CPU's translation lookaside buffer (TLB).

5.2. Execution and Leakage Detection

In the second stage ②, TREVEX generates an execution environment to run the previously generated attacker and victim code. This step resembles the main interaction between TREVEX and the CPU. The previously generated test cases are executed on the CPU under test, and TREVEX observes their behavior to decide whether they trigger a TEV.

For doing so, TREVEX removes noise from the execution results by repeating the execution of each test case multiple times and aggregating the results. To decide whether a test case triggers a TEV, TREVEX uses $\mathcal{T}2$(cf. Section 4.3).

Environment Setup. Before executing the generated code, TREVEX sets up the execution environment based on the parameters generated in the previous stage. For mutating bits of the page-table entries, TREVEX relies on a modified version of PTEditor [58], a kernel module exposing an interface for page-table modifications. A combination of crafted routines and PTEditor is further used to modify the TLB and cache states of the attacker's memory. Note that while some of these mutations are permanent, such as modifying the User bit in a page-table entry, others are volatile, such as modifying the Dirty bit in a page-table entry or changing the memory's cache state. Thus, TREVEX ensures that all volatile mutations are restored before repeating a test case execution. Note that the actual capabilities of an attacker to set up the environment depend on the threat model. For example, while page-table modifications are realistic for privileged attackers targeting TEEs [4], [5], [9], they are not realistic for every given scenario.

Code Execution. Given the execution environment, TREVEX executes the generated attacker and victim contexts in parallel. While the victim code executes a tight endless loop until it is stopped by TREVEX, the attacker code does not loop. It is executed a configurable number of times to reduce the influence of system noise further. As previously discussed, the attacker code encodes the architectural and transient content of a register using Flush+Reload, allowing TREVEX to store the encoded results. Afterwards, the code execution is repeated another configurable number of times to improve the quality of the measurements further. Thus, TREVEX increases the likelihood of observing transient leakage and allows reasoning about the transient behavior of multiple executions.

Leakage Detection. To determine whether the execution of the attacker code triggers a DF-TEV, TREVEX checks whether unexpected behavior occurs in the transient execution domain. Thus, this step operates on the results of the attacker code, which are reported back from the previously discussed covert channel. Our approach for detecting unexpected transient behavior ($\mathcal{T}2$) is based on the idea that DF-TEA require a transient data flow that differs from the architectural data flow. Test cases, i.e., the combination of attacker code, victim code, and environment setup, that lead to an unexpected transient data flow behavior are flagged by the data-flow divergence testing algorithm (cf. Section 4.3) are forwarded to the next steps of TREVEX's workflow.

Data-Dependency Identification. After the leakage detection, TREVEX identifies whether the transiently leaked data depends on the data that the victim context processes. For this purpose, TREVEX re-executes the test cases forwarded from the leakage detection step and follows the algorithm outlined in Section 4.4 and Appendix C.

Independent of whether there is a data dependency, TREVEX forwards the test case to the next step of its workflow, the result classification stage. This is done to ensure that vulnerabilities not relying on data flow between the attacker and victim context, e.g., value injection vulnerabilities, are detected, while enabling classification based on whether there is a data dependency.

5.3. Result Classification

In its third stage ③, TREVEX further classifies the positive results from the execution stage, i.e., test cases that show TEV behavior. Due to the number of parameters mutated by TREVEX, the number of positive test cases is often hundreds to thousands (cf. Section 6.2). Thus, the result classification is crucial for bringing the human effort for analyzing the final results down to a manageable level. Furthermore, the result classification stage uses heuristics

that throw away false positives, e.g., in case of erroneous measurements due to noise.

TREVEX classifies true positives into subclasses of TEVs. For doing so, TREVEX applies a heuristic-based classification informed by expert knowledge about TEVs. To ensure flexibility, TREVEX exposes an API that allows a human expert to conveniently define rulesets that classify the test cases forwarded by the leakage detection step. For example, the LVI-NULL [7] classification rule checks whether there has been an exception, either terminal or non-terminal, whether there has been a memory load involved, and whether the value '0' was forwarded. We provide a set of base rules that classify known vulnerabilities, such as Meltdown-US [1], MDS [5], GDS [9], and FPVI [8]. Furthermore, we provide rules that classify unknown behavior into similarity clusters. All rules are applied in a cascading manner, with each rule only applying to results that a previous rule has not classified. Thus, the API is designed to allow human experts to define rules that first classify known vulnerabilities, followed by rules that cluster unknown vulnerabilities based on their behavior.

6. Evaluation

In this section, we evaluate TREVEX's ability to discover DF-TEAs. First, we discuss the performance in terms of execution time and the effectiveness of TREVEX's clustering. We also conduct an extensive CPU fuzzing campaign across 20 microarchitectures from the x86 CPU vendors Intel, AMD, and Zhaoxin.

6.1. Instantiation

For our evaluation, we instantiate TREVEX to use a userspace process as the attacker context and a userspace process as the victim context. This privilege domain was selected as it is a common scenario for DF-TEAs [4], [5], [9], [44], [45] without requiring specific hardware features or implementations, e.g., support for trusted execution environments, hypervisor availability, or making assumptions about the kernel implementation. It is also highly portable across different CPU architectures, allowing us to test a wide range of CPUs. In addition, leakage between processes often directly correlates with exploitable leakage between further privilege domains. More specifically, the attacker and victim are executed on co-located sibling hardware threads to ensure they execute in parallel instead of just interleaved while maximizing the amount of microarchitectural resources shared between them. For TREVEX's instruction corpus, we use the complete list of x86 instructions provided by uops.info [59], which contains exactly 14 000 instructions. TREVEX tests each instruction from the corpus, and we configure it to execute 100 test cases for each, resulting in 1 400 000 test cases. We further configure TREVEX to repeat each test case 100 times to ensure stable results. Overall, this configuration results in 140 million total test case executions. We further configure TREVEX to use strict similarity for the instruction shadowing. This configuration, on the machine used for the performance evaluation (cf. Section 6.2), leads to a runtime that is similar to related work [35], [56], [60]. Thus, it enables a fair comparison and follows the fuzzer evaluation guidelines by Schloegel et al. [61].

6.2. General Performance

To assess the general performance of TREVEX, we execute it on an Intel Xeon E3-1505M v5 running Ubuntu 20.04.5 LTS with kernel 5.4.0-214-generic. Given the configuration described in Section 6.1, TREVEX finishes in 25 h and 38 min.

After the execution stage and before the classification stage, TREVEX flags 8752 test cases, i.e., instances of potential DF-TEVs. The classification discards 1791 of these test cases, e.g., due to noisy measurements, thus leaving 6961 test cases remaining. These are classified into 42 different classes based on TREVEX's heuristic-based classification. 671 are classified as instances of Meltdown-US [1], 28 as GDS [9], and 935 as MDS [4], [5], [44], [45]. 4078 test cases are classified as instances of value-injection attacks, e.g., LVI [7] or FPVI [8], [37], [38]. The remainder is split into 553 cases of behavior that are not directly classified as a concrete vulnerability but share common characteristics and 696 test cases, which are not clustered by TREVEX, as they do not match the set of base rules.

To give an idea of the time-to-bug for the discovered vulnerabilities, we inspect the results on the Intel Xeon E3-1505M v5 after the first 3 hours of fuzzing and confirm that TREVEX already discovers all mentioned vulnerabilities, i.e., Meltdown-US, MDS, FPVI, LVI(-NULL), and GDS. Furthermore, we argue the robustness of our findings based on the high number of test cases found for each vulnerability.

To evaluate TREVEX's true positive rate and classification accuracy, we randomly sample test cases and manually investigate them. To come up with a statistically significant sample size, we use a sample size calculator [62] based on the formulas by Cochran et al. [63]. We aim for a confidence level of 90 % with a margin of error of 10 %. This results in a sample size of 68 test cases, which we randomly select. Out of these 68 test cases, all but one test case can be manually reproduced, leaving us with a true positive rate of 98.5 %. For the remaining 67 test cases, we find that TREVEX's classification was correct for 71.6 % of the cases and correct towards the top 3 classifications (out of 42 classes) for 77.6 % of the cases.

We conclude that test cases flagged positively by TREVEX are very likely to be true positives and reproduce with high reliability. In addition, TREVEX's classification is accurate enough to give a good prediction of whether a vulnerability belongs to a known class of TEAs, allowing the classification to guide further analysis. For a discussion of its false negative rate, we refer to Section 6.5, showcasing that TREVEX is able to discover all vulnerabilities that we expect it to find.

6.3. Fuzzing Campaign Setup

To showcase the effectiveness of our approach, we conduct a large-scale fuzzing campaign. The goal behind this is twofold: First, we use it to further evaluate TREVEX's performance across microarchitecture. Second, we use it to analyze the status quo of DF-TEAs on x86 CPUs. We include CPUs from most microarchitectures released over the last 10 years by Intel, AMD, and Zhaoxin. We test 20 CPUs including 13 Intel, 6 AMD, and 1 Zhaoxin microarchitectures, which gives a total of 20 tested microarchitectures.

The campaign's scale shows that TREVEX's generic black-box approach allows for easily executing test cases across different microarchitectures and vendors. TREVEX can test all CPUs without significant tweaks. Adapting TREVEX to a new system only requires minor setup changes, such as disabling CPU features like Intel Turbo Boost and AMD Turbo Core to improve the measurement stability. While TREVEX's performance varies depending on the tested CPU, the performance generally remains within the same order of magnitude as the measurement on Intel Xeon E3-1505M v5. In the following, we discuss the vulnerabilities discovered by TREVEX. An overview of these results is shown in Table 1. For details on the tested systems, including the CPU model, OS, kernel, and microcode versions, we refer the reader to Table 3 in Appendix A.

6.4. Novel Vulnerabilities

In this subsection, we discuss the previously unknown vulnerabilities discovered during the fuzzing campaign.

Floating Point Divider State Sampling (FP-DSS). On the AMD Ryzen 5 2500U and AMD Ryzen 5 3550H, TREVEX reports test cases in which the attacker code could leak data from the division executed on the victim process. We refer to this previously unknown vulnerability as Floating Point Divider State Sampling (FP-DSS). To trigger the leakage, the attacker code executes either a non-faulting SSE division instruction, e.g., `DIVSD`, or a non-faulting AVX division instruction, e.g., `VDIVSD`. The victim instance, which is executed either on the same or a sibling hardware thread, also executes a division instruction. The attacker-executed division transiently forwards data that depend on the victim's inputs to the division. This enables data leakage from the victim process to the attacker process. While we observe that attacker-controlled AVX division instructions leak data from victim-controlled AVX divisions, and that the same works for SSE divisions, we do not observe data leakage from AVX to SSE divisions, or vice versa.

The affected machines, i.e., AMD Ryzen 5 2500U and AMD Ryzen 5 3550H, belong to AMD's Zen and Zen+ microarchitectures, respectively, and are not affected by DSS [64], a previously known vulnerability that leaks data from the *integer* division unit. We verify that the existing mitigations against DSS [65], [66], i.e., clearing the integer division unit's state, do not prevent the FP-DSS leakage. We hypothesize that the data leakage is from an intermediate buffer of AMD's *floating-point* division unit. As the floating-point and integer division units are separate on AMD CPUs, our hypothesis would explain why the DSS mitigations do not prevent the leakage. We responsibly disclosed the vulnerability to AMD. AMD confirmed the finding and requested an embargo. They further informed us that the vulnerability will be assigned CVE-2025-54505 and that they are working on security guidance for their customers.

To demonstrate the security impact of this vulnerability, we verify that an unprivileged attacker can leak data from another process, the Linux kernel, and a browser instance. The latter is based on the fact that the JavaScript engines of both Firefox and Chrome emit instructions vulnerable to FP-DSS. This shows that this leakage can even be exploited by a malicious website. For the browser exploit, we use WebAssembly to craft a floating-point division (`f64.div`) and encode the result of the division into the cache using a `Uint8Array` access. For the recovery of encoded values, we employ Evict+Reload with a straightforward eviction of repeatedly accessing $8\,\mathrm{MB}$ of memory. We successfully verified FP-DSS leakage from an unprivileged process in Chrome 140.0 (AMD Ryzen 5 3550H, microcode 0x8108102, Ubuntu 22.04.1). Moreover, we use the leakage as a side channel, as shown by Weber et al. [55], to successfully monitor keystrokes in GUI applications. Appendix B provides more details on the keystroke attack and shows a plot where real keystroke timings perfectly match the timestamps when we see divider leakage.

To analyze the leakage bandwidth, we construct a cross-process covert channel using FP-DSS that does not rely on shared memory. The covert channel uses FP-DSS to transmit information from a sender process to a receiver process. During each fixed time interval of $25\,000$ CPU cycles, the sender executes SSE divisions that produce 1 of 9 distinct results. 8 of these results encode 3 bits of data, while the 9th indicates the start of a new 3-bit transmission. The receiver on the sibling core continuously executes an SSE division that architecturally produces '0' and encodes the non-zero transient result into the cache. It then applies Flush+Reload to 9 cache lines to recover the transmitted value. This design enables efficient multi-bit transmission per encoding event while maintaining reasonable bandwidth on the receiver side. We evaluate the covert channel by transferring $50\,\mathrm{kB}$ of data. The channel achieves a transmission rate of $158.732\,\mathrm{kbit/s}$, with an error rate of $0.38\,\%$.

FPVI without Denormal Input Values. On several AMD CPUs, including the AMD Ryzen AI 9 HX 370 belonging to one of AMD's newest microarchitecture generations (Zen 5), TREVEX discovers that vector instructions may transiently forward incorrect values even in the absence of denormal input values. This behavior conflicts with the known conditions required for FPVI [8], which rely on the presence of denormal input values. To give an example of a concrete instance, the SSE instruction `MULPD XMM1, XMM2`, which is used to multiply two packed double-precision floating point values, transiently forwards incorrect values to the output register. The behavior is sufficiently similar to the FPVI vulnerability [8], [37], [38], resulting in similar exploitation

primitives, i.e., attackers can inject incorrect vector values into victim applications.

We responsibly disclosed this behavior to AMD. AMD confirmed that this is an alternative way to trigger FPVI on their CPUs, requested an embargo for the finding, and informed us about an upcoming security brief. We could not reproduce the behavior on Intel CPUs.

Zero-at-ret. TREVEX discovers instances of '0' values being forwarded in the transient domain on multiple Intel CPUs (cf. Table 1) marked as unaffected by LVI-NULL [31]. For example, on the Intel Xeon Gold 6346, TREVEX reports an instance where the instruction `ADOX`, e.g., with the full encoding `ADOX RSI, [RDX]`, transiently forwards the value '0' to the output register. TREVEX was able to trigger this behavior by clearing the access bit in the page-table entry of the memory page pointed to by `RDX`. We confirm that clearing the access bit is sufficient to forward the incorrect computation result transiently. We initially attributed this behavior to a microcode assist and hypothesized that it is a variant of the LVI-NULL vulnerability. During our disclosure process, Intel indicated that the findings correspond to instances of *Zero-at-ret* [36]. Publicly available information on Zero-at-ret appears to be limited to a brief mention within the LVI Security guidance [36], without any accompanying list of affected CPUs [31].

To evaluate Zero-at-ret's exploitability and compare its behavior with known LVI-NULL-affected systems, we test the SGX proof-of-concept attacks by Giner et al. [67]. We confirm that the indirect jump gadget attack succeeds on Intel Core i3-1005G1, achieving a success rate exceeding 99.9 %. These gadgets enable arbitrary code execution in the transient domain and can be used to leak enclave secrets. They first load a pointer to a function pointer table from memory. Then, a function pointer is fetched from the table and used as an indirect jump target. With our variant of LVI NULL, a zero is injected into the table pointer load, causing the CPU to fetch the jump target from virtual address '0' in the transient domain. Because the SGX threat model permits the attacker to map address '0', control flow can be transiently redirected [68].

During the disclosure process, Intel confirmed that, from an attacker's point of view, the difference between Zero-at-ret and the original LVI-NULL vulnerability is only the length of the transient window. While our disclosed proof-of-concepts are not capable of leaking *arbitrary* data, Intel informed us that, due to our disclosure, they are currently internally re-evaluating the security impact of Zero-at-ret.

FPVI on Zhaoxin. On the Zhaoxin KX-U6780A, TREVEX creates code that transiently forwards stale floating point values, similar to FPVI [8], [37], [38]. We found that on the affected Zhaoxin CPU, these instructions transiently forward the values of their input operands to the output register if one of the input operands contains a denormal floating point value. While most instructions showing this behavior are VEX-encoded, not all of them are. For example, the instruction `SUBPD XMM1, XMM2` also shows this behavior. If `XMM2` contains a denormal floating point value, `SUBPD` transiently forwards its value to `XMM1`. Similarly, if `XMM1` contains the denormal floating point value, `SUBPD` transiently computes on the original value of `XMM1`, rather than on the result of the subtraction. While this behavior features characteristics of FPVI [8], [37], [38], FPVI is not known to affect Zhaoxin. Nevertheless, the behavior enables similar attacks, i.e., injecting incorrect floating point values into victim applications [8]. We contacted Zhaoxin to responsibly disclose this finding.

TABLE 1: TEVs discovered by TREVEX during our fuzzing campaign. We mark TEVs as ● if they were discovered and already documented on the CPU, ★ if they were discovered and previously unknown for the CPU, and ▲ if new variants were discovered on CPUs known to be affected. Also, - is displayed for CPUs not affected by the vulnerability.

	Microarch.	MD	MDS[a] LVI	FPVI	LVI-NULL	GDS	FP-DSS
Intel	Skylake	●	●	●	●	●	-
	Kaby Lake	●	●	●	●	●	-
	Coffee Lake-HR	-	●	●	●	●	-
	Comet Lake	-	●	●	●	●	-
	Ice Lake	-	●	●	★	●	-
	Tiger Lake	-	-	●	★	●	-
	Elkhart Lake	-	-	●	-	-	-
	Rocket Lake	-	-	●	●	●	-
	Ice Lake-SP	-	-	●	★	●	-
	Alder Lake	-	-	●	-	-	-
	Alder Lake-N	-	-	●	-	-	-
	Raptor Lake	-	-	●	-	-	-
	Meteor Lake	-	-	●	-	-	-
AMD	Zen	-	-	▲	-	-	★
	Zen+	-	-	▲	-	-	★
	Zen 2	-	-	▲	-	-	-
	Zen 4	-	-	▲	-	-	-
	Zen 5	-	-	▲	-	-	-
	Zen 5/5c	-	-	▲	-	-	-
Zhaoxin	LuJiaZui	-	-	★	-	-	-

[a] Our definition of MDS includes Vector Register Sampling (VRS) [69].

6.5. Rediscovering Known Vulnerabilities

TREVEX successfully rediscovers all vulnerabilities that we expect it to find. TREVEX successfully discovers Meltdown-US [1], Microarchitectural Data Sampling (MDS) [4], [5], Load Value Injection(-NULL) [7], and Gather Data Sampling (GDS) [9] on affected CPUs. In addition, TREVEX discovers Floating Point Value Injection (FPVI) [8] on all tested CPUs. While Intel explicitly documents that its CPUs are affected by FPVI, AMD's last statement on the vulnerability dates back to 2021 [37]. Nevertheless, our fuzzing campaign results show that even AMD's novel microarchitectures are still affected by FPVI.

While our approach can identify Meltdown-CPL-REG [3], [70], [71], the tested CPUs are only affected by this vulnerability if access to system registers is explicitly disabled, e.g., by setting the kernel parameter `nofsgsbase`. As there is already work by Weber et al. [72] on analyzing the remaining spread of Meltdown-CPL-

REG, we opt to only exemplarily test TREVEX's ability to discover this vulnerability. For this purpose, we ensure that userspace access to Intel Xeon E3-1505M v5's `fsbase` and `gsbase` registers is disabled and execute TREVEX. Given this vulnerable configuration, TREVEX successfully discovers Meltdown-CPL-REG. Similarly, we verify that TREVEX is capable of discovering the Foreshadow vulnerability [6]. For Divider State Sampling (DSS) [18], we manually verify that TREVEX outputs code capable of triggering the vulnerability. As we do not have access to a CPU that is affected by DSS [64], we cannot further verify the DSS discovery. We conclude that TREVEX is reliable in rediscovering known vulnerabilities, demonstrating its applicability to regression testing and showing its general effectiveness in discovering a diverse set of DF-TEVs. Speculative Store Bypass (SSB) [47] requires a sophisticated code pattern to be triggered. A typical example for SSB is a store to address `A` using a pointer `AP1` followed by another store to `A` using a pointer `AP2`, which is delayed in its execution, and a subsequent load from `A` using the pointer `AP1`. The delay of updating `A` via `AP2` then leads to a transiently bypassed store. To trigger SSB, the second store using `AP2` can be delayed via an indirection over a second uncached memory pointer to `A` or a pointer chase [47], [73]. Due to limitations in TREVEX's current code generation strategy and the complexity that SSB triggering code requires, we do not expect TREVEX to discover SSB. We verify that this is not a design limitation but rather a PoC limitation that further engineering can overcome. We empirically validate that when TREVEX emits code that triggers SSB behavior, the test case is flagged as a vulnerability, demonstrating that the general design is capable of discovering SSB.

6.6. Effectiveness of TREVEX's Key Techniques

The key components of TREVEX are the techniques introduced in Section 4. In the following, we discuss the effectiveness of these techniques in the context of the vulnerabilities discovered by TREVEX. We elaborate how instruction shadowing was a requirement to discover FP-DSS. We discuss how data-flow divergence testing is crucial to reason about actual exploitable vulnerabilities in an automated manner while staying sufficiently generic to find a broad range of vulnerabilities, e.g., enabling TREVEX to also find value-injection TEVs such as FPVI [8], [37], [38] and LVI [7], [46] *and* data-leakage TEVs, such as Meltdown-US [1], MDS [4], [5], [44], [45], and GDS [9]. Finally, we discuss how cross-privilege domain data-dependency analysis allows TREVEX to flag vulnerabilities with cross-privilege data flow, thus enabling human analysts to prioritize TREVEX's results.

Microarchitectural Buffer Tainting via Instruction Shadowing. The discovery of FP-DSS demonstrates the success of TREVEX's instruction shadowing. We manually verify that FP-DSS only exposes leakage during transient execution when the floating point division unit was used very recently. We do not observe any transient forwarding of incorrect values on an idle Ubuntu 22.04 system, which we attribute to the pressure on the floating point division unit being too low. While stressed systems could trigger such behavior, they make it hard to identify the leakage source correctly. Hence, we conclude that TREVEX's instruction shadowing, which enables it to taint *arbitrary* microarchitectural buffers, is crucial to discover vulnerabilities like FP-DSS.

Data-Flow Divergence Testing. TREVEX's data-flow divergence testing algorithm ensures that it only reports transient windows when data is actually forwarded to a register that an attacker can recover from the transient domain. Thus, TREVEX is able to filter out false positives that would be reported by simpler tests, such as the ones used by previous works [35], [56]. Previous black-box approaches have either opted for mechanisms reporting every code snippet creating a transient window [35], [56] or only report transiently encoded data after terminal faults [15]. Furthermore, the algorithm does not require any emulated ground-truth of the CPU like MRT-based approaches [17], [18], [19], [20], which makes it applicable to arbitrary code snippets without requiring knowledge about its semantics. This enables TREVEX to test complex instructions (e.g., floating point instructions, which are not handled by the existing MRT-based approaches [17]) and also undocumented instructions, as it does not require knowledge of an instruction's semantics.

Cross-Privilege Domain Data-Dependency Detection. The success of TREVEX's cross-privilege domain data-dependency detection is demonstrated by its ability to flag data-leakage TEVs, e.g., MDS, GDS, and FP-DSS. Flagging data-leakage TEVs is beneficial, as it allows analysts to prioritize the inspection of these vulnerabilities due to their typical higher severity. For example, TREVEX flags certain test cases on the AMD Ryzen 5 2500U and AMD Ryzen 5 3550H as having cross-privilege data dependencies, despite these CPUs not being known to be affected by data-leakage TEVs previously. This immediately drew our attention to the test cases triggering FP-DSS.

7. Comparison to Prior Work

In this section, we discuss previous work on CPU vulnerability discovery. We discuss TEV detection approaches, as they are the most comparable to TREVEX. In addition, we discuss related approaches for domain-specific vulnerability discovery, such as side-channel leakage and architectural CPU vulnerabilities. While these approaches are also considered CPU fuzzing, their goals are orthogonal.

7.1. Black-Box Transient Execution Vulnerability Discovery

Black-box TEV discovery approaches aim to discover TEVs without requiring any knowledge about the CPU's internal design. In other words, the CPU is treated as a black box, making these approaches applicable to open- *and* closed-source CPUs. This allows independent researchers to discover vulnerabilities in commercial CPUs, where the design is not publicly available. In the following, we discuss the most relevant black-box approaches for TEV detection.

TABLE 2: Comparison of TREVEX and previous post-silicon black-box DF-TEV detection.

Tool	Cross-Priv. Tainting	Tainting Unknown Buffers	Detects Data Dependencies	Classification	Novel Vuln.
SpeechMiner[a][74]	✗	✗	✗	✗	✗
Transynther [15]	✓	✗	✗	✓	✓
RegCheck [72]	✗	✗	✗	✓	✗
Shesha [35]	✗	✗	✗	✗	✗
μRL [56]	✗	✗	✗	✗	✗
TREVEX	✓	✓	✓	✓	✓

[a] SpeechMiner's focus is primarily focused on assisting human analysis instead of automated vulnerability discovery.

SpeechMiner [74] is an early approach for tool-assisted analysis of TEAs. While SpeechMiner's focus is not on automation but rather on assisting human analysts, some of its building blocks are still relevant for automated TEV discovery. Thus, we include it in our comparison for completeness. Transynther [15] interleaves parts of known Meltdown-type attacks to generate new variants. This approach uncovered a new MDS variant and a regression bug affecting Intel CPUs [15], [16]. Other tooling, e.g., RegCheck [72], focusses on detecting variants of known vulnerabilities, such as Meltdown-CPL-REG [3], [70], [71]. Such highly-targeted approaches, however, inherently cannot discover more than variants of existing vulnerabilities, as they are limited to building blocks of existing attack patterns while focusing their detection on known behavior.

The Shesha framework [35] aims to discover TEVs by searching for novel triggers for transient execution windows. It uses a guidance strategy based on particle swarm optimization and performance counters for vulnerability detection. Similarly, a recent approach by Tol et al. [56] explores performance-counter-based guidance techniques in combation with reinforcement learning for guidance. However, both approaches [35], [56] are merely designed for detecting novel triggers of transient execution without reasoning about the resulting control- or data-flow behavior or their security impact. Thus, all reported triggers for transient windows of the approaches require further manual analysis to determine whether they may lead to TEVs. Thus, these approaches leave space for improvement, which is also reflected by the fact that these works did not discover any novel TEV. Furthermore, the impact of their respective guidance mechanisms has not been evaluated and thus remains unclear.

Improvements by TREVEX. TREVEX improves significantly upon previous black-box approaches by introducing novel techniques that allow it to discover new types of TEVs. Table 2 provides an overview of the main differences between TREVEX and previous approaches.

Previous approaches either do not taint the microarchitecture [35], [55], [56], [74] or only taint a fixed set of microarchitectural buffers, e.g., the memory subsystem [15]. While the memory subsystem is responsible for some TEVs, a mere focus on tainting the memory subsystem is insufficient to discover all TEVs, as discussed in Section 6.6. In contrast, TREVEX's instruction shadowing technique allows it to taint arbitrary microarchitectural buffers, as long as the currently tested instructions can influence the buffer. This enables TREVEX to discover novel vulnerabilities such as FP-DSS and new FPVI variants (cf. Section 6.6).

Moreover, previous approaches either do not trigger cross-privilege leakage, as they do not taint across privilege domains [35], [55], [56], [74], or they only detect fixed values injected by a victim process [15]. In contrast, TREVEX taints the microarchitecture from foreign cross-privilege domains while not assuming a one-to-one mapping of the tainted values. Consequently, TREVEX can detect data dependencies between tainted values and leaked values, e.g., as exploited in FP-DSS and discussed in Section 6.6. Furthermore, TREVEX is able to reason about arbitrary data-flow divergence in the attacker code.

Lastly, we consider TREVEX's classification stage as a key component for a mature vulnerability detection tool. While Moghimi et al. [15] already introduced a classification stage, TREVEX improves over its design, as the gathered information in combination with TREVEX's API, allow it to improve the classification. In other words, TREVEX's rule-based API allows for a straightforward extension of the classification stage by human experts. While RegCheck [72] classifies subcategories of Meltdown-CPL-REG, it does so solely based on the instruction exposing the vulnerability, as its limited scope (Meltdown-CPL-REG subcategories) does not require a more sophisticated classification.

7.2. Alternative Approaches for Transient Execution Vulnerability Discovery

In the following, we discuss model-based post-silicon approaches and RTL-based pre-silicon approaches for TEV detection. These approaches fundamentally differ from the previously discussed black-box approaches, as they require either a reference model or access to the CPU's design before fabrication. For this reason, they are limited in their applicability—especially pre-silicon approaches, which are infeasible for independent research on the vast majority of commercial CPUs.

Model-Based Approaches. Model-based post-silicon approaches [17], [18], [19], [20] leverage formal reasoning about a microarchitecture. Assuming a formal model of a microarchitecture, such as hardware-software contracts [75], any deviation from this model indicates a potential vulnerability. The Revizor framework [17], [18], [19], [20] leverages model-based relational testing to discover violations for leakage contracts and works as follows. First, Revizor generates test cases that are executed with varying

inputs under a given execution. This execution generates *contract traces*. Second, Revizor executes the test cases on the target CPU while monitoring the cache state to generate *hardware traces*. When the contract traces of two executions match but the hardware traces differ, Revizor flags the executions as a contract violation. Until recently, Revizor created contract traces using the Unicorn engine as an ISA simulator. While this yields a powerful approach for discovering vulnerabilities, it is strictly bound by the capabilities of the emulator. For example, Hofmann et al. [18] exclude floating point instructions, SIMD, and CFI instructions. Furthermore, emulators naturally do not model undocumented instructions. Recently, the Revizor framework added support for DynamoRIO-based contract trace generation [76]. The DynamoRIO backend allows Revizor to generate contract traces by instrumenting a native binary instead of relying on the Unicorn emulation. While Revizor and TREVEX are both capable of identifying vulnerabilities, their underlying goals explain their different designs. Revizor's goal is to test whether a CPU obeys a given model in the form of a leakage contract. If a violation is found for a leakage model, including the leakage of all known CPU vulnerabilities, it indicates a new vulnerability. Thus, while Revizor does not allow for formally proving that a CPU follows a given leakage contract, it can be used to heuristically test the validity of a given contract. TREVEX is designed solely to discover vulnerabilities and classify them by their behavior. Hence, TREVEX does not require prior modeling of the CPU's microarchitecture in a leakage contract and can be used to discover vulnerabilities without assumptions on the microarchitecture.

Pre-Silicon Approaches. Various pre-silicon approaches have been proposed to discover TEVs based on the CPU's RTL [12], [23], [24], [25], [26], [27], [28], [29], [30]. RTL-based approaches leverage detailed access to the CPU's internals and can thus partially reason about the microarchitecture. On the downside, they come with two main limitations. First, they commonly suffer from limited performance and scalability issues [54], [77]. In this context, Thomas et al. [54] showed that unguided post-silicon approaches can easily outperform guided RTL-based approaches by several orders of magnitude, which was further backed by the independent work of Bölcskei et al. [77]. Second, pre-silicon approaches naturally require access to a CPU's RTL, which is generally not publicly available for commercial CPUs. This makes them unsuitable for independent research on these CPUs. Even if available, recent research has shown that even CPUs marketed as open-source are often only partially open-source and can contain severe bugs in the non-public components [54].

7.3. Domain-Specific Vulnerability Tooling

In addition to tools searching for TEVs, there are several domain-specific tools that focus on automated side-channel discovery [43], [78], [79] and automated discovery of architectural CPU vulnerabilities [54], [80], [81], [82], [83].While these architectural vulnerabilities are also rooted in a CPU's microarchitecture, their discovery is orthogonal to TEAs due to the fundamental differences in their nature (cf. Section 2). Furthermore, prior work [8], [74], [84] has focused on reversing the transient execution behavior of CPUs to deepen understanding of the root causes of TEVs. In contrast to TREVEX, these tools only focus on fundamental properties of TEVs and not on their discovery.

7.4. Orthogonal Pre-Silicon Approaches

Leakage contracts have been studied extensively for providing guarantees to software applications [75], [85], [86], [87], [88], [89]. Guarnieri et al. [75] introduced a framework for formalizing the allowed side-channel leakage contained in a program in the form of leakage contracts. Wang et al. [85] adapted their framework to run on RTL designs, thus allowing to verify whether the RTL design satisfies contract properties for a given program. The works by Hsiao et al. [88], [89] and Wang et al. [87] proposed automated approaches to synthesize leakage contracts for a given RTL design. Bloem et al. [86] extended the idea of leakage contracts to power side-channel attacks and developed a technique to verify their contracts directly against a CPU's netlist, i.e., the CPU's description after synthesis.

Writing constant-time code requires *safe* instructions, i.e., instructions whose timing is independent of their operands. ConjunCT [90] and VeloCT [91] aim to derive such sets of instructions by automatically reasoning about a CPU's RTL design. ConjunCT combines symbolic analysis and inductive invariant learning to decide whether an instruction is safe. VeloCT improves over the ConjunCT's performance by leveraging the underlying problem structure.

8. Limitations and Future Improvements

In this section, we discuss the limitations and potential future improvements of TREVEX.

Other Architectures. TREVEX currently only supports x86 but due to its modular design and the fact that it does not rely on any x86-specific features, can be ported to other architectures, e.g., ARM or RISC-V. Since the number of known TEVs on those architectures is lower, and research into ARM and RISC-V silicon is generally limited, we choose to demonstrate TREVEX on x86 and leave the porting to other architectures as future work.

Control- vs. Data-flow Transient Execution Vulnerabilities. As TREVEX is designed to discover DF-TEVs, it currently cannot be used to find CF-TEVs, e.g., Spectre-BTB. While TREVEX can be extended to generate code for its attacker context that triggers CF-TEVs, this is orthogonal to TREVEX's goal (cf. Section 3). As discussed in Section 2 and Section 3, most CF-TEVs are caused by branch prediction units, which are typically documented for a CPU's design and well-understood [2], [32], [33], [34]. In that sense, CF-TEVs are fundamentally different from DF-TEVs, which are typically caused by unintentional bugs, as opposed to intentional design decisions. Additionally, the root cause of DF-TEVs is generally difficult to attribute and

thus much more complex to discover. Whereas CF-TEVs are typically easier to detect due to the underlying design choices being well-documented [2], [32], [33], [34], making it possible to enumerate a CPU's predictors.

Code Generation. TREVEX uses instruction shadowing to taint and leak from arbitrary microarchitectural buffers. While we show that this is an improvement over the state-of-the-art, it is not fully generic and can still miss some leakage channels. For example, leakage scenarios where the pair of tainting and leaking instructions are fundamentally different types are generally challenging to discover in an efficient manner. TREVEX could be extended to support arbitrary instruction pairs, similar to Osiris [43]. This would, however, significantly increase the complexity of the search space and thus TREVEX's runtime. For this reason, we decided to focus on instruction pairs that are more likely to induce leakage.

All CPU fuzzers, including TREVEX, face the limitation that they can only find vulnerabilities in code that they actually generate. Thus, TREVEX could miss vulnerabilities that require very specific instruction sequences that are unlikely to be generated by the current proof-of-concept. However, TREVEX already uncovers special variants of vulnerabilities (cf. Section 6.3), showing that it already succeeds at triggering complex microarchitectural conditions. Nevertheless, our proof-of-concept implementation could be extended with a more complex sequence generation technique to test further complex microarchitectural conditions. This would, however, again significantly increase the complexity of the search space and thus the runtime of TREVEX.

9. Conclusion

Our work demonstrates that black-box detection frameworks can overcome previous limitations and play a crucial role in the discovery of TEVs in commercial CPUs. With TREVEX, we designed and evaluated a black-box vulnerability detection framework capable of finding generic DF-TEVs.

Running TREVEX across 20 x86 microarchitectures from Intel, AMD, and Zhaoxin produced 4 prominent results. First, it discovered Floating Point Divider State Sampling (FP-DSS), a previously unknown TEA. We show that FP-DSS can be weaponized from unprivileged native code to leak kernel memory. We further show that JavaScript running inside Google Chrome can exploit the vulnerability to attack native processes. Second, TREVEX revealed that Zhaoxin CPUs are affected by FPVI and that FPVI can be triggered without involving denormal input values on AMD CPUs. Third, TREVEX exposes 3 instances of Zero-at-ret, a CPU vulnerability for which the list of affected CPUs is undocumented. Finally, it rediscovered every DF-TEV that we expect from current public information, including FPVI and GDS, demonstrating that the approach is both generic and reliable. Our findings underline the need for continuous and more generic vulnerability discovery frameworks, and we believe TREVEX is a solid step in that direction.

Ethics Considerations

This work led to the discovery of multiple previously unknown vulnerabilities. Thus, we make sure to follow well-established responsible disclosure processes and disclose all vulnerabilities to the respective vendors. To ensure that vendors have sufficient time for response and mitigation strategies, we have disclosed our findings more than 90 days before publically releasing our findings. This also allows vendors to request an embargo period, if they wish to do so. AMD has requested an embargo period, which we granted. The initial embargo ended on March 16, 2026 and was extended to April 17, 2026, after AMD requested an extension. While Intel did not request an embargo period, they were informed about AMD's embargo. We work closely with the affected vendors and will provide them with any information requested.

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.

Acknowledgment

We thank the reviewers for their valuable feedback and suggestions. We thank Lukas Gerlach, Sara-Elena Vatavu, and Luis Wollenschneider for their contributions to exploratory experiments during this project. We thank Martin Schwarzl and Tristan Hornetz for their constructive feedback and helpful discussions. We thank AMD, Intel, and Zhaoxin for their close collaborations during the responsible disclosure process. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491039149. This work was also partly supported by the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS).

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] 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.

[3] 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/.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] Intel Corporation, "Register File Data Sampling," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/register-file-data-sampling.html

[11] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, "Analyzing and exploiting branch mispredictions in microcode," *arXiv preprint*, 2025.

[12] A. de Faveri Tron, R. Isemann, H. Ragab, C. Giuffrida, K. von Gleissenthall, and H. Bos, "Phantom trails: Practical pre-silicon discovery of transient data leaks," in *USENIX Security*, 2025.

[13] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "Flop: Breaking the apple m3 cpu via false load output predictions," in *USENIX Security*, 2025.

[14] J. Kim, D. Genkin, and Y. Yarom, "Slap: Data speculation attacks via load address prediction on apple silicon," in *S&P*, 2025.

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

[16] D. Moghimi, "Data sampling on mds-resistant 10th generation intel core (ice lake)," *arXiv preprint arXiv:2007.07428*, 2020.

[17] 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.

[18] 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.

[19] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, "Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing," in *IEEE S&P*, 2023.

[20] O. Oleksenko, F. Solt, C. Fournet, J. Hofmann, B. Köpf, and S. Volos, "Enter, exit, page fault, leak: Testing isolation boundaries for microarchitectural leaks," *arXiv preprint*, 2025.

[21] C. Domas, "Breaking the x86 isa," *Black Hat*, 2017.

[22] F. Strupe and R. Kumar, "Uncovering hidden instructions in Armv8-A implementations," in *HASP*, 2020.

[23] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *ISCA*, 2021.

[24] J. Hur, S. Song, S. Kim, and B. Lee, "Specdoctor: Differential fuzz testing to find transient execution vulnerabilities," in *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[25] M. R. Fadiheh, A. Wezel, J. Müller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An exhaustive approach to detecting transient execution side channels in rtl designs of processors," *IEEE Transactions on Computers*, 2022.

[26] G. Cabodi, P. Camurati, F. Finocchiaro, and D. Vendraminetto, "Model-checking speculation-dependent security properties: Abstracting and reducing processor models for sound and complete verification," *MDPI Electronics*, 2019.

[27] M. Balliu, M. Dam, and R. Guanciale, "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis," *arXiv:1911.00868*, 2019.

[28] J. Xu, Y. Zhou, X. Zhang, Y. Li, Q. Tan, Y. Zhang, Y. Zhou, R. Chang, and W. Shen, "DejaVuzz: Disclosing Transient Execution Bugs with Dynamic Swappable Memory and Differential Information Flow Tracking assisted Processor Fuzzing," *arXiv preprint*, 2025.

[29] K. Ceesay-Seitz, F. Solt, and K. Razavi, "μcfi: Formal verification of microarchitectural control-flow integrity," in *ACM CCS*, 2024.

[30] Q. Tan, Y. Yang, T. Bourgeat, S. Malik, and M. Yan, "RTL Verification for Secure Speculation Using Contract Shadow Logic," in *ASPLOS*, 2025.

[31] Intel, "Affected Processors: Transient Execution Attacks," 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html

[32] O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.

[33] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.

[34] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.

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

[36] Intel, "Software Security Guidance: Load Value Injection," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/load-value-injection.html

[37] AMD, "Speculative Code Store Bypass and Floating-Point Value Injection," 2021. [Online]. Available: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1003.html

[38] Intel, "Floating Point Value Injection," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/floating-point-value-injection.html

[39] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.

[40] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.

[41] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.

[42] 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.

[43] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated Discovery of Microarchitectural Side Channels," in *USENIX Security*, 2021.

[44] 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.

[45] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, 2021.

[46] Intel, "Deep Dive: Load Value Injection," 2020. [Online]. Available: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-load-value-injection

[47] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018. [Online]. Available: https://project-zero.issues.chromium.org/issues/42450580

[48] AMD, "Security analysis of amd predictive store forwarding," 2023. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/security-analysis-of-amd-predictive-store-forwarding.pdf

[49] Y. Fan, Y. Jin, C. Liu, M. Sun, X. Song, T. Yin, and S. Deng, "Drow: Training-free load speculative execution attacks on apple silicon," *uASC*, 2026.

[50] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, 1990.

[51] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, 2008.

[52] AFL, "More about AFL - History," 2025. [Online]. Available: https://afl-1.readthedocs.io/en/latest/about_afl.html

[53] K. Serebryany, "Oss-fuzz - google's continuous fuzzing service for open source software," 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[54] F. Thomas, L. Hetterich, R. Zhang, D. Weber, L. Gerlach, and M. Schwarz, "RISCVuzz: Discovering architectural CPU vulnerabilities via differential hardware fuzzing," https://ghostwriteattack.com/, 2024.

[55] D. Weber, F. Thomas, L. Gerlach, R. Zhang, and M. Schwarz, "Indirect Meltdown: Building Novel Side-Channel Attacks from Transient Execution Attacks," in *ESORICS*, 2023.

[56] M. C. Tol, K. Derya, and B. Sunar, "μRL: Discovering Transient Execution Vulnerabilities Using Reinforcement Learning," *arXiv preprint arXiv:2502.14307*, 2025.

[57] Intel, "Speculative Code Store Bypass," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-code-store-bypass.html

[58] M. Schwarz, M. Lipp, and C. Canella, "misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8," 2018. [Online]. Available: https://github.com/misc0110/PTEditor

[59] A. Abel and J. Reineke, "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures," in *ASPLOS*, 2019.

[60] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: Cpu fuzzing via intricate program generation," in *USENIX Security*, 2024.

[61] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *IEEE S&P*, 2024.

[62] Standard Insights, "Sample Size Calculator," 2025. [Online]. Available: https://standard-insights.com/tools/sample-size-calculator/

[63] W. G. Cochran, "Sampling techniques, 3rd edition," 1977.

[64] AMD, "Speculative Leaks Security Notice - AMD-SB-7007," 2023. [Online]. Available: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7007.html

[65] Linux Maintainers, "Fix the DIV(0) inital fix attempt," 2023. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f58d6fbcb7c848b7f2469be339bc571f2e9d245b

[66] Xenproject.org Security Team, "XSA-439: Divide speculative information leak," 2023. [Online]. Available: https://xenbits.xen.org/xsa/advisory-439.html

[67] L. Giner, A. Kogler, C. A. Canella, M. Schwarz, and D. Gruss, "Repurposing segmentation as a practical lvi-null mitigation in sgx," in *USENIX Security Symposium*, 2022.

[68] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," in *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[69] Intel, "Vector Register Sampling / CVE-2020-0548 / INTEL-SA-OO329," 2020. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/vector-register-sampling

[70] ARM, "Cache speculation side-channels," 2018. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/Cache_Speculation_Side-channels_03May18.pdf

[71] Intel, "Instructions affected by rogue system register read," 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/instructions-affected-rogue-system-register-read.html

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

[73] IAIK, "Transient Fail (GitHub)," 2020. [Online]. Available: https://github.com/IAIK/transientfail

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

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

[76] The Revizor Project Contributors, "Revizor Releases: v2.0.0," 2026. [Online]. Available: https://github.com/microsoft/side-channel-fuzzer/releases/tag/v2.0.0

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

[78] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS*, 2020.

[79] C. Chen, J. Cui, and J. Zhang, "Beta: Automated black-box exploration for timing attacks in processors," *arXiv:2410.16648*, 2024.

[80] C. Domas, "Breaking the x86 ISA," *Black Hat US*, 2017.

[81] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," *arXiv:2110.11519*, 2021.

[82] 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.

[83] T. Ormandy, "Zenbleed," 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

[84] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations," in *ACM ACSAC*, 2019.

[85] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," in *CCS*, 2023.

[86] R. Bloem, B. Gigerl, M. Gourjon, V. Hadzic, S. Mangard, and R. Primas, "Power contracts: Provably complete power leakage models for processors," in *ACM CCS*, 2022.

[87] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Synthesis of sound and precise leakage contracts for open-source risc-v processors," in *ACM CCS*, 2025.

[88] Y. Hsiao, D. P. Mulligan, N. Nikoleris, G. Petri, and C. Trippel, "Synthesizing formal models of hardware from rtl for efficient verification of memory model implementations," in *MICRO*, 2021.

[89] Y. Hsiao, N. Nikoleris, A. Khyzha, D. P. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "Rtl2mμpath: Multi-μpath synthesis with applications to hardware security verification," in *MICRO*, 2024.

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

[91] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-houdini: Scalable invariant learning," in *ASPLOS*, 2025.

[92] J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.

Appendix A. Versions

Table 3 lists the tested microarchitectures, CPUs, and respective software versions used in our fuzzing campaign. The table includes the tested Ubuntu version, kernel version, and microcode version for each CPU. It spans Intel, AMD, and Zhaoxin CPUs, covering a wide range of microarchitectures released over the last 10 years.

TABLE 3: The full list of microarchitectures and respective CPUs tested in our fuzzing campaign. For each CPU, we list the tested Ubuntu version, the kernel version, and the microcode version. For the Zhaoxin KX-U6780A, we could not find a documented way to retrieve the microcode version.

Microarch.	CPU Model	Ubuntu Ver.	Kernel Ver.	Microcode
Skylake	Intel Xeon E3-1505M v5	20.04.5 LTS	5.4.0-214-generic	0xd6
Kaby Lake	Intel Core i3-7100T	22.04.4 LTS	5.15.0-141-generic	0xf8
Coffee Lake-HR	Intel Core i9-9980HK	24.04.1 LTS	6.1.0-060100-generic	0xaa
Comet Lake	Intel Core i7-10510U	22.04.3 LTS	5.15.0-118-generic	0xc6
Ice Lake	Intel Core i3-1005G1	24.04.1 LTS	6.1.0-060100-generic	0x70
Tiger Lake	Intel Core i7-1185G7	22.04.5 LTS	5.15.0-138-generic	0x72
Elkhart Lake	Intel Atom x6425E	24.04 LTS	6.1.0-060100-generic	0x1a
Rocket Lake	Intel Core i7-11700	22.04.5 LTS	5.15.0-143-generic	0x1c
Ice Lake-SP	Intel Xeon Gold 6346	22.04.5 LTS	5.15.0-141-generic	0xd000332
Alder Lake	Intel Core i9-12900K	22.04.4 LTS	5.15.0-47-generic	0xf
Alder Lake-N	Intel N100	22.04.4 LTS	5.15.0-122-generic	0x1a
Raptor Lake	Intel Core i5-13420H	24.04.2 LTS	6.1.0-060100-generic	0x4128
Meteor Lake	Intel Core Ultra 7 155H	24.04.2 LTS	6.1.0-060100-generic	0x24
Zen	AMD Ryzen 5 2500U	24.04.1 LTS	6.1.0-060100-generic	0x810100b
Zen+	AMD Ryzen 5 3550H	22.04.1 LTS	5.15.0-143-generic	0x8108102
Zen 2	AMD EPYC 7252	22.04 LTS	6.1.0-sev-es	0x830107c
Zen 4	AMD Ryzen 9 7940HS	22.04.4 LTS	5.15.0-138-generic	0xa704104
Zen 5	AMD Ryzen 9 9950X	24.04.2 LTS	6.1.0-060100-generic	0xb404006
Zen 5/5c	AMD Ryzen AI 9 HX 370	24.04.1 LTS	6.1.0-060100-generic	0xb204019
LuJiaZui	Zhaoxin KX-U6780A	22.04.3 LTS	5.15.0-143-generic	n/a

Appendix B. Keystroke Timing Attack

We describe a keystroke-timing attack using the FP-DSS vulnerability (cf. Section 6.4). Instead of relying on the leakage, we use the presence of leakage as a side channel, converting the TEA into a side-channel attack as described by Weber et al. [55].

Threat Model. We assume an attacker with unprivileged code execution. In line with previous keystroke-timing attacks [92], the attacker wants to obtain the timestamps when a key is pressed but does not get the concrete key. We assume that the attacker can run the code on the same physical core as the victim application. We target GUI applications on Ubuntu 22.04.5 LTS with KDE Plasma 5.24.7 as desktop manager.

Setup. The attacker constantly mounts FP-DSS. However, instead of recovering the leaked values, the attacker uses data-flow divergence testing (cf. Figure 2) to detect whether the architectural and microarchitectural return values differ. If this is the case, the attacker logs the timestamp. We verify that several GUI applications, including Firefox, Chrome, and Kate, trigger such leakage upon pressing a key in their respective GUI.

Evaluation. We run the attacker on one logical core on an AMD Ryzen 5 3550H (microcode 0x8108102), and Firefox

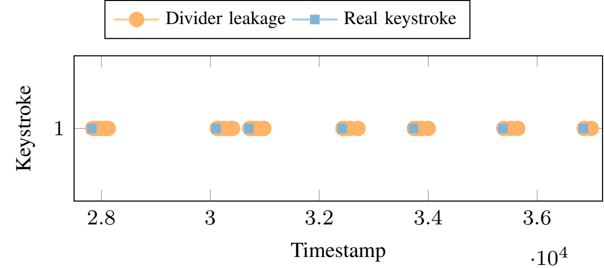


Figure 3: Timestamps of keystrokes by the user. Blue squares are the ground truth obtained from the operating system, orange circles are the timestamps where we see FP-DSS leakage on the sibling logical core.

on the sibling logical core. As a ground truth, we detect keystrokes by monitoring `/dev/input/event*` as root user. Figure 3 shows a plot with the timestamps (as CPU cycles) on the x-axis. For each real keystroke, we detect divider leakage, with multiple values being leaked.

Appendix C. Data-Dependency Algorithm

The pseudocode of TREVEX's data-dependency algorithm is shown in Listing 1. The algorithm is designed to decide whether the taint that is used inside the victim context influences the leakage observed by the attacker. For doing so, the algorithm first executes the test case for each taint and collects the observed leakage in a map, where each taint maps to its respective leakage (`leakage_for_taint`). Then, the algorithm removes noise from each leakage map (`RemoveNoise`). Afterwards, the algorithm removes every leaked value that is contained in multiple leakage maps, i.e., observed for multiple taints (`SymmetricDifference`). Finally, the algorithm checks whether each taint can be uniquely identified in the remaining leakage and decides whether data dependency exists based on the given accuracy ratio (`AccRatio`).

```
1  func RemoveNoise(leakage) {
2    for entry in leakage:
3      idx, val = entry
4      if val < NoiseThreshold:
5        leakage.delete(entry)
6    return leakage
7  }
8
9  func SymmetricDifference(leakage_for_taint,
10     Taints) {
11   for t in Taints:
12     leakage = leakage_for_taint[t]
13     // all maps but the current one
14     other_maps = leakage_for_taint - leakage
15     for idx in leakage:
16       // check if idx is already in another
       leakage map
17       for other_leakage in other_maps:
18         for other_idx in other_leakage:
19           if other_idx == idx:
20             // already exists in other map,
21             // thus, not unique
22             leakage.remove(idx)
23     // update the map
24     leakage_for_taint[t] = leakage
25   // return the map containing
26   // only unique entries
27   return leakage_for_taint
28 }
29
30 func CheckForDependency(Taints, NoiseThreshold,
     AccRatio) {
31   leakage_for_taint = EmptyMap()
32   for t in Taints:
33     // returns list of tuples constaining
34     // (encoded-value, number-of-observations)
35     leakage = ExecuteTestCaseForTaint(t)
36     leakage_for_taint[t] = RemoveNoise(leakage)
37
38   // remove every leaked entry that is
39   // not unique to a taint
40   unique_peaks = SymmetricDifference(
     leakage_for_taint)
41
42   dependent_leakage = 0
43   for t in Taints:
44     for entry in leakage_for_taint[t]:
45       if entry in unique_peaks:
46         // taint can be uniquely identified
47         // in leakage
48         dependent_leakage += 1
49         // do not check further for this taint
50         break
51   // return success if enough taints are
52   // uniquely identifiable
53   return dependent_leakage > len(Taints) *
     AccRatio
54 }
```

Listing 1: TREVEX's Data-Dependency Algorithm. The algorithm takes as input a set of taints and decides whether parts of the leakage observed by the attacker depend on the initial taints.

Appendix D.
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.

D.1. Summary

This paper presents a post-silicon black-box fuzzer for automatically finding data-flow transient execution vulnerabilities (Meltdown-style), without needing access to RTL, ISA reference model, or known attacks. The approach works by detecting mismatches between architectural and transient data flow, using instruction shadowing to stress shared microarchitectural state, a divergence-based test to identify unintended transient behavior, and a cross-privilege analysis to determine whether leakage originates from a victim context. The framework was evaluated on 20 Intel, AMD, and Zhaoxin x86 microarchitectures. It rediscovers known issues and finds new ones (FP-DSS, new FPVI variants, and new Zero-at-Ret cases).

D.2. Scientific Contributions

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

D.3. Reasons for Acceptance

1) The paper creates a new tool to enable future science. The framework is a way to discover vulnerabilities in a more systematic way, is easily portable across different microarchitectures, and could be a valuable tool for testing CPU designs for vulnerabilities in the future.
2) The paper identifies new vulnerabilities and variants, most notably FP-DSS on AMD CPUs, with convincing exploit demonstrations across privilege boundaries and even from browser-based attackers.
3) The paper addresses a gap in the field of testing tools for discovering new transient-execution vulnerabilities: the lack of generic, automated techniques that do not rely on RTL access, ISA semantics, or mutation of known attacks. By demonstrating that such detection is feasible at scale on real, commercial CPUs, the work advances the state of the art in CPU vulnerability discovery.

D.4. Noteworthy Concerns

1) As acknowledged by the authors, the code generation strategy is limited and does not autonomously find training-based DF-TEV, nor DF-TEV requiring complex code patterns like SSB. Developing more sophisticated code generation strategies is an interesting future work.
2) Evaluation is confined to x86 CPUs. While this is understandable and does not weaken the core results, it leaves open questions about how well the approach would transfer to other architectures such as ARM or RISC-V.