SKYFALL: Microarchitectural Data Sampling on AMD

Benedict Schlüter* *ETH Zurich*
Alessandro Vezzù* *ETH Zurich*
Daniel Moghimi *Google*
Shweta Shinde *ETH Zurich*

***Abstract*—While microarchitectural data sampling (MDS) vulnerabilities have been extensively studied on Intel CPUs, their exploitability and impact on AMD CPUs remain unexplored.**

In this work, we revisit Transient Scheduler Attacks (TSA) on AMD CPUs. We systematically analyze TSA to build a novel Microarchitectural Data Sampling (MDS) style exploitation technique that leaks data from internal microarchitectural buffers. Then, we leverage Instruction Based Sampling (IBS), an overlooked AMD performance monitoring feature, to reverse-engineer privileged CPU instructions, exposing critical design flaws in existing TSA mitigations. We combine the above observations to develop the SKYFALL attack, which exploits these flaws and successfully bypasses physical address space isolation boundaries on AMD CPUs. Specifically, we show that SKYFALL can be used for three high-impact attacks: (1) leaking arbitrary register state from Confidential VMs, (2) capturing data in-transit on the proprietary HyperTransport interconnect to extract encrypted microcode patches, and (3) leaking data residing in protected System Management Mode RAM.

We responsibly disclosed our findings to AMD, prompting the release of new microcode updates. We hope our methodology and test vectors will help secure future CPU architectures against transient execution attacks.

1. Introduction

Over the past decade, particularly since the discovery of transient execution vulnerabilities like Meltdown and Spectre, there has been a growing effort to identify security flaws in complex superscalar CPUs that expose user data to adversaries [1], [2], [3], [4], [5], [6], [7], [8], [9]. Specifically, transient execution attacks exploit subtle footprints of user data within the CPU pipeline, which are then exposed via microarchitectural side channels such as cache access patterns [1], [8], [10]. Consequently, this microarchitectural exposure can lead to cross-domain data leakage, violating user-kernel, process, or virtualization boundaries, the very cornerstones of modern system security.

Confidential computing technologies that rely on trusted execution environments (TEEs), such as confidential virtual machines (CVMs) [11], [12], further exacerbate the impact of transient execution attacks. These technologies rely on hardware-enforced access controls and strict privilege isolation to ensure that the host system software, including the hypervisor, cannot access tenant data. However, these security boundaries are ensured only at the architectural level. As demonstrated by previous CPU vulnerabilities affecting commodity TEEs like Intel SGX [2], [13], microarchitectural flaws may completely bypass these architectural isolation barriers. Remarkably, the specific threat model of confidential computing grants the host adversary unprecedented capabilities to execute highly precise transient execution attacks [14], [15]. For instance, an untrusted host controls physical address mappings, dictates the scheduling of victim workloads, and retains access to privileged system interfaces like performance monitoring units, all of which can be leveraged to orchestrate and amplify microarchitectural exploits.

Microarchitectural Data Sampling (MDS) [2], [3], [4], [5], [6], [7] represents a specialized class of transient execution attacks that specifically targets internal CPU structures. To maximize instruction throughput and memory bandwidth, modern out-of-order, speculative processors rely on various internal SRAM and Flip-Flop structures, often referred to as microarchitectural buffers, to temporarily store data, metadata, and address translations. MDS exploits these structures by leveraging Meltdown-style transient execution to extract stale data directly from buffers such as the store buffer [4], load buffer [3], line fill buffer [2], [3], and register files [6]. Due to how the speculative memory subsystem handles memory operations, these vulnerabilities have historically exposed a broad attack surface, resulting in a series of variants that compromise nearly every internal buffer across CPU generations dating back to the Intel Skylake microarchitecture.

For a long time, the academic study of MDS focused almost exclusively on Intel microarchitectures, as none of the initial MDS disclosures affected AMD processors. This landscape began to shift in 2021 when AMD disclosed Predictive Store Forwarding (PSF) [16], a feature susceptible to store buffer leakage similar to Microarchitectural Store Buffer Data Sampling (MSBDS) [4], though triggered by speculative mispredictions rather than architectural faults. More recently, the development of a new CPU vulnerability fuzzer [17] has led to the discovery of new transient execution vulnerabilities, called Transient Scheduler Attack (TSA), on AMD CPUs. Following a deeper analysis of TSA, AMD's internal assessment team identified two variants: TSA-L1 and TSA-SQ. These leak data from the L1D cache and store buffer, respectively, drawing clear parallels to Intel's L1 Terminal Fault (L1TF) [13] and MSBDS [4].

* denotes equal contribution

Despite these discoveries, independent, in-depth analyses of these AMD vulnerabilities' mechanics, exploitability, and systemic impact remain scarce. Official vendor documentation often abstracts the necessary exploitation criteria under vague descriptions like "complex microarchitectural conditions" [18]. This knowledge gap prevents security practitioners from independently verifying security fixes or constructing robust testing methodologies to prevent similar flaws in future CPU design. Systematically analyzing how these vulnerabilities behave in practice, specifically within confidential computing environments, provides the necessary foundation for validating mitigations, constructing new tests, and guiding future secure CPU design.

In this paper, we demonstrate critical shortcomings in the efficacy of AMD's proposed mitigations against TSA. To enable systematic evaluation, we construct two proof-of-concept (PoC) exploits targeting the distinct TSA sub-variants, providing a robust framework for regression testing and deeper systemic analysis of this vulnerability class. Then, leveraging our PoC, we expose deficiencies in AMD's TSA threat assessment.

Our investigation into the cache way prediction mechanism underlying TSA-L1 reveals an undocumented microarchitectural behavior: contrary to official vendor specifications, AMD computes the way prediction hash, known as the μtag, based on either virtual or physical address components. Furthermore, we demonstrate that linear and physical μtags are susceptible to collisions, effectively allowing an attacker to exploit TSA-L1 to speculatively exfiltrate data indexed by the physical μtag. To achieve granular visibility into the x86 instruction execution, we repurpose Instruction Based Sampling (IBS), a μop-precise hardware performance monitoring facility on AMD processors, to retrace the execution of individual μops within complex x86 instructions.

Consequently, we reverse-engineer the microcode updates intended to mitigate TSA, ultimately identifying execution paths where the hardware defenses are absent. By exploiting this, we show that a host adversary can extract data across three privilege boundaries. Concretely, we demonstrate that privileged software can use TSA to extract SEV-SNP Confidential Virtual Machine (CVM) register states, arbitrary CVM memory at specific offsets, decrypted microcode, and System Management Mode (SMM) memory.

In summary, we make the following contributions:

- **Bit-level Data Sampling:** We introduce a data extraction methodology that leverages Performance Monitoring Counters (PMCs) to measure scheduler contention. Then, we reverse-engineer the undocumented L1D way-predictor micro-tag (μtag) hash functions across Zen 3-5, enabling precise bit-level recovery of speculative data from the L1D cache and store queue on Zen 3 and Zen 4.
- **Mitigation Gaps:** We reverse-engineer AMD's official `VERW`-based TSA mitigations using Instruction Based Sampling (IBS). We reveal that while the mitigations address virtual-address-based leaks, physical-address-based and CPU-internal memory paths (such as those triggered by privileged instructions like `VMSAVE`, `VMLOAD`, and `VMRUN`) continue to populate way-predictor state, leaving the CPU vulnerable.
- **Security Impact:** We design and implement SKYFALL, an end-to-end transient execution attack exploiting physical address way-prediction, and demonstrate the severity of this attack by successfully extracting data from an AMD SEV-SNP guest, leaking SMM RAM, and extracting decrypted microcode patches.

Responsible Disclosure We reported our findings concerning SEV-SNP to AMD on 21 April 2026; subsequently, we sent a follow-up about the impact on SMM and HyperTransport on 28 May 2026. AMD acknowledged the findings and issued CVE-2026-40671.

Open Science Our code is public at github.com/skyfall-attack.

2. Background

In this section, we provide the necessary background information, covering AMD Zen pipeline behavior, cache hierarchy and cache way prediction, AMD virtualization and SEV-SNP, and special AMD CPU features that are relevant.

2.1. Pipeline and Micro-Operations

AMD Zen processors are based on the x86 ISA that translates architectural instructions, also called macro-ops, into simpler internal micro-operations (μops). Simple instructions are decoded directly into one or more μops, whereas complex instructions may invoke the Microcode Sequencer Read-Only Memory (MSROM), which emits longer μop sequences from internal microcode storage. Processors also contain microcode patch RAM and associated patch state, which allow vendor-supplied updates to redirect selected microcode flows to patched implementations after deployment.

Decoded μops enter the out-of-order backend, where they may execute as soon as their operands are ready and a suitable execution unit is available, even if older independent μops are still waiting. The processor preserves the architectural illusion of in-order execution by committing instruction results in program order. This separation between execution and retirement allows transient work to proceed before the CPU fully resolves all older operations.

Before execution, μops wait in scheduler structures that track operand readiness and issue ready operations to the backend. The backend contains several classes of execution units, including integer units, branch units, floating-point and vector units, and load/store units. Each unit class exposes only a limited number of execution resources. When more ready μops require a resource than the backend can issue in the same cycle, the remaining μops must wait. This condition is known as execution-unit or port contention.

2.2. Cache Hierarchy and L1D Way Prediction

AMD Zen processors feature a private L1D cache and L2 cache per core, with a shared Last Level Cache (LLC) across cores in a Core Complex. Cache lines are 64 bytes wide, and an access hits when the requested line is present in the addressed cache set. The L1D is Virtually Indexed Physically Tagged (VIPT), meaning that the cache set can be selected using virtual-address bits before physical address translation completes. The physical tag is then used to determine whether the selected cache entry actually corresponds to the requested physical line [19].

To avoid comparing tags across all ways on every access, AMD processors employ an L1D way predictor that selects one cache way early in the load path [19]. This selection is based on a micro-tag (μtag), a short hash derived from the linear address. Because the μtag depends only on the linear address, the CPU computes it before the Page Walker resolves the corresponding physical address. Each L1D line is associated with the μtag of the linear address that last accessed it. On a load, the predictor uses the incoming μtag to select the matching cache way, allowing the processor to read a single predicted way rather than all ways and thereby reducing power consumption. If the prediction is wrong, whether by falsely predicting a hit or missing an actual hit, the processor initiates a fill request to the L2 cache or to a lower cache level and updates the μtag when the fill completes.

Because the CPU derives the μtag from a subset of the linear address, different linear addresses can collide in the prediction structure. This includes linear aliases, where distinct virtual addresses map to the same physical address, as well as unrelated addresses that happen to share the same μtag hash at the same L1D index. At any given L1D index, only one cache line per linear hash is accessible through the μtag at a time. The CPU marks old conflicting lines as invalid and therefore effectively evicts them from the L1D. Importantly, the CPU tags the μtag with the thread ID.

A second form of early memory disambiguation occurs in the load-store unit through store-to-load forwarding (STLF). STLF allows a younger load to receive data directly from an older in-flight store while the store is still buffered in the store queue, provided that the store contains all bytes required by the load. On Zen, forwarding eligibility is determined using the lower 12 bits of the linear address, so stores and loads with the same page offset can interact before the CPU resolves the full physical address [19].

2.3. AMD CPU Virtualization

Virtualization allows a single physical host to run multiple isolated Virtual Machines (VMs). A VM consists of one or more virtual CPUs (vCPUs), each representing a logical processor in the VM, managed by the hypervisor.

On AMD systems, a hardware structure called the Virtual Machine Control Block (VMCB) corresponds to each vCPU [20]. The VMCB contains the control state the processor uses when entering and exiting the guest. VMCB contains a pointer to a memory region called the Virtual Machine Save Area (VMSA) that contains general-purpose registers, control registers, instruction pointer state, flags, segment state, and other architectural state required to enter the guest. When exiting VM mode, the CPU saves the vCPU state in the VMSA.

To enter a guest, the hypervisor executes the x86 instruction `VMRUN` with the physical address of the VMCB. The processor then performs a sequence of consistency, permission, and state checks before switching into guest execution. AMD implements those checks in microcode, which involve multiple internal memory accesses to the VMCB, the VMSA, and related virtualization metadata. If one of the checks fails, `VMRUN` aborts and the processor does not enter guest mode.

2.4. AMD SEV-SNP

AMD offers confidential VMs based on the SEV-SNP architecture [12]. Before SEV-SNP, AMD introduced two earlier revisions of this technology, each improving upon its predecessor. The first revision, SEV, encrypts guest memory such that a malicious hypervisor cannot read SEV guest data, but does not encrypt guest register state. SEV Encrypted State (SEV-ES) addresses this shortcoming by extending encryption to guest CPU register state. SEV-SNP achieves integrity protection for guest memory through the Reverse Map Table (RMP). The RMP contains metadata for every page in the system. During memory accesses, the processor checks if the current access is allowed according to the policy in the RMP. Within the RMP, the processor marks VMSA pages that hold the encrypted guest state with a special bit. This ensures that the hypervisor cannot use arbitrary CVM pages as VMSA pages.

To decrypt encrypted data, the processor uses unused bits in the physical address space to propagate encryption and decryption information to the memory controller. Crucially, AMD prevents any software from setting these bits directly when issuing memory requests. Only trusted microcode possesses this capability, and when executing within guest mode, hardware will set these bits automatically. This also means that inside the processor, especially in the caches, SEV-SNP CVM data resides unencrypted.

2.5. HyperTransport

HyperTransport was originally developed as a point-to-point high-performance motherboard interconnect bus [21]. AMD used it in its FX CPU motherboard chips to connect the CPU with the Northbridge. With the latest AMD Zen architecture, AMD pivoted to a fundamentally different hardware design and does not physically use HyperTransport-branded interconnects. The HyperTransport interconnect was succeeded by the Infinity Fabric.

However, at the protocol level, the legacy technology continues as the backbone in AMD Infinity Fabric Interconnect. With the Zen CPU generation, AMD uses

the enhanced coherent HyperTransport+ protocol to handle inter-chip communication [22]. The protocol reserves a range in the physical address space. On Zen 4 the HyperTransport address range spans `0xFFFD00000000-0xFFFDFFFFFFFF` [23]. The CPU does not route physical addresses within that range to DRAM, as they have a special internal use. The CPU uses the HyperTransport address space when performing L3 cache flushes, handling `IN`/`OUT` instructions, or when accessing CPU-exclusive SRAM [24]. As the HyperTransport address space houses such sensitive operations and data, AMD restricts its access. When the CPU translates a virtual address to a HyperTransport address, the page walker blocks such translations and faults with a reserved bit set violation. To access the HyperTransport address space, despite the restrictions for kernel space, microcode uses a special addressing mode when executing move μops.

2.6. Instruction Based Sampling

IBS is an AMD performance monitoring feature that provides detailed information about individual sampled instruction executions. AMD distinguishes between IBS for execution sampling, which samples information about instruction execution, and IBS for fetch sampling, which samples information about instruction fetches. Initially intended as a statistical sampling tool, prior research identified that IBS can be used to precisely sample individual μops of instructions [20], [24].

The precise sampling works by setting an IBS counter within a Model Specific Register (MSR). When privileged software activates the counter by setting another bit in an MSR, the counter decrements for every executed μop that passes through the CPU frontend. Once the counter reaches zero, it tags the next μop about to be executed, such that the CPU samples information about its execution. Depending on the type of μop, the CPU samples different execution characteristics. For load and store operations, this includes the linear and physical address, cache hit or miss status, translation information, and latency measurement. For branch operations, IBS records whether the branch was taken and whether it was mispredicted. All other operations appear only as ALU operations and expose only limited metadata such as timing and retirement information. We define an IBS trace as the μop sequence of an x86 assembly instruction.

2.7. Transient Scheduler Attacks

Transient Scheduler Attacks (TSA) are speculative side-channel vulnerabilities on AMD processors. They include two variants, TSA-L1 and TSA-SQ. In TSA-L1, under a speculative condition, the μtag of an in-flight load can match the μtag of a line that is already resident in the L1D. The processor may then predict that the load can complete from this L1D entry and forward the corresponding data to dependent operations before translating the virtual address. TSA-SQ follows an analogous pattern in the store queue, where the processor may predict that a load can complete via store-to-load forwarding and provide data from the selected store queue entry. In both cases, if the CPU makes an incorrect prediction, the forwarded value is not architecturally committed, but dependent transient execution may already have consumed it.

Data served through such a transient path carries an internal marker that propagates to dependent operations. If a later load or store consumes marked data, the processor prevents the memory operation from fetching data or updating cache or TLB state. As a result, neither TSA-L1 nor TSA-SQ can use the Spectre- and Meltdown-style encoding pattern in which transient data selects a secret-dependent cache line. Instead, the signal must come from effects within the transient execution window, such as differences in execution-unit or scheduler contention.

3. Overview

We introduce the threat model and give a high-level overview of SKYFALL.

Threat Model. We assume a malicious kernel or hypervisor with standard privileged access to performance monitoring counters, physical address mappings, and VM-control interfaces. This matches the confidential-computing threat model, where the hypervisor is untrusted but still manages guest scheduling, nested page tables, and VMCB state. The attacker cannot directly read encrypted CVM memory, encrypted VMSA state, SMM RAM, or HyperTransport-addressed CPU-internal state.

SKYFALL Attack. While TSA has been discovered by academic researchers, the impact analysis was mostly performed by AMD. However, their public analysis lacks technical details and concrete code examples [18]. This hinders researchers from studying TSA and performing independent analysis.

In SKYFALL, we develop the first public proof-of-concept for TSA vulnerabilities using Performance Monitoring Counters (PMCs) as a new technique to measure speculative scheduler contention. Concretely, we develop a Proof-of-Concept (PoC) for TSA-SQ and TSA-L1, where we analyze the microarchitectural conditions for successful data exfiltration (Section 4). To validate the effectiveness of TSA mitigations, we use IBS to analyze the microarchitectural mitigations AMD deployed. Our analysis shows that AMD overwrites the μtag in the L1D to prevent TSA-L1, which most likely also overwrites the store queue slots, mitigating TSA-SQ (Section 5.1). We further study the cache way-prediction behavior when the μtag of a virtual address collides with the μtag of a physical address (Section 5.2). Our results contradict AMD documentation, which only states that virtual addresses have a μtag [19]. Based on this observation, we show that we can extract L1D cache lines associated only with a physical address by virtual address-based μtag collision (Section 5.3).

Based on the insights into the mitigation and the broader scope of TSA beyond virtual address-based accesses, we

analyze SEV-SNP-related instructions by inspecting their μops. Our analysis shows that AMD does not protect all execution paths, which allows us to break SEV-SNP confidentiality (Section 6.1). We show that we can leak other data that is exclusively indexed by a physical address. We leak microcode loader state, concretely, decrypted microcode patches (Section 6.2), and extend the analysis to the System Management Mode, where we present the first MDS-style attack targeting SMM (Section 6.3) [25].

4. Microarchitectural Data Sampling

In this section, we first discuss the difficulty in reproducing TSA and how we solve it. We then build a bit-level data sampling methodology and provide a working TSA-SQ exploit. We extend our analysis to TSA-L1 by reverse-engineering the μtag function to obtain collision addresses. Subsequently, we use μtags in an extraction gadget to leak data from the L1D cache, which we use as the main attack vector throughout the rest of the paper.

4.1. Transient Scheduler Attacks

At the time of writing, no public TSA reproducer can leak memory. The absence of such a reproducer stems from the difficulty of reproducing TSA and is twofold. First, one has to find an architectural mechanism that actually causes speculative scheduler contention, and second, one has to find a way to reliably measure said contention. Because these two observations are interleaved, one must solve for both at a time, which significantly increases the search space. The next two paragraphs explain at a high level how we solve those difficulties.

Measuring Scheduler Contention. We develop a custom methodology to extract the TSA signal by combining AMD's sparse documentation with reverse engineering. While the Microsoft Research paper discovering TSA already provides a way to detect TSA leakage across privileges, it is not directly convertible into an exploit due to heavy noise and missing microarchitectural details [17]. Microsoft researchers measured the secret-dependent scheduler contention caused by TSA by measuring changes in the caches. This initially sounds contradictory to AMD's analysis, where they state that the value fetched under TSA does not interact with the cache / TLB [18]. However, Microsoft researchers used an indirect method of extracting the data speculatively using caches. Depending on the TSA secret value, the CPU executes more or fewer instructions speculatively. The secret thus controls the CPU scheduling by causing more or less scheduling contention. Under high contention, the CPU evaluates a fault that happens prior to the cache-modifying load faster; thus, the load will not execute, and the CPU will not fetch the cache line into the L1D. Under low contention, the cache line load is faster than the fault and will execute. We refer to this technique as indirect cache-based exfiltration, as the CPU does not perform the cache modification directly on the speculatively obtained value. This technique is fragile, as co-located SMT workloads influence the execution duration of the extraction gadget and introduce heavy noise. For this reason, it takes Revizor, the fuzzer that discovered TSA initially, around 5 hours to detect statistical leakage [26].

Instead of relying on an indirect cache-based exfiltration gadget, we use PMCs to measure the scheduler contention induced by TSA. This works because the CPU does not roll back PMCs when detecting a false completion, the root cause of TSA. Further, PMCs are substantially more robust to SMT contention, providing a substantially more stable extraction methodology, bringing the detection down to a few milliseconds. Section 4.2 and Section 8.1 contain more details regarding the PMC signal.

> **Observation 1:** Performance Monitoring Counters can reliably measure scheduler contention caused by speculative execution.

Gadget Analysis. We evaluate several leak gadgets to analyze which induce scheduler contention in the TSA setting. Of all the gadgets we analyze, only the one in Listing 1 shows TSA leakage.

```
asm volatile(
    [...]
    /* r9 carries secret speculatively  */
    "movq    %[memory_ptr], %%r10    \n\t"
    "addq    $56, %%r10              \n\t"
    ".rept   20                      \n\t"
    "movq    %%r10, %%rsi            \n\t"
    "addq    %%r9, %%rsi             \n\t"
    "lodsq                           \n\t"
    ".endr                           \n\t"
);
```

Listing 1. TSA Extraction Gadget.

Within the speculation window, the gadget either executes up to 20 cache line crossing load instructions or no cache line crossing loads. `lodsq` loads 8 bytes from `rsi` into `rax`. When the secret is 0, `rsi` has a page offset of 56, and when the secret is 1, it has a page offset of 57. Thus `lodsq` will only cross a cache line boundary when `r9` is equal to 1.

Using cache line crossing loads as a TSA gadget may seem counterintuitive, as AMD states that loads consuming a TSA speculative forwarded value will not interact with the cache or TLB state [18]. We deduce that the scheduler still forwards those load instructions depending on `r9` to the execution units. When these instructions execute, they are prohibited from interacting with the memory subsystem.

The other gadgets we evaluate consist of floating-point and integer division, whose latency depends on the operands [27]. By placing these instructions in a dependency chain on `r9`, we aim to amplify the transient bit into a long or short backend stall. We further evaluate branch-based gadgets, vector gathers where the transient bit controls the mask and therefore the number of loaded elements, and REP-prefixed sequences where the bit controls the iteration count. None of these alternatives yield results on any of our

platforms when doing a full PMC sweep. The unsuccessful gadgets indicate that the CPU does not expose the speculatively completed value in `r9` to instructions with data-dependent latency.

Observation 2: With TSA, only simple memory load instructions cause secret-dependent speculative scheduler contention. Other gadgets, such as divisions, branches, and scatter-gather instructions, do not induce measurable scheduler contention.

4.2. TSA-SQ

We exploit TSA-SQ by creating a speculative store-to-load forwarding condition by choosing store and load addresses with identical lower 12 address bits, forming a 4 KiB alias. Listing 2 shows a TSA-SQ snippet. Architecturally, `r9` will either be 0 or 1 in Line 9, depending on which bit of `target` we select.

```
1  asm volatile(
2      "movb %b[bit], %%cl\n\t"
3      /* Put secret into store queue. */
4      "movq %[secret], (%[victim])\n\t"
5      /* Load from target; target must have
6      the same low 12 bits as victim */
7      "movq (%[target]), %%r9\n\t"
8      "shrq %%cl, %%r9\n\t"
9      "andq $1, %%r9\n\t"
10
11     /* cause speculative scheduler
12        contention based on r9  */
13     "movq    %[memory_ptr], %%r10    \n\t"
14     "addq    $56, %%r10              \n\t"
15     ".rept   20                      \n\t"
16     "movq    %%r10, %%rsi            \n\t"
17     "addq    %%r9, %%rsi             \n\t"
18     "lodsq                           \n\t"
19     ".endr                           \n\t"
20 );
```

Listing 2. TSA-SQ Proof-of-Concept.

In the speculative domain, the load in Line 7 executes with the previously stored value, `secret`. It seems that a straightforward way to exfiltrate the value is to use a cache line load depending on the value of `r9`. This would be in line with prior store-to-load forwarding vulnerability [16]. However, our experiments show that this does not induce any leakage. More precisely, an `r9`-dependent memory load following Line 9 does not influence the cache state in the speculative domain. This follows AMD's TSA analysis, where they state that a TSA speculative forwarded value does not influence cache or TLB state [18]. However, AMD also states that such speculative loads may influence the scheduling of other instructions. Our experiments show that PMCs combined with the extraction gadget in Listing 2 reveal TSA leakage. We elaborate on how we use the leakage gadget for a full TSA-SQ PoC and which microarchitectural conditions need to be satisfied.

Proof of Concept. For each PMC, we perform a probing phase, where we sample the PMC parameters of the code in Listing 2 for the case where the CPU speculatively forwards a 1 or 0 in `r9`. This means `secret` in the probing phase either consists only of ones or zeroes. While the speculatively executed loads (Lines 15- 19) do not interact with the cache, they cause scheduler contention [18] that some PMCs collect. We use Otsu's method to find a threshold that separates the PMC values in the two classes [28]. In the last step, we aim to leak a 32-bit secret by collecting the PMCs for each bit and, based on the threshold, assigning the corresponding value.

TABLE 1. MEDIAN NUMBER OF USABLE PMCS FOR TSA-L1 AND TSA-SQ EXTRACTION WITH DIFFERENT NUMBERS OF SAMPLES PER BIT ACROSS 20 RUNS. VALUES ARE REPORTED AS MEDIAN ± STANDARD DEVIATION. FOR EACH CONFIGURATION, WE EVALUATE EXTRACTION ACCURACY ON A 32-BIT SECRET AND COUNT A PMC AS USABLE IF IT EXCEEDS 87.5% ACCURACY.

	EPYC 9124 (Zen 4)		EPYC 7313 (Zen 3)	
Samples	TSA-L1	TSA-SQ	TSA-L1	TSA-SQ
1	1927 ± 402	3061 ± 84	2348 ± 146	2855 ± 51
2	2474 ± 239	3026 ± 65	2370 ± 120	2888 ± 419
4	3022 ± 73	2918 ± 71	2379 ± 97	2904 ± 26
8	3051 ± 89	2992 ± 35	2394 ± 66	2904 ± 27
16	3059 ± 100	2990 ± 56	2397 ± 69	2904 ± 52
32	3089 ± 95	2961 ± 58	2400 ± 73	2902 ± 56
50	3054 ± 89	2937 ± 56	2399 ± 72	2900 ± 53
64	3063 ± 88	2950 ± 78	2400 ± 73	2886 ± 51
100	3072 ± 92	2940 ± 68	2399 ± 74	2854 ± 34
128	3029 ± 92	2947 ± 89	2394 ± 88	2854 ± 24
256	2973 ± 75	2933 ± 80	2391 ± 101	2790 ± 55
512	2876 ± 60	2866 ± 128	2386 ± 9	2848 ± 63

Table 1 shows the median number of usable PMCs identified for the TSA-SQ gadget in Listing 2 across different numbers of samples per bit. To rule out Predictive Store Forwarding (PSF) as the source of our leakage, we disabled PSF via the SSBD and PSFD bits in MSR `48h` [16]. The signal remains observable. For the exploit in Listing 2 to work, `target` and `victim` must have an active TLB entry and should share the same 12 lower address bits. If the conditions are not met, we do not observe any signal.

4.3. TSA-L1

Similarly to the TSA-SQ, we develop a TSA-L1 PoC. Specifically, we analyze which microarchitectural conditions need to be fulfilled for the transient forwarding to occur.

Reversing the μTag. To build a TSA-L1 proof of concept, we first need to construct virtual addresses that collide in the L1D way-prediction structure [18]. AMD uses a virtual address-based hash, called the μtag, to optimize cache way prediction. Prior work studies the μtag hash for Zen 1-2 [29]. We extend this approach to newer AMD EPYC chips and recover the μtag function across Zen 3, Zen 4, and Zen 5.

According to AMD documentation, the μtag is unique for every L1D index [30]. In other words, if two cache lines have the same index and μtag, they will evict each other from the L1D to the L2. This latency difference between an L1D and L2 access provides an empirical collision test.

To find the μtag hash function, we use two pointers that chase each other. One pointer resembles a μtag of zero and another pointer whose virtual address we vary. Should both pointers have a conflicting μtag, the pointer chasing results in a higher average access latency since each load must come from the L2 rather than the L1D for non-conflicting μtags.

We use this approach to first validate which bits in the virtual address contribute to the μtag. Our experiments show AMD uses bits [12 - 27] for Zen 3 and Zen 4 and bits [12 - 31] for Zen 5 for the μtag computation. Next, we use this information to vary the active bits and determine which combinations cause a conflict with the zero μtag. Should we encounter a conflict, we save the corresponding vector of set bits in the virtual address. When we have x linearly independent vectors, where x denotes the number of μtag bits, we can solve for the μtag hash function. We may scan the entire set of x active bits in $\mathbb{F}_2^{64}$ without finding x independent vectors. This means that some bits are not XORed with other bits. These are precisely the vector bits that occur as stand-alones in the μtag hash function. Appendix A contains a mathematical description of how we recovered the μtag. Equation 1 shows the resulting μtag for Zen 3 and Zen 4. It is the same hash function AMD uses for Zen+ and Zen 2 [29]. Zen 5, however, uses a different hash function, shown in Equation 2.

$$u_i = \begin{cases} v_{i+12} \oplus v_{27-i} & \text{if } 0 \leq i \leq 2 \\ v_{22-i} \oplus v_{27-i} & \text{if } 3 \leq i \leq 7 \end{cases} \quad U = \sum_{i=0}^{7} u_i \cdot 2^i \quad (1)$$

$$u_i = \begin{cases} v_{31-i} & \text{if } 0 \leq i \leq 3 \\ v_{i+8} \oplus v_{31-i} & \text{if } 4 \leq i \leq 6 \\ v_{26-i} \oplus v_{31-i} & \text{if } 7 \leq i \leq 11 \end{cases} \quad U = \sum_{i=0}^{11} u_i \cdot 2^i \quad (2)$$

Appendix A contains additional details of how we obtain the μtag formulas.

Triggering transient forwarding. When we modify our TSA-SQ PoC such that instead of colliding with a previous store, we make `target` collide with a μtag in the L1D, we expect measurable scheduler contention. However, when we test this setting in Listing 3, we do not observe any leakage via the PMCs despite the load from `target` colliding with the victim address in the L1D way predictor.

```
1  asm volatile(
2      [...]
3      "movb    %b[bit_idx], %%cl       \n\t"
4      "movq    (%[target]), %%r9       \n\t"
5      "shrq    %%cl, %%r9              \n\t"
6      "andq    $1, %%r9                \n\t"
7
8      "movq    %[memory_ptr], %%r10    \n\t"
9      "addq    $56, %%r10              \n\t"
10     ".rept   20                      \n\t"
11     "movq    %%r10, %%rsi            \n\t"
12     "addq    %%r9, %%rsi             \n\t"
13     "lodsq                           \n\t"
14     ".endr                           \n\t"
15     [...]
16 );
```

Listing 3. TSA-L1 Proof-of-Concept.

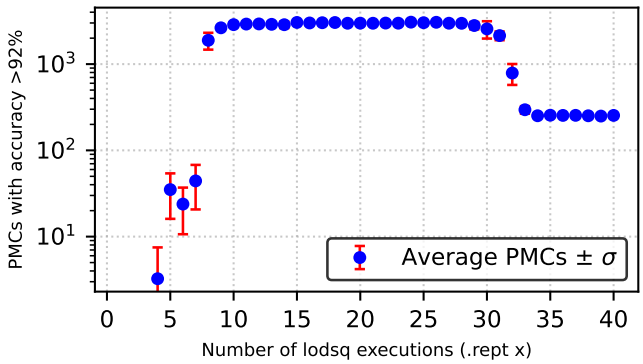


Figure 1. Number of working PMCs (accuracy over 92% for 64-bit secret) when varying the number of the rept prefix in Listing 3, Line 10 over 10 runs.

We hypothesize that the CPU does not speculate if `target` already has a TLB entry. In this case, the CPU directly sees that the L1D entry is incorrect. Only when there is no TLB entry, meaning the physical address is not known, does the CPU use the μtag to speculate that the load will hit the virtually-indexed physically-tagged L1D.

```
1 asm volatile(
2     "movq    (%[target]), %%rax \n\t"
3     "invlpg  (%[target])        \n\t"
4     "movq    (%[victim]), %%rax \n\t"
5 );
```

Listing 4. TSA-L1 Leak Requirements.

To confirm our hypothesis, we insert the code from Listing 4 before our actual exploit code in Listing 3. Importantly, Line 3 in Listing 4 flushes the TLB entry of `target`. Combining and executing both code snippets yields a working TSA-L1 PoC. Table 1 reports how many usable PMCs we find for different numbers of samples per bit. We further evaluate how the number of load instructions influences the speculative leakage with TSA-L1, which is depicted in Figure 1.

Observation 3: The CPU only performs TSA-L1 speculation if two addresses have a conflicting μtag and the conflicting μtag load does not have a TLB entry.

Other conditions. Importantly, the architectural execution of the code in Listing 3 should be slow, which means the architectural load of `target` should be 1. This is to ensure we have a massive slow path if both the architectural and speculative load are 1.

If the architectural load is 0, the signal becomes substantially noisier and yields roughly 85% fewer usable PMCs for extraction. Furthermore, to avoid the PMCs measuring memory-hierarchy latency, we first bring `target` into the L2 cache via a μtag collision, as shown in Listing 4

5. SKYFALL Primitives

As AMD provides only limited technical documentation for its TSA mitigations, we explore the underlying mechanisms. To reverse engineer AMD's mitigation behavior, we use IBS and compare the recommended mitigations against cache-flushing instructions. We then show that the μtag collision can be extended to physical address-based accesses.

5.1. Reverse Engineering AMD TSA Mitigations

We reverse-engineer the TSA mitigations by using IBS. IBS allows us to inspect individual μops of complex instructions. By comparing an IBS trace of the TSA mitigation instruction `VERW` before and after AMD changed the microcode, we can deduce which μops AMD inserted. AMD recommends only using the x86 instruction `VERW` to mitigate TSA and specifically excludes cache flushing instructions.

VERW Instruction. `VERW` checks if a data segment specified by the segment selector is writable. Following Intel RIDL and related MDS attacks, Intel repurposed `VERW` as a mitigation primitive, where updated microcode causes `VERW` to trigger internal buffer-clearing operations that remove stale transiently accessible data [31]. AMD recommends executing `VERW` on privilege transitions from kernel to userspace, or from hypervisor to VM to mitigate TSA.

We observe that the IBS trace of `VERW` does not resemble an L1D flush and differs clearly from an IBS trace of cache-line flush instructions such as `CLFLUSH`. On systems vulnerable to TSA-L1, we observe 512 one-byte stores to CPU-internal addresses. Based on the L1D size of 32 KiB and a cache line size of 64 bytes, we hypothesize that these 512 stores overwrite their matching 512 L1D cache lines and the one-byte μtag used by the way predictor [19]. Listing 5 shows one out of the 512 overwriting sequences in the IBS trace. We see that the CPU writes to physical address `0x60`. As the CPU has around 32 addressing modes, it likely selects a physical address space that belongs to one of its internal control units [24].

```
[...]
[ALU]                  CompToRet[4] TagToRet[10]
[ALU]                  CompToRet[3] TagToRet[10]
[store1] p=0x60 v=0x0 CompToRet[1] TagToRet[ 9]
[...]
```

Listing 5. Partial `VERW` IBS trace that overwrites the μtag in the L1D.

After execution of `VERW`, the next memory access consistently misses the L1D cache but hits the L2 cache, suggesting that the CPU does not physically evict the data from the private cache hierarchy.

Virtualization Boundaries. According to AMD, TSA-L1 impacts AMD SEV-SNP [18]. In particular, TSA-L1 allows a malicious hypervisor to infer data an SEV-SNP CVM accesses. AMD states that its microcode update automatically applies the mitigation for SEV-SNP, such that an attacker cannot use TSA-L1 to leak CVM data. Our IBS trace confirms that microcode overwrites the L1D μtags, similar to `VERW`, whenever a CVM exits.

TABLE 2. EFFECTIVENESS OF CANDIDATE MITIGATIONS AGAINST TSA-L1. A MITIGATION IS CONSIDERED INEFFECTIVE IF WE FIND A VALID PMC CONFIGURATION THAT EXTRACTS A 32-BIT SECRET WITH MORE THAN 87.5% ACCURACY.

CPU	VERW	WBINVD	CLFLUSH	WRMSR L1D_FLUSH
Zen 3	✓	✗	✗	–
Zen 4	✓	✗	✗	✗

Cache Flushing Instructions. AMD recommends `VERW` as a TSA mitigation and notes that cache-flushing mechanisms such as `WBINVD` and `CLFLUSH` may not be effective against TSA-L1 [18]. We experimentally adjust our PoC and insert `CLFLUSH` or `WBINVD` after bringing the victim data into the L1D. Our results experimentally confirm that these instructions do not mitigate TSA-L1. Only `VERW` prevents extraction with our TSA-L1 gadget, as shown in Table 2.

Observation 4: `VERW` mitigates TSA-L1 by overwriting the μtag in the L1D cache.

5.2. Physical μtag

AMD derives the L1D μtag from the virtual address [19], [30]. This suggests that TSA-L1 only applies to data accessed through ordinary virtual-address-based loads, since the attack relies on constructing a virtual address with a colliding μtag. However, several privileged AMD instructions (e.g., `VMRUN`, `VMLOAD`, and `VMSAVE`) operate directly on physical addresses. IBS traces of these instructions show that these instructions perform memory operations using physical addresses, while IBS marks the virtual address as zero and invalid. This led us to the hypothesis that, for physical-address-based accesses, the processor may compute the μtag from the physical address instead of the virtual address.

To validate our hypothesis, we use `VMSAVE` as a controlled physical-address access primitive. `VMSAVE` writes host state to the physical address provided by its operand, allowing us to trigger a known physical-address-based memory access without using an ordinary load or store. We allocate three physical pages. We use the first page as the `VMSAVE` target. Further, we use the other two pages as probe pages and map them at manually chosen virtual addresses. For one probe page, we choose a virtual address whose computed μtag matches the μtag computed over the physical address passed to `VMSAVE`. For the second probe page, we choose a virtual address with a different μtag. Before each trial, we bring the probe line into the L1D cache, execute `VMSAVE` on the target physical page, and then test whether the probe line is still an L1D hit.

For the matching-μtag address, `VMSAVE` consistently displaces the probe line from the L1D cache. For the non-matching-μtag address, the probe line remains L1D-resident

in all trials. This behavior indicates that the physical-address-based writes VMSAVE performs interact with the L1D way-prediction state selected by the μtag.

Observation 5: The CPU computes the μtag on physical-address-based memory accesses.

5.3. Physical Address Way-Prediction

Observation 5 suggests that physical-address-based loads may be extractable using TSA-L1. To validate this hypothesis, we use VMSAVE to force physical-address-based memory accesses. Since we control the register state, we choose the physical address used as the VMSAVE target. We then create a mapping for a virtual address VA whose μtag collides with the μtag of the physical address passed to VMSAVE, while VA itself maps to a different physical address. We execute VMSAVE and run the TSA-L1 extraction gadget from Section 4.3, using VA as the extraction address. The extraction recovers the register values written by VMSAVE to its physical target address. This shows that TSA-L1 can extract L1D-resident data populated by physical-address-based memory accesses. More broadly, this result indicates that TSA-L1 can be used to extract arbitrary data from the L1D cache.

Observation 6: TSA-L1 can recover data that the CPU loads into the L1D through physical addressing.

6. SKYFALL Attack

Building on our observations from Section 5, we demonstrate that the AMD TSA mitigations are insufficient. Adversaries can still leak data across three privilege boundaries.

6.1. Breaking SEV-SNP Confidentiality

SKYFALL breaks SEV-SNP confidentiality by leaking register state and memory of SEV-SNP CVMs. We first discuss how we leak arbitrary register state. The hypervisor enters SEV-SNP CVMs using the VMRUN instruction. Upon execution, the processor switches execution modes to handle CVM code and data. If the hardware receives a physical interrupt or the CVM executes a special instruction, the processor leaves the CVM mode and returns control to the hypervisor. As discussed in Section 5.1, transitioning from CVM execution back to the host triggers the TSA mitigations.

Whenever the processor switches modes, it must save the current register state and load the new one. Specifically, during VMRUN (hypervisor to VM transition), the hardware saves the host state to the physical address specified in the VM_HSAVE_PA MSR and retrieves the CVM configuration from the encrypted VMSA. By inspecting the VMRUN IBS trace, we can retrace this VMSA loading process. The hardware obtains the VMSA physical address from the VMCB and uses it to restore the CVM context.

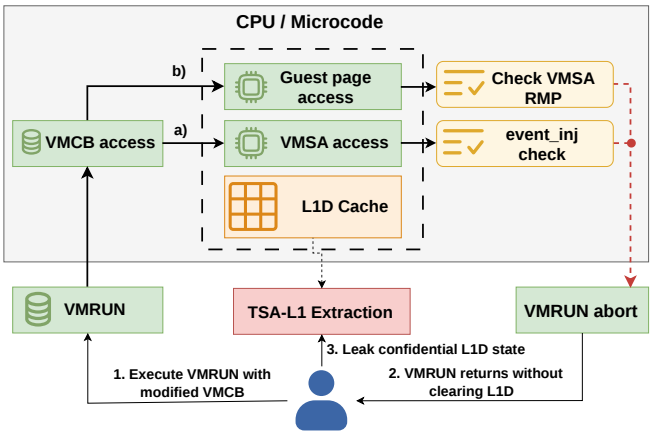


Figure 2. VMRUN abort path for extracting VMSA state and guest memory with TSA-L1. a) CVM VMSA register state leakage; b) CVM memory leakage.

Leaking VMSA state. Interestingly, VMSA loading occurs before other consistency checks complete. One such check relates to interrupt injection. If the hypervisor programs that field with a malformed value, VMRUN aborts. The malformed event-injection field therefore provides a controlled way to force an early VMRUN abort after the processor has already fetched and decrypted VMSA data. Crucially, this abort path bypasses the TSA mitigations usually triggered at CVM exit.

This microarchitectural behavior is insecure because the hardware has already loaded and decrypted the encrypted guest context. We observe that the processor keeps these values in the L1D cache. Concretely, we observe that an aborted VMRUN can load up to 3392 bytes of the VMSA into the L1D cache without applying the VERW mitigation that overwrites the μtag state in the L1D. Since the abort path does not execute VERW, the L1D cache state remains usable for TSA-L1 extraction after control returns to the hypervisor. Consequently, as shown in Figure 2 (a), a malicious hypervisor can employ TSA-L1 to leak the cached CVM register state from the encrypted VMSA. The affected VMSA region contains architecturally relevant guest state, including control-flow state, general-purpose registers, and extended architectural state such as floating-point/vector registers. Such plaintext access effectively re-enables previous passive attacks that were possible against the initial revision of SEV, before the VMSA got encrypted in SEV-ES [32].

Leaking CVM Memory. Worse, we show that the VMRUN checks the VMSA RMP permission after performing memory loads from it. As shown in Figure 2 (b), we can replace the VMSA pointer with an arbitrary CVM page address, causing VMRUN to fetch decrypted guest memory into the L1D cache before eventually aborting. The RMP permission check that should reject a non-VMSA guest page therefore occurs too late to prevent plaintext data from reaching the L1D cache. We see that the CPU issues three loads to an arbitrary CVM page determined by a malicious hypervisor. These loads contain the ASID of the CVM in the upper physical bits and will thus result in fetching encrypted

CVM memory decrypted into the L1D cache. By using TSA-L1, a hypervisor can thus decrypt arbitrary CVM data which resides at page offsets `0x0c0`, `0x380`, and `0x3c0`. Misaligning the supplied guest-page address does not expose additional cache lines, as `VMRUN` ignores bits 0–11 and proceeds as if the address were page-aligned.

6.2. Leaking Microcode

AMD encrypts more recent microcode updates for Zen CPUs. This makes inspecting recent updates with tools such as Zentool impossible [33]. Since AMD must decrypt the microcode patch at some point to apply it, we hypothesize that it may reside in the L1D cache and we can extract it using TSA-L1.

Privileged software can trigger microcode updates by executing `WRMSR` with the microcode patch address in `RAX` and `MSR_AMD64_PATCH_LOADER` as the register. We trace the microcode update using IBS. During the microcode update, the CPU verifies the signature of the update blob and decrypts it. AMD temporarily decrypts the microcode in-place in memory accessible through the HyperTransport address space, as shown by the IBS trace in Listing 6. It shows that the CPU first loads the encrypted microcode at an address, performs multiple operations, and later writes to the same address. Based on the access latency, we assume the HyperTransport range the CPU accesses indexes to CPU internal SRAM or L1/L2 caches. The IBS trace further shows that when the CPU finishes the microcode update, it does not apply the TSA-L1 mitigation. This would leave the decrypted microcode patch in the L1D cache should the HyperTransport address loads fetch the data into the cache. Upon closer IBS trace inspection, we note that the CPU decrypts the microcode patch in-place and that the writes hit the caches.

Next, we test whether we can speculatively leak HyperTransport memory by using TSA-L1. We use virtual addresses that have a colliding μtag with the physical HyperTransport addresses. Subsequently, we use those addresses to extract the HyperTransport memory with TSA-L1. We can successfully extract microcode. To validate that we leak correct microcode, we use Zentool to analyze the decrypted HyperTransport memory. The analysis shows that we leak valid μops.

```
[...]
[load4]  p=0xffffdf70008c8  v=0x0
[ALU]
[store4] p=0xffffdf70008c8  v=0x0
[...]
[load4]  p=0xffffdf7001b58  v=0x0
[ALU]
[...]
[load4]  p=0xffffdf7001b5c  v=0x0
[ALU]
[store4] p=0xffffdf7001b5c  v=0x0
[...]
```

Listing 6. Partial IBS trace showing in-place microcode patch decryption.

TABLE 3. CPU PLATFORMS USED FOR THE EXPERIMENTS.

CPU	Gen.	Fam.	Model	Step.	Microcode
EPYC 7313	Zen 3	25	1	1	`0x0a0011d7`
EPYC 9124	Zen 4	25	17	1	`0x0a101158`
EPYC 9334	Zen 4	25	17	1	`0x0a101158`

6.3. Leaking System Management Mode RAM

System Management Mode, also known as Ring -2, runs privileged code responsible for low-level hardware management. A System Management Interrupt (SMI) triggers the transition from kernel or user space to System Management Mode. When transitioning to System Management Mode, the CPU saves the host state into a State-Save Area [20]. Contrary to the switch from host to guest, the State-Save Area when switching to SMM mode does not reside in host memory, but in SMM memory. SMM memory is not accessible by the kernel or user space. We observe that AMD does block any IBS tracing when entering SMM. The blocking prevents IBS from sampling instructions in SMM execution mode. Thus, we cannot reason about AMD-applied TSA mitigations on the SMM exit path.

To experimentally confirm whether mitigations are in place, we aim to leak known values from the SMM State-Save Area within SMM memory with TSA-L1. We create an address collision with the physical address of the SMM State-Save Area, which resides at a known per-core address [20]. The SMM State-Save Area holds registers such as `cr3` or `SMBASE`. We run the TSA-L1 extraction by creating a virtual address with the same μtag as the State-Save Area and running our extraction script. We confirm that kernel space can extract SMM RAM data using TSA-L1, violating AMD platform guarantees.

7. Implementation & Evaluation

In this section, we describe the implementation of our mitigation-bypass experiments. We use the TSA-L1 gadget from Listing 3 together with the PMCs identified in Section 4 to extract transiently reachable data from the vulnerable execution paths. We evaluate our implementation on the AMD EPYC 7313, EPYC 9124, and EPYC 9334 platforms listed in Table 3.

7.1. Bulk Leakage Accuracy

We evaluate the accuracy and leakage rate of TSA-L1 over 5 KiB of known data. Since TSA-L1 operates at cache-line granularity, we reuse the same fixed address across measurements while evaluating 5 KiB of ground-truth data.

Experiment 1. We use three attacker-controlled pages, P_{ext}, P_{cal}, and P_{sec}, mapped at virtual addresses sharing the same μtag. Before each extraction, we use P_{ext} and P_{cal} to calibrate PMC `0x76` with umask `0x80`, identified in Section 4.3. The calibration computes the threshold used to distinguish whether the speculatively forwarded bit is

TABLE 4. TSA-L1 EXTRACTION ACCURACY AND LEAKAGE RATE ON EPYC 7313 (ZEN 3) AND EPYC 9124 (ZEN 4) FOR DIFFERENT NUMBERS OF SAMPLES PER BIT USING `PMCx076/0x80`. VALUES ARE REPORTED AS MEAN ± STANDARD DEVIATION ACROSS 100 RUNS.

Samples	Zen 3		Zen 4	
	Accuracy [%]	Rate [KiB/s]	Accuracy [%]	Rate [KiB/s]
1	98.65±0.59	463±11.50	98.67±0.72	443±0.48
2	95.45±16.81	228±40.02	79.52±39.76	180±89.93
4	98.46±0.79	118±0.06	87.31±32.25	99±36.71
8	99.12±0.82	59±0.20	99.37±0.53	57±0.03
16	98.36±0.98	30±0.07	99.15±0.71	28±0.06
32	99.39±0.67	15±<0.01	98.93±0.76	14±0.02
50	99.29±0.67	9±<0.01	98.73±0.84	9±<0.01
64	99.17±0.66	7±0.01	98.91±0.73	7±<0.01
100	98.84±0.78	5±<0.01	98.40±0.81	5±<0.01
128	98.72±0.79	4±<0.01	98.85±0.83	4±<0.01
256	91.66±25.16	2±0.47	97.65±9.85	2±0.18
512	98.75±0.79	1±<0.01	97.74±1.11	1±<0.01

interpreted as 0 or 1. After calibration, we bring the corresponding cache line of P_{sec} into the L1D. Then we use a P_{ext} with the TSA-L1 gadget from Listing 3 to extract the cache-line contents of P_{sec}. We repeat the procedure for 5 KiB of data.

Result 1. Table 4 reports the leakage accuracy, defined as the fraction of correctly recovered bits, and the leakage rate, defined as the KiB/s throughput independent of bit correctness, for different numbers of samples per bit. With per-cache-line recalibration, we achieve 98–99% accuracy for most sampling configurations on both Zen 3 and Zen 4.

7.2. Decrypting SEV-SNP CVM Register State

We extract arbitrary register state from a SEV-SNP CVM.

Experiment 2. To leak data from the encrypted VMSA, we first prepare a minimal Linux guest running under SEV-SNP. We use a modified Linux guest kernel that does not complete the normal boot process, but instead reaches a controlled test point where it repeatedly loads a fixed marker value into `RAX`. This ensures that the corresponding field in the VMSA contains a known value before the hypervisor executes the aborting `VMRUN`.

For the extraction, we use two host pages, P_{ext} and P_{cal}. Both pages are distinct from the encrypted VMSA page and therefore do not contain the secret value. We map them at virtual addresses selected to collide with the μtag of the physical VMSA page. We use P_{ext} as the extraction address, while we use P_{cal} to collect calibration samples for the selected PMC.

After calibration, the hypervisor writes an invalid value to the `event_inj` field in the VMCB and executes `VMRUN`. The instruction aborts and returns to the host. The hypervisor then uses P_{ext} plus the page offset corresponding to the `RAX` field with our TSA-L1 gadget to extract the marker value from the VMSA. We repeat this procedure for each selected PMC and merge the recovered values across PMCs.

Result 2. With this implementation, we recover the 64-bit `RAX` marker from the encrypted VMSA on all tested systems with 100% accuracy.

Experiment 3. To demonstrate the impact of VMSA extraction, we leak disk-encryption keys from a minimal Linux guest using dm-crypt with AES-XTS under SEV-SNP. In a real attack, the hypervisor can first extract control-flow state, such as `LSTAR` or `RIP`, from the VMSA to break virtual KASLR and then use SEV-Step [15] to single-step the guest until the key material reaches architectural registers. For our demonstration, we emulate this trigger point by modifying the guest kernel to execute `VMMCALL` directly after the AES-XTS set-key function has placed the disk-encryption key in registers. The resulting VM exit gives the hypervisor a controlled point at which the key is present in architectural state.

We reuse the calibration procedure and μtag-colliding extraction setup from the previous experiment. The hypervisor then writes an invalid value to the `event_inj` field in the VMCB and executes `VMRUN`. The instruction aborts and returns to the host after accessing the decrypted VMSA state. The hypervisor then uses P_{ext} plus the page offsets corresponding to the `XMM0` and `XMM2` fields with our TSA-L1 gadget to extract the AES-XTS disk-encryption key from the VMSA.

Result 3. We fully recover the AES-XTS disk-encryption key with 100% accuracy.

7.3. Extracting SEV-SNP CVM Memory

We can leak not only register state but also CVM memory.

Experiment 4. To leak guest-page contents, we prepare the guest similarly to Section 7.2. Instead of writing a marker value only into `RAX`, the guest writes known marker values to an allocated guest page P_{guest}. The calibration procedure is also identical. The difference is that we map P_{ext} and P_{cal} at virtual addresses selected to collide with the μtag computed for the host physical address backing P_{guest}, rather than the physical address of the VMSA page.

After calibration, we replace the VMSA pointer in the VMCB with the host physical address of P_{guest} and execute `VMRUN`. The instruction aborts and returns to the host. We then use P_{ext} plus the desired page offset with our TSA-L1 gadget to leak the corresponding cache line accessed by the aborting `VMRUN` path.

Result 4. With this implementation, we recover three cache lines at page offsets `0x0c0`, `0x380`, and `0x3c0` on all tested systems with 100% accuracy. Based on our IBS trace, other page offsets are not loaded into the cache, and attempts to extract them fail.

7.4. Extracting Microcode Loader State

We extract residual state that the microcode patch loader leaves in the L1D cache after execution.

Experiment 5. We use the same base structure as in Section 7.2. Instead of targeting the physical address of the VMSA, we target the HyperTransport address the microcode patch loader uses. The extraction again uses two attacker-controlled host pages, P_{ext} for extraction and P_{cal} for calibration. We map both pages at virtual addresses selected to collide with the μtag of the target HyperTransport-addressed access.

After calibration, we execute `WRMSR` to `MSR_AMD64_PATCH_LOADER` to bring the target HyperTransport-addressed line into the cache. We then use P_{ext} plus the page offset corresponding to the selected microcode with our TSA-L1 gadget to extract the line contents.

Result 5. We recover decrypted microcode patch contents from the HyperTransport-addressed patch-loader region on all tested systems. On the AMD EPYC 7313, we additionally recover the CMAC key that AMD used before the EntrySign vulnerability to validate the signature of a microcode update [24]. Microcode on our AMD EPYC 9334 and AMD EPYC 9124 machines already contains the EntrySign mitigations and thus prevents recovery of the CMAC key with the same procedure. However, we still leak decrypted microcode patches on those.

Compared to the `VMRUN`-based extraction, microcode leakage produces a weaker and less stable PMC signal. Usable configurations are sparser and less consistent across runs, requiring a full PMC sweep and repeated measurements for promising events. We suspect that this stems from the patch-loader path requiring approximately 1.1 million μops, which introduces substantially more intervening microarchitectural activity before extraction.

7.5. Extracting System Management Mode RAM

We leak memory residing in kernel-mode inaccessible memory, the SMM RAM.

Experiment 6. To leak SMM RAM contents, we trigger an SMI and extract data from the SMM State Save Area after the processor returns from SMM. As a validation target, we leak the `SMBASE` field, which is located at offset `0xFF00` relative to the SMM base. We obtain the expected value by reading MSR `0xC0010111` on the same core, since the SMM base is core-local. This known value also serves as calibration input: for each tested PMC, we choose two bit positions in `SMBASE` with known opposite values and repeatedly trigger an SMI using Listing 7 to determine the leakage threshold. After calibration, we trigger the SMI again and use the calibrated TSA-L1 gadget to extract the full `SMBASE` value.

```
apic->send_IPI(RUN_ON_THREAD, APIC_DM_SMI);
```

Listing 7. Linux kernel SMI Trigger.

Result 6. With this implementation, we recover the `SMBASE` value from SMM RAM on all tested systems with 100% accuracy. We further repeat the experiment for additional SMM State Save Area fields (i.e., `CR3` and `R13`) and recover all tested values with 100% accuracy.

TABLE 5. DOCUMENTED WORKING PMC CONFIGURATIONS FOUND FOR TSA-L1 BY THE EVENT-SELECT AND UMASK SWEEP.

CPU	Event	Description	Count
Zen 3	PMCx076	Cycles not in halt	256
	PMCx087	Instruction-pipe stall	200
	PMCx090	L3 cache miss latency	236
Zen 4	PMCx076	Cycles not in halt	256
	PMCx0A9	Op queue empty	129
	PMCx0D6	Cycles with no retire	128
	PMCx1A0	Dispatch stalls per slot	128
	PMCx1A2	Dispatch additional resource stalls	224

8. Discussion

In this section, we contextualize our findings by analyzing the microarchitectural signals observed during evaluation, isolating the root cause of the vulnerability, and discussing mitigation strategies that can eliminate or reduce the leakage.

8.1. Performance Counters

We sweep all available PMCs to see which documented and undocumented PMCs measure speculative scheduler contention. The exact set of working PMCs varies between different executions. Most PMCs that measure speculative scheduler contention are undocumented. We were unable to locate their meanings in the Linux kernel source code or in any public AMD documentation.

Nevertheless, the documented events in Table 5 provide insight into the behavior our gadget amplifies. Many correspond to stalls, waiting cycles, empty queues, or dispatch and retirement backpressure, suggesting that the secret-dependent signal primarily appears as an execution delay. This matches our amplification gadget, where the transiently forwarded bit selects between memory-operation patterns that create different load-side pressure.

Figure 3 further shows many duplicate counter values, indicating that several undocumented event/umask combinations may alias the same underlying event or closely related internal conditions. Overall, the sweep identifies noisy but usable measurement sources rather than a fixed set of architecturally stable leakage counters.

8.2. TSA Amplification Gadget

Our TSA PoCs in Listing 2 and Listing 3 use unaligned loads to cause speculative scheduler contention. We inspect the scheduler contention by comparing different kinds of loads and measure the execution time using IBS. With IBS, we measure the latency of the load in Line 4 in Listing 8. We vary `offset` to measure aligned and unaligned cases within the same cache line, and unaligned cache line crossing and aligned cache line crossing. We see that only cache-line-crossing loads require 1 additional cycle, in total 10

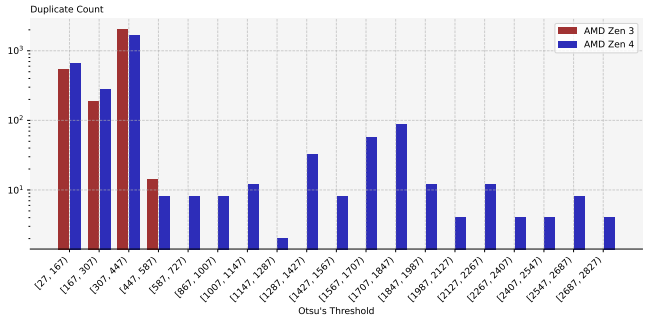


Figure 3. Grouped PMC counter values.

```
asm volatile(
  ".rept 20\n\t"
  "lfence \n\t"
  "movq (%0),%%rax\n\t"
  "lfence \n\t"
  ".endr\n\t"
  ::"r"(ptr+offset):"rax","memory"
);
```

Listing 8. Move latency code sequence.

cycles, to complete compared to non-crossing loads. Thus, the root cause for the scheduler contention is most likely the speculatively executed cache-line-crossing load. It is either observable through PMCs or will offset other instructions such that they may not execute [17].

8.3. SKYFALL Root Cause

Our results suggest that the root cause of SKYFALL is a mismatch between the assumed and actual scope of TSA-L1. The currently deployed mitigation focuses on ordinary virtual-address-based accesses and privilege transitions. However, our experiments show that physical-address-based and CPU-internal paths can also introduce sensitive data into L1D-reachable predictor state. This state can subsequently alias with attacker-controlled virtual addresses, allowing leakage across architectural boundaries.

The VERW-based mitigation is effective when executed, but certain sensitive producer paths return without clearing TSA-L1-relevant predictor state. Any instruction path that brings sensitive physical or internal data into L1D-reachable state without a subsequent predictor-state clear can therefore become a leakage source.

8.4. Mitigations

We discuss potential mitigations against the TSA variants we find.

Disable way prediction. A direct mitigation would be to disable L1D way prediction, or at least to prevent physical-address-based and CPU-internal accesses from creating μtag state that can be matched by attacker-controlled virtual addresses. This would prevent sensitive data introduced through privileged instruction paths from becoming reachable through TSA-L1. However, disabling way prediction entirely carries a non-trivial performance cost, since the predictor exists to reduce power consumption and bank conflicts.

Domain tagging. A cleaner architectural fix would be to introduce domain tagging in the way predictor, analogous to ASID tagging in TLBs. Physical-address-based and CPU-internal accesses would then populate a separate predictor namespace that is unreachable by ordinary virtual-address loads. This would make the aliasing condition required by our attack architecturally impossible without disabling prediction entirely. On the downside, this mitigation requires hardware changes and more space in the hardware, as additional bits must be added.

Non-cacheable memory. A complementary approach is to avoid placing sensitive internal data in L1D-reachable structures. Privileged instructions, firmware, and microcode could instead operate on non-cacheable memory or on structures that are not exposed to the L1D way predictor. This would address TSA-L1 specifically, but it would not eliminate other transient channels, such as store queue leakage, through which recently written values may remain transiently accessible.

Extend `VERW` based mitigation. AMD can extend existing VERW-based mitigation beyond explicit trust-boundary transitions. Because VERW is effective wherever it executes, it should also be issued before returning from privileged instruction paths that may have touched sensitive physical-addressed or CPU-internal data, including abort and error paths. However, this approach requires AMD to correctly identify every such path, which is precisely the kind of incomplete coverage that enabled SKYFALL in the first place.

9. Related Work

Transient execution attacks are broadly categorized into those targeting the CPU's branch prediction mechanisms, such as Spectre-style attacks [8], [9], and those exploiting the memory subsystem, such as Meltdown and Microarchitectural Data Sampling (MDS) [1], [2], [3]. Because our work is situated in the memory domain, we focus our discussion on the latter class of vulnerabilities.

The foundational Meltdown [1] attack exploited transient data forwarding during memory access permission faults, enabling unauthorized read access to privileged kernel memory. Subsequently, Foreshadow [14] (L1 Terminal Fault or L1TF) demonstrated that transient data forwarding also occurs during page table terminal faults. By matching physical address tags in the L1D cache even when a page table entry is marked invalid, Foreshadow bypassed isolation boundaries in Intel SGX enclaves and cloud virtual machines. The core exploitation paradigm of these attacks relies on triggering an invalid memory access, speculatively encoding the resulting transient data into a secondary microarchitectural

state (such as a cache line), and extracting it using cache side-channel analysis techniques like Flush+Reload.

This paradigm has since been extended to target microarchitectural buffers beyond the L1D cache. Specifically, Microarchitectural Data Sampling (MDS) attacks exploit transient leakage from internal queues: ZombieLoad and RIDL target the Line Fill Buffer (MFBDS) [2], [3], RIDL targets the Load Buffer (MLBDS) [3], Fallout targets the Store Buffer (MSBDS) [4], and CrossTalk targets a special staging buffer (SRBDS) [7]. Automated frameworks like Transynther [34] have streamlined regression testing and the discovery of these vulnerabilities, uncovering gaps in MDS mitigations [35]. Most recently, Downfall [6] demonstrated that vector register files can be leaked via gather instructions (Gather Data Sampling, or GDS), while subsequent analysis by Intel revealed that the integer register file is similarly vulnerable to Register File Data Sampling (RFDS) [36]. Notably, all of these MDS attacks have targeted Intel CPUs.

Confidential CVMs. The latest AMD confidential computing technology, SEV-SNP, has been targeted by many attacks in the past. Google was the first to analyze the security of SEV-SNP and uncovered severe flaws [37]. Numerous ciphertext-based attacks have been used to break confidentiality [38], [39], [40], [41], [42], [43]. These all focus on breaking confidentiality without attacking integrity. CacheWarp was the first integrity attack on SEV-SNP by dropping modified cache lines [44]. Hecker and WeSee corrupt integrity by injecting interrupts into SEV-SNP CVMs [45], [46]. RMPocalypse uses incomplete protections during RMP initialization to hijack control over SEV-SNP [47]. Fabricked corrupts the Infinity Fabric to redirect Platform Security Processor (PSP) memory requests into the void [48]. Mior builds a device-as-memory primitive to redirect PSP memory reads to MMIO memory [49]. BreakFAST reroutes PSP memory requests into an adjacent address space in the control fabric to gain privilege escalation on said control fabric [50]. Staleus makes the PSP non-memory coherent to cause a split view between the PSP and x86 cores [51]. Counter-SEVeillance uses PMCs to observe CVM execution state [52]. BadAML corrupts SEV-SNP CVMs by using an attestation gap of CVM UEFIs [53]. Multiple attacks use physical DRAM attacks to break SEV-SNP [54], [55], [56]. Stackwarp uses an unlocked MSR bit to desynchronize the stack engine to corrupt guest state [57]. On Intel, similar analyses have been undertaken to violate security invariants. TDXdown and TDXploit circumvent single-stepping mitigations on Intel TDX [58], [59]. TDXray combines known side-channels to extract LLM prompts from TDX CVMs [60]. In contrast, SKYFALL is the first attack exploiting an MDS-style vulnerability against CVMs.

10. Conclusion

In this paper, we revisited Transient Scheduler Attacks (TSA) on AMD Zen processors and exposed limitations in their existing mitigations. By reverse-engineering the undocumented L1D way-predictor micro-tag (μtag) hash functions and leveraging Instruction Based Sampling (IBS), we demonstrated that physical-address-based and CPU-internal memory paths (such as those used by privileged virtualization and microcode instructions) still populate predictor state. This oversight enables our proposed attack, SKYFALL, to bypass defenses (like VERW) and leak sensitive architectural state. We demonstrated the impact of this vulnerability by successfully leaking data from an AMD SEV-SNP confidential virtual machine, extracting decrypted microcode updates, and leaking SMM RAM. Ultimately, our findings show that transient execution defenses must comprehensively cover both physical and virtual memory access paths, highlighting the need for better domain isolation in way-predictor designs and verification of mitigations for future processor architectures.

11. Acknowledgment

We thank the anonymous reviewers and Philipp Giersfeld for their feedback, which helped in improving the paper. We further thank Eduardo Vela Nava for inspiring us to investigate TSA more closely. Benedict Schlüter is supported by the Google PhD Fellowship in Privacy, Safety, and Security.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom *et al.*, "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, 2020.

[2] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[3] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[4] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[5] 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[6] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[7] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021.

[8] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[9] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against {Cross-Privilege} spectre-v2 attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[10] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014.

[11] Intel, "Intel Trust Domain Extensions (Intel TDX)," Accessed: 2026-06-10.

[12] AMD, "AMD SEV-SNP: Strengthening VM Isolation with Integrity protection and more," https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020.

[13] 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 *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018.

[14] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *SysTEX*, 2017.

[15] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "Sev-step a single-stepping framework for amd-sev," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/11250

[16] Advanced Micro Devices, Inc., "Security Analysis of AMD Predictive Store Forwarding," AMD, Tech. Rep., 2023, white paper, updated August 2023.

[17] O. Oleksenko, F. Solt, C. Fournet, J. Hofmann, B. Köpf, and S. Volos, "Enter, exit, page fault, leak: Testing isolation boundaries for microarchitectural leaks," in *2026 IEEE Symposium on Security and Privacy*, May 2026, accepted at IEEE SP 2025, publication delayed due to embargo. [Online]. Available: https://www.microsoft.com/en-us/research/publication/enter-exit-page-fault-leak-testing-isolation-boundaries-for-microarchitectural-leaks/

[18] Advanced Micro Devices, "Technical guidance for mitigating transient scheduler attacks," Advanced Micro Devices, Inc., Whitepaper, 2025, revision 2.0. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/resources/bulletin/technical-guidance-for-mitigating-transient-scheduler-attacks.pdf

[19] *Software Optimization Guide for the AMD Zen4 Microarchitecture*, Advanced Micro Devices, Inc., 2023.

[20] AMD, "AMD64 Architecture Programmer's Manual Volumes 1–5, Rev. 4.08 (40332)," https://docs.amd.com/v/u/en-US/40332-PUB_4.08, 2023.

[21] B. Holden, D. Anderson, J. Trodden, and M. Daves, *HyperTransport 3.1 Interconnect Technology*, 1st ed. Mindshare Press, 2008.

[22] T. Burd, N. Beck, S. White, M. Paraschou, N. Kalyanasundharam, G. Donley, A. Smith, L. Hewitt, and S. Naffziger, ""Zeppelin": An SoC for Multichip Architectures," *IEEE Journal of Solid-State Circuits*, 2019.

[23] AMD, "AMD I/O virtualization technology (IOMMU) specification," http://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf, 2016.

[24] Google, "Zen and the Art of Microcode Hacking," https://bughunters.google.com/blog/5424842357473280/zen-and-the-art-of-microcode-hacking, 2025.

[25] A. Louka and J. Van Bulck, "Sok: 20 years of power, privilege, and peril in x86 system management mode," in *20th USENIX WOOT Conference on Offensive Technologies*, Aug. 2026.

[26] Microsoft, "Detecting TSA-SQ with Revizor," https://github.com/microsoft/side-channel-fuzzer/blob/208025d54ca9987f76f10a0e384b448b3c1f2f24/docs/intro/tutorials/tsa-sq.md, 2025, accessed: 2026-10-10.

[27] A. Fog, *Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD, and VIA CPUs*, 2025, version last updated 2025-09-20. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[28] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, 1979.

[29] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3320269.3384746

[30] *Software Optimization Guide for the AMD Zen5 Microarchitecture*, Advanced Micro Devices, Inc., 2024, 58455.

[31] Intel Corporation, "Microarchitectural data sampling," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html, 2021.

[32] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose, "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves," in *ACM AsiaCCS*, 2019.

[33] J. Eads, M. Rizzo, K. Janke, E. Vela Nava, T. Ormandy, and S. Schmieg, "Zentool - AMD Zen Microcode Manipulation Utility," https://github.com/google/security-research/tree/8f33fa850e3b94ce5f29d63528d21e81a6c62f44/pocs/cpus/entrysign/zentool, Google Hardware Security Team, 2024.

[34] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[35] D. Moghimi, "Data sampling on mds-resistant 10th generation intel core (ice lake)," 2020. [Online]. Available: https://arxiv.org/abs/2007.07428

[36] Intel, "Register File Data Sampling / CVE-2023-28746 / INTEL-SA-00898," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/register-file-data-sampling.html, 2025.

[37] Google, "AMD Secure Processor for Confidential Computing," https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/AMD_GPZ-Technical_Report_FINAL_05_2022.pdf, 2022.

[38] B. Schlüter, C. Wech, and S. Shinde, "Heracles: Chosen Plaintext Attack on AMD SEV-SNP," in *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[39] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[40] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, " CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels ," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

[41] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[42] Y. Yan, W. Huang, I. Grishchenko, G. Saileshwar, A. Mehta, and D. Lie, "Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels," in *Proceedings of the 34th USENIX Conference on Security Symposium*, 2025.

[43] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, "HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[44] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[45] B. Schlüter, S. Sridhara, M. Kuhne, A. Bertschi, and S. Shinde, "Heckler: Breaking Confidential VMs with Malicious Interrupts," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[46] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde, "WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[47] B. Schlüter and S. Shinde, "RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP," in *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[48] B. Schlüter, C. Wech, and S. Shinde, "Fabricked: Misconfiguring Infinity Fabric to Break AMD SEV-SNP," in *35th USENIX Security Symposium (USENIX Security 26)*. USENIX Association, 2026.

[49] C. Wech, P. Giersfeld, B. Schlüter, and S. Shinde, "Mior: Breaking AMD SEV-SNP with Device-as-Memory," in *ACM CCS*, The Hague, Netherlands, 2026.

[50] P. Giersfeld, B. Schlüter, and S. Shinde, "BreakFAST: Confused Deputy Attack on Infinity Fabric to Break AMD SEV-SNP," in *47th IEEE Symposium on Security and Privacy (S&P 26)*. IEEE, 2026.

[51] B. Schlüter and S. Shinde, "Staleus: Breaking AMD SEV-SNP via Memory Incoherence," in *35th USENIX Security Symposium (USENIX Security 26)*. Baltimore, MD: USENIX Association, Aug. 2026.

[52] S. Gast, H. Weissteiner, R. L. Schröder, and D. Gruss, "Counter-SEVeillance: Performance-Counter Attacks on AMD SEV-SNP," in *Network and Distributed System Security Symposium 2025: NDSS 2025*, 2025.

[53] S. Takekoshi, M. Mori, T. Fukai, and T. Shinagawa, "BadAML: Exploiting Legacy Firmware Interfaces to Compromise Confidential Virtual Machines," in *ACM CCS*, 2025. [Online]. Available: https://doi.org/10.1145/3719027.3765123

[54] J. De Meulemeester, L. Wilke, D. Oswald, T. Eisenbarth, I. Verbauwhede, and J. Van Bulck, "Badram: Practical memory aliasing attacks on trusted execution environments," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

[55] J. Chuang, A. Seto, N. Berrios, S. van Schaik, C. Garman, and D. Genkin, "TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition," in *47th IEEE Symposium on Security and Privacy (IEEE S&P '26)*, 2026.

[56] J. De Meulemeester, D. Oswald, I. Verbauwhede, and J. Van Bulck, "Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing," in *47th IEEE Symposium on Security and Privacy (S&P)*, 2026.

[57] R. Zhang, T. Hornetz, D. Weber, F. Thomas, and M. Schwarz, "StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine," in *USENIX Security*, 2026.

[58] L. Wilke, F. Sieck, and T. Eisenbarth, "Tdxdown: Single-stepping and instruction counting attacks against intel tdx," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3658644.3690230

[59] F. Rauscher, L. Wilke, H. Weissteiner, T. Eisenbarth, and D. Gruss, "TDXploit: Novel techniques for Single-Stepping and cache attacks on intel TDX," in *34th USENIX Security Symposium (USENIX Security 25)*. Seattle, WA: USENIX Association, Aug. 2025. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/rauscher

[60] T. Hornetz, H. Yavarzadeh, A. Cheu, A. Gascon, L. Gerlach, D. Moghimi, P. Schoppmann, M. Schwarz, and R. Zhang, "TDXRay: Microarchitectural side-channel analysis of intel TDX for real-world workloads," in *IEEE Symposium on Security and Privacy (S&P)*, 2026.

Appendix A. Finding the μtag

We assume the μtag function is linear and thus consists only of XORs. We model the μtag computation as an $\mathbb{F}_2$-linear mapping $h : \mathbb{F}_2^{64} \to \mathbb{F}_2^m$, represented by a binary transformation matrix $H \in \mathbb{F}_2^{m \times 64}$ such that $\mu\text{tag}(\mathbf{v}) = H\mathbf{v}$. We first determine the active bits of the input by identifying which virtual address bits contribute to the image of H. We use virtual addresses with a Hamming weight of 1 and see if they collide with the null virtual address. The experiment shows that the effective input domain is restricted to bits $[12, 27]$ on Zen 3 and Zen 4 ($n = 16$), and bits $[12, 31]$ on Zen 5 ($n = 20$). Because h is linear, $h(\mathbf{0}) = \mathbf{0}$; thus, any address vector $\mathbf{v}$ yielding a tag collision with the zero vector lies in the kernel (null space) of the transformation, satisfying $H\mathbf{v} = \mathbf{0}$. To avoid an exhaustive exponential search over the 2^n candidate inputs, we optimize the search by evaluating vectors with a Hamming weight of 2, setting $\mathbf{v} = \mathbf{e}_i \oplus \mathbf{e}_j$ for $i, j \in [12, 27]$ or $i, j \in [12, 31]$. A collision with the zero vector implies $h(\mathbf{e}_i \oplus \mathbf{e}_j) = \mathbf{0} \iff h(\mathbf{e}_i) = h(\mathbf{e}_j)$, identifying pairs of columns in H that are identical. From this, we learn that one bit of the μtag is the result of XORing input bits i and j of the virtual address. Testing all $\binom{n}{2}$ pairs yields a system of linearly independent kernel vectors over $\mathbb{F}_2$ from which we reconstruct the linear combinations in H. Active

bit positions that produce no 2-bit collisions correspond to singleton coordinate projections. In other words, those are address bits mapped directly to a tag bit without being XORed with another bit.

Example. Consider a 2-bit μtag function $h\colon \mathbb{F}_2^{64} \to \mathbb{F}_2^2$ where the first bit is formed by XORing bits 15 and 31, and the second bit is the standalone bit 32. We know that bits 15,31 and 32 are the active bits that contribute to the μtag.

$$\mu\text{tag}(\mathbf{v}) = \begin{bmatrix} v_{15} \oplus v_{31} \\ v_{32} \end{bmatrix}, \tag{3}$$

which corresponds to the matrix multiplication:

$$H\mathbf{v} = \begin{bmatrix} \dots & 1 & \dots & 1 & 0 & \dots \\ \dots & 0 & \dots & 0 & 1 & \dots \end{bmatrix} \begin{bmatrix} \vdots \\ v_{15} \\ \vdots \\ v_{31} \\ v_{32} \\ \vdots \end{bmatrix}. \tag{4}$$

When evaluating candidate 2-bit perturbations:

1) **Detecting XOR'ed Pairs:** Querying $\mathbf{v} = \mathbf{e}_{15} \oplus \mathbf{e}_{31}$ yields:

$$h(\mathbf{e}_{15} \oplus \mathbf{e}_{31}) = \begin{bmatrix} 1 \oplus 1 \\ 0 \oplus 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \mathbf{0}. \tag{5}$$

Since $h(\mathbf{e}_{15} \oplus \mathbf{e}_{31}) = \mathbf{0}$, this vector lies in $\ker(h)$, proving $h(\mathbf{e}_{15}) = h(\mathbf{e}_{31})$ and confirming that bits 15 and 31 combine via XOR into one of the two tag bits. So our search script would show a collision when bits 15 and 31 are set simultaneously.

2) **Isolating Standalone Projections:** Querying pairs involving bit 32 (e.g., $\mathbf{v} = \mathbf{e}_{15} \oplus \mathbf{e}_{32}$) yields:

$$h(\mathbf{e}_{15} \oplus \mathbf{e}_{32}) = \begin{bmatrix} 1 \oplus 0 \\ 0 \oplus 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \neq \mathbf{0}. \tag{6}$$

Because no pairwise perturbation over the active bits in $\mathbb{F}_2^{64}$ involving bit 32 produces a zero collision, it is isolated as a standalone coordinate projection mapping directly to the second tag bit.

Limitation With this method, we cannot order the output coordinates. However, for our use case where we are only interested in collisions, this limitation is irrelevant.

Ethics considerations

We engaged in a coordinated vulnerability disclosure process with AMD and did not make any exploit code public before the embargo.

Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

Summary

The paper presents SKYFALL, a transient execution attack on AMD Zen processors that exploits disclosed microarchitectural leakage in the physical address paths that AMD's VERW-based TSA mitigation fails to address. The paper shows that SKYFALL can be used to (1) leak a confidential VM's private states, (2) extract microcode patches, and (3) leak data sealed within the SMM memory.

Scientific Contributions

- 1. Independent Confirmation of Important Results with Limited Prior Research.
- 5. Identifies an Impactful Vulnerability.
- 6. Provides a Valuable Step Forward in an Established Field.

Reasons for Acceptance

1) The paper provides an independent confirmation of important results with limited prior research. The paper discloses new insights about the TSA vulnerabilities through undocumented processor behaviors (L1D cache way prediction) and mitigation mechanisms (`VERW`) on AMD processors.
2) The paper identifies an impactful vulnerability. The paper introduces the SKYFALL attack against AMD Zen processors, which achieves leakage of SEV-SNP CVM states and SMM memory, as well as recovery of a decrypted microcode patch.
3) The paper provides a valuable step forward in an established field. The authors demonstrate that TSA-based attacks remain feasible despite existing countermeasures by exploiting the microarchitectural leakages that they uncovered.

Noteworthy Concerns

1) The paper lacks a comprehensive discussion of critical insights that lead to vulnerabilities and defense strategies to mitigate the SKYFALL attack beyond those about microcode updates. It also lacks a concrete evaluation of the performance trade-offs of the mitigation. This context would help future processor designs benefit from it to defend against the relevant class of transient execution attacks.
2) The paper fails to rigorously connect transient forwarding to the exploited PMC signals, which are undocumented, and to how this addresses the limitations of indirect cache-based exfiltration; these are fundamentals that the attack depends on. This context would strengthen the findings and validate them on technical grounds.
3) The paper lacks an extended discussion of the broader security consequences of the attack's capabilities in leaking CVM memory at three fixed offsets and AMD's decrypted microcode patches. This insight will strengthen the paper's takeaways.
4) The paper overlooks key limitations and requirements needed for a realistic end-to-end attack against CVMs: (1) core isolation without noise, (2) the attacker's ability to execute failing VMRUN and PMC reads at high frequency, and (3) assumptions of no CVM crashes/timeouts during the attack. This context would make the argument for real-world practicality more compelling.