StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine

Ruiyi Zhang, Tristan Hornetz, Daniel Weber, Fabian Thomas, Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Confidential Virtual Machines (CVMs), such as AMD SEV-SNP, aim to protect guest operating systems from an untrusted host by encrypting state and constraining privileged control. These platforms promise isolation even in multi-tenant cloud setups where simultaneous multithreading (SMT) remains enabled. While prior attacks focus on the memory hierarchy or execution units, they largely ignore frontend configurations.

In this paper, we present StackWarp, a software-based architectural attack exploiting the stack engine on AMD Zen CPUs to modify the stack pointer within an SEV-SNP guest, fully breaking integrity. StackWarp relies on an undocumented bit within a shared model-specific register (MSR) available on AMD Zen 1–5 CPUs that enables or disables the stack engine. Our reverse engineering shows that the state of the stack engine is not correctly synchronized across the logical cores, allowing an attacker to deterministically adjust the stack pointer on the sibling logical core across Zen generations, including fully patched Zen 5. We discover StackWarp via a systematic exploration of the MSR space, including undocumented MSRs. By flipping MSR bits, we discover bits that affect SEV-SNP guests running on a sibling logical core. To demonstrate the security impact, we show StackWarp in four end-to-end attacks on SEV-SNP guests: RSA-CRT private-key recovery, OpenSSH password-authentication bypass, and privilege escalations using either sudo or a kernel-mode ROP chain. We conclude with software hardening guidance and argue for a microcode or hardware change that prevents cross-core control of the stack engine when CVMs are active. Our results show that leaving SMT enabled undermines SEV-SNP integrity guarantees today.

1 Introduction

Cloud providers increasingly market Confidential Virtual Machines (CVMs)—notably AMD SEV-SNP and Intel TDX—as a drop-in way to run unmodified guest operating systems on untrusted hosts. These systems promise that an untrusted hypervisor can neither read plaintext guest memory nor alter a guest's execution, while still supporting cloud multiplexing practices such as simultaneous multithreading (SMT). AMD's implementation of CVMs is called SEV (Secure Encrypted Virtualization). SEV evolved in stages: SEV introduced memory encryption without protecting saved VM state. SEV-ES extended confidentiality to guest register state. The current version, SEV-SNP, added integrity by hardening nested paging so a host cannot freely rewrite a guest's address space. For current CPU generations, i.e., Zen 4 and Zen 5, only SEV-SNP is relevant, as the previous variants are considered broken and thus insecure [1].

Despite the increased security guarantees of SEV-SNP, recent papers show that CVMs can still be influenced in ways that bypass the intended isolation guarantees [2–6]. CacheWarp [4] broke the integrity guarantees by demonstrating that a malicious host can roll back selected dirty cache lines so the guest continues with architecturally stale values, enabling end-to-end exploits on SEV-SNP without plaintext access. This vulnerability has been mitigated with microcode updates on Zen 3, and was fixed on Zen 4 CPUs. Similarly, interrupt-based integrity attacks on CVMs [5, 6] that allow attackers to divert the control flow and corrupt register values have also been mitigated. Concurrently, a separate line of work exploits that SEV-SNP allows the host to observe ciphertext and manage encrypted pages, leading to chosen-plaintext [7, 8] and ciphertext-side-channel attacks [2, 3] that break confidentiality. While these attacks, and traditional cache-based [9, 10] and contention-based [11–13] attacks, are not mitigated, they only impact confidentiality and can be mitigated by the guest in software [14, 15].

In this paper, we introduce StackWarp, a software-only architectural attack that breaks SEV-SNP integrity by abusing the *stack engine* on AMD Zen CPUs. StackWarp differs significantly from previous vulnerabilities, as it originates in the *core frontend*, not in the memory hierarchy or encryption interface. Across Zen 1-5, we exploit an undocumented, core-scoped MSR bit that enables or disables the stack engine. Crucially, its state is not correctly synchronized across sibling logical cores. When a sibling logical core toggles this bit

while the victim executes a stack instruction, that instruction retires (i.e., commits its results to the architectural state) with a mismatch between the memory access and the architectural update of the stack pointer. This results in a deterministic, attacker-chosen shift of the victim's stack pointer. The primitive requires no access to guest plaintext, gadget placement, or injected interrupts visible inside the guest.

At a high level, the stack engine tracks a running stack-pointer delta so common stack operations complete efficiently [16, 17]. Our measurements show that disabling the engine while stack operations are in flight "freezes" the accumulated delta such that stores commit, but the architectural stack-pointer update is withheld. Re-enabling later "releases" the deferred delta in one step.As logical cores are not correctly synchronized, an attacker can use this behavior to modify the stack pointer on the sibling hyperthread in a deterministic way, even if the sibling hyperthread executes an SEV-SNP guest. Only stack-pointer operations are affected, while other operations are unaffected. The injected offset is repeatable, bidirectional, and large enough for practical exploitation, as we observe single-shot shifts up to 640 bytes. Consequently, StackWarp can (i) redirect control flow to an earlier frame without corrupting architectural state or (ii) overwrite or skip arbitrary stack slots while preserving the overall stack depth.

Our attacker model follows SEV-SNP's: the attacker controls the hypervisor but has no control over the guest OS or guest applications. The attacker can pin vCPUs, issue privileged MSR writes, inject interrupts, and step execution using standard host capabilities, but cannot read guest plaintext memory or registers protected in the encrypted save area, nor run code inside the guest. This is in line with prior SEV-SNP integrity attacks [4–6] and stepping setups [18].

We demonstrate the real-world impact of StackWarp on fully patched Zen 4 and Zen 5 CPUs with SMT enabled via 4 case studies. First, we recover an RSA-2048 private key from Intel IPP using a single faulty RSA-CRT signature: our controlled stack shift corrupts exactly one CRT branch while the other remains correct, enabling classical Bellcore factorization [19, 20]. Second, we bypass OpenSSH password authentication by unwinding the nested call chain and substituting the return value from the inner call. Third, we bypass sudo password authentication by tampering with the epilogue of the `getuid()` system call to misdirect the return value load to the root UID. Finally, we obtain ring 0 code execution by placing attacker-controlled bytes onto the kernel stack via a syscall's in-stack buffer (e.g., used in the `select` syscall) and pivoting the stack pointer to that buffer right before `ret`. Across all four, the hypervisor mounts the attack without guest gadgets or plaintext and without relying on weaknesses in the memory-encryption mode.

These findings indicate that keeping SMT enabled today undermines SEV-SNP's integrity goals: a sibling core can change a guest's control and data flow through a shared frontend switch with instruction-level precision. The immediate stopgap is to run CVMs with SMT disabled on affected hardware. Longer-term, a microcode or hardware change should prevent cross-hyperthread control of the stack engine in CVM contexts. Such fixes could scope the bit to the issuing thread, clear the hidden delta on context switches, or lock the control when a CVM is active.

Contributions. In summary, our paper makes the following contributions:

1. We uncover cross-hyperthread control of the Zen stack engine via an undocumented MSR bit and show that its desynchronized state enables deterministic, attacker-chosen stack-pointer shifts inside SEV-SNP guests.
2. We introduce StackWarp, a software-only primitive that operates on all tested Zen generations, including fully patched Zen 5, and we quantify its determinism, directionality, and offset bounds with targeted microbenchmarks and performance counters.
3. We develop exploitation techniques that (a) redirect return-based control flow to earlier frames without perturbing other architectural state and (b) manipulate on-stack data while preserving stack depth, and we validate them in four end-to-end case studies (RSA-CRT key recovery, OpenSSH authentication bypass, Sudo authentication bypass, kernel-mode ROP).
4. We provide hardening guidance and argue for a microcode/hardware fix that locks or scopes the offending control when CVMs are active. Disabling SMT is an immediate, effective stopgap.

Outline. Section 2 provides the necessary background. Section 3 describes the threat model and our automated approach that discovered StackWarp. Section 4 attributes the fault to the stack engine and presents our measurement-based root-cause analysis. Section 5 introduces attack overview and the control-flow and data-flow primitives, and Section 6 evaluates the four case studies. We discuss mitigations in Section 7 and related work in Section 8. Section 9 concludes.

Availability. The source code of proof-of-concept code and measurement scripts are open-sourced at `https://github.com/cispa/StackWarp`.

Responsible Disclosure. We responsibly disclosed our findings to AMD on March 24, 2025, and received acknowledgment on March 25. AMD assigned CVE-2025-29943 and embargoed the findings until January 15, 2026. Prior to the public disclosure, AMD confirmed that hot-loadable microcode patches have already been released to their customers.

2 Background

This section covers the background information relevant to the paper.

2.1 Trusted Execution Environment

Trusted Execution Environments (TEEs) are hardware-based mechanisms that provide isolated execution contexts to protect sensitive code and data from higher-privileged software, such as operating systems and hypervisors. TEEs ensure confidentiality and integrity by restricting access to specific memory regions and preventing interference from external components during execution within the secure context. Representative implementations include Intel Software Guard Extensions (SGX), which isolates user-level enclaves, and ARM TrustZone, which partitions system resources into secure and non-secure worlds. These technologies are widely used to safeguard cryptographic operations, digital rights management, and secure key storage.

Confidential Virtual Machines (CVMs) extend the TEE concept to entire virtual machines, enabling secure execution of complete operating systems on potentially untrusted platforms. Technologies such as AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP) and Intel Trust Domain Extensions (TDX) provide hardware-enforced isolation by encrypting guest memory and performing integrity checks to prevent tampering by the host. CVMs require minimal modification to guest software and are particularly suited to cloud computing environments, where tenants must protect workloads from the underlying infrastructure. By securing the full virtual machine context, CVMs offer a scalable solution for confidential computing in multi-tenant systems.

2.2 Software-based Fault Attacks

Software-based fault attacks induce errors through software-accessible mechanisms, without physical access, to break integrity guarantees. Well-known generic software-based fault attacks include DRAM disturbance (Rowhammer [21]) and CPU undervolting [22–24]. In Rowhammer attacks, an attacker repeatedly accesses a DRAM location, leading to bit flips in adjacent non-attacker-controlled memory locations. In contrast, undervolting affects the CPU, leading to incorrect computations, e.g., bit flips in multiplication results. In CVMs, recent work shows that faults can be injected architecturally from the host [4–6]. CacheWarp [4] allows a malicious hypervisor to revert selected dirty cache lines to an older value at attacker-chosen points, causing the guest to continue with architecturally stale data. Related to software-based fault attacks are Ahoi attacks [5, 6]. These attacks inject interrupts into confidential VMs to cause a state change.

2.3 Hyperthreading

Hyperthreading, also known as Simultaneous Multithreading (SMT), enables a single physical CPU core to execute multiple threads simultaneously. Throughout this paper, we unify Intel's Hyper-Threading Technology (HT) [25] and AMD's Simultaneous Multithreading (SMT) [26] under the term *hyperthreading* for the remainder of this paper.

In hyperthreaded CPUs, each physical core is exposed as two logical *sibling cores* to the operating system, allowing two concurrent *sibling threads* to share a single physical core. While each sibling core maintains distinct architectural contexts, such as register files and instruction pointers, they share essential execution resources, including arithmetic logic units (ALUs), floating-point units (FPUs), caches, and memory bandwidth. This resource-sharing model can increase resource utilization by enabling threads to use execution units that would otherwise be idle due to pipeline stalls or waiting periods. However, shared resources can also lead to contention among threads, potentially degrading performance and introducing security risks, notably through side channels [12, 27].

2.4 Stack Engine

Modern x86 CPUs rely on a deeply pipelined architecture. The pipeline's *frontend*, which decomposes instructions into simpler micro-operations (μops), plays a crucial role in this. It consists of several tightly integrated components, e.g., instruction fetch unit, branch predictor, decoders, and μop queue, all orchestrated to supply a continuous stream of μops to the out-of-order (OoO) execution engine. A notable optimization within the frontend is the stack engine, which first appeared in Intel Pentium M CPUs. The stack engine handles stack-related instructions like `push`, `pop`, `call`, and `ret`, which frequently modify the stack pointer (`esp` or `rsp`). Without the stack engine, these instructions would typically break down into multiple μops, burdening general-purpose ALUs and the scheduler.

To avoid this overhead, the stack engine simplifies stack operations early in the pipeline. During decoding, it tracks an 8-bit delta corresponding to stack pointer changes, maintaining an offset that helps dependent instructions proceed without delay [28]. This mechanism enables stack instructions to execute with zero apparent latency and only in a single μop. AMD CPUs have adopted similar techniques following cross-licensing agreements, resulting in a convergence of stack engine behavior across major x86 vendors [17].

3 StackWarp: Manipulating the Stack Pointer via the Stack Engine

In this section, we introduce StackWarp, a software-exploitable vulnerability in the stack engine that enables attackers to manipulate the stack pointer inside an SEV-SNP guest, fully breaking the integrity guarantees. In Section 3.1, we present an overview of StackWarp. Section 3.3 describes the automated approach that discovered StackWarp.

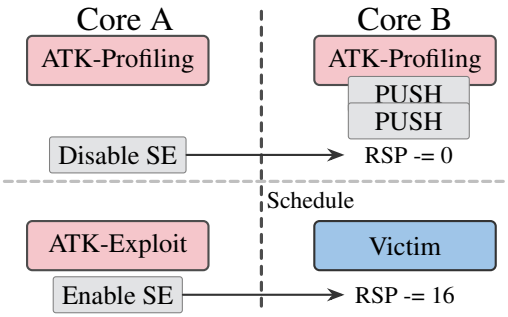


Figure 1: Illustration of StackWarp

CPU	Stepping	Microcode	μarch.	Sync Bug	StackWarp
AMD Ryzen 5 2500U	0	0x810100b	Zen	✓	†
AMD Ryzen 5 3550H	1	0x8108102	Zen+	✓	†
AMD EPYC 7252	0	0x830107c	Zen 2	✓	✓
AMD Ryzen 7 5700G	0	0xa50000d	Zen 3	✓	†
AMD Ryzen 9 6900HX	1	0xa404102	Zen 3+	✓	†
AMD Ryzen 7 PRO 7840U	1	0xa704107	Zen 4	✓	†
AMD EPYC 9124	1	0xa101148	Zen 4	✓	✓
AMD Ryzen 9 9950X	0	0xb404006	Zen 5	✓	†

Table 1: Tested CPUs affected by StackWarp. *Sync Bug* confirms the stack engine synchronization issue. *StackWarp* indicates SEV support (required for exploitation); † denotes non-applicability.

3.1 Overview

StackWarp is a software-only integrity breach against AMD SEV-SNP guests that exploits a cross-hyperthread control of the stack engine on Zen CPUs. At a high level, a malicious hypervisor running on one logical thread of a physical core toggles an undocumented, core-scoped MSR bit that enables/disables the stack engine. The stack engine tracks a hidden running delta for `rsp` so that stack instructions retire efficiently. We find that its enable state is not properly synchronized across sibling threads (Figure 1). If this toggle occurs while the sibling thread (the victim SEV-SNP vCPU) has stack operations in flight (`push`/`pop`/`call`/`ret`), the memory side of the stack instruction completes but the architectural update of `rsp` is deferred and later "released" in one step. This mis-synchronization yields a deterministic, attacker-chosen shift of the victim's stack pointer by $\pm\Delta$, without injecting interrupts, observing plaintext, or modifying guest code. A single injection can: (i) redirect control flow by returning to an earlier frame or ROP-style, to attacker-controlled addresses, and (ii) manipulate on-stack variables while preserving overall stack depth (e.g., drop writes or read stale data).

The attack proceeds in two short phases as we detail in Section 5: In the *profiling phase* on the host thread, StackWarp creates a desired Δ using controlled `push`/`pop` sequences while the sibling flips the MSR. In the *injection phase*, the sibling issues a single `wrmsr` while the guest executes a targeted snippet so the stack engine state change lands between the guest's VM-entry load and its next retirement, shifting `rsp` by $\pm\Delta$ at instruction-level precision.

3.2 Threat Model

We consider a cloud setting in which a victim-controlled workload executes inside a *Confidential Virtual Machine (CVM)* launched with AMD SEV-SNP. All guest memory pages, CPU registers saved in the VMSA STATE, and attestation reports are protected by the respective hardware so that a correct host cannot observe plaintext or tamper with the state.Importantly, hyperthreading is enabled, so each physical core exposes two sibling cores that may be scheduled for unrelated tenants.

Attacker. In line with the threat model of SEV-SNP, the attacker controls the hypervisor. The attacker can execute privileged instructions, including `wrmsr`, `rdmsr`, and other ring-0 operations that affect core configuration. The attacker is further in control of when and where each virtual CPU runs, can pin a victim vCPU to a logical thread of their choice, and may offline or idle the sibling thread of any core. Finally, the attacker can trigger VM exits, inject interrupts, read performance counters, and manipulate nested page–table entries.However, the attacker *cannot* read or write guest memory, alter guest register contents in the encrypted save area, or insert code into the guest. We do not assume knowledge of guest virtual or guest physical addresses for specific variables or functions. Any information the attack requires must be inferred at run time through architectural events that remain visible to the host.

Attack surface. The only shared resources between attacker and victim are those inherently exposed by simultaneous multithreading: model-specific registers (MSRs), certain microarchitectural units, and parts of the cache hierarchy. We use StackWarp to refer to the synchronization issue between two SMT threads. We further clarify that due to MSR writes requiring a privileged attacker, StackWarp only becomes a security vulnerability in the context of a TEE/SEV scenario. All other cross-VM communication channels (e.g., DMA, shared memory, paravirtual devices) are outside our scope and may be disabled or passed-through as usual.

We do not consider physical attacks, such as voltage or clock glitching [29,30], exploiting DRAM modules [21,31] in-scope. Also, side-channel attacks that merely *observe* victim activity (e.g., cache timing) are orthogonal to our work.

3.3 StackWarp Discovery

Our search for StackWarp is inspired by *MSR templating* as introduced by Kogler et al. [32]: systematically flipping MSR bits while observing how instruction behavior changes. We build on the approach described by Kogler et al. [32] and adapt it to a hyperthreaded setting with an SEV-SNP guest

on the sibling thread. Our key twist is that all *writes* to candidate MSRs happen on one hyperthread in the host, while the *measurements* run inside an SEV-SNP VM pinned to the sibling hyperthread. We then watch for *architectural* deviations inside the guest. Hard crashes are treated as noise, as it is trivial to crash machines with invalid MSRs configurations. In such a case, the harness restarts and continues the scan. We still provide a collection of crashing MSRs in Section A for future work to investigate if such crashes might be relevant in a cloud scenario, e.g., to cause denial of service.

3.3.1 Templating

We pin the victim SEV-SNP vCPU to logical thread C_{sev} of a physical core, and run a privileged host templating agent on the sibling thread C_a. The agent iterates over the MSR space and enumerates writable MSR bits as done by MSRevelio [32]. To avoid immediate system crashes, we exclude documented bits known to cause system instability, previously verified as crash-inducing [33]. Additionally, we exclude bits that are write-once after boot, i.e., bits that cannot be reset to their original state once modified.

Host Agent We use a kernel module for modifying MSRs on the host, i.e., hypervisor. In the kernel module, a kernel thread is started on logical thread C_a to flip all bits in a given MSR according to a user-provided bitmask. Depending on the test configuration, the module either flips the bits once or continuously in a tight loop. Before flipping an MSR, the guest state is reset to a clean state. The host receives the info whether the guest crashed or continued execution, and additionally, whether any of the test cases inside the guest return an unexpected result. If a flip causes a guest crash or host hang, we log the MSR/bit and immediately restart. Such events are not considered exploitable for our purposes.

Guest Probe The guest SEV-SNP VM is pinned to core C_{sev}. Inside the guest, we execute short, deterministic snippets that (i) stress specific instruction groups and (ii) record both register and memory state.If a test case produces a different program state with changed MSR settings or while flipping MSR bits in the hyperthread, we flag the corresponding MSR bitmask and include it in our test report. We also record faults in the user-mode program and crashes of the host system.

To achieve broad coverage over the x86 ISA, we run test cases for every non-privileged instruction, excluding non-deterministic instructions such as `rdrand`. Furthermore, we create test cases to check the integrity of both data and control flow structures. For data flow, we test memory accesses on data residing in the L1d, L2, and L3 caches, or DRAM, using `mov` instructions, stack instructions like `push` and *pop*, and string moves like `rep movsb`. For control flow, we test direct and indirect branches, as well as unconventional control flow structures like deeply nested function calls, self-modifying code, and direct manipulation of return pointers on the stack.

We run all these test cases with various MSR settings, covering all previously-identified MSR bits. This includes settings where all bits in an MSR are set to '0' or '1', and settings with the MSR in its default state, with individual bits flipped. Additionally, we run the tests while continuously flipping all or individual MSR bits in the hyperthread.

3.3.2 Results

We use a Zen 4 CPU (AMD EPYC 9124; microcode 0xa101148) running a 6.11.0 host kernel and a v6.10.0 guest kernel for the VMs. We use QEMU version 9.0.92, provided by AMD. Running the templating on this machine with hyperthreading enabled results in a single undocumented, core-scoped bit in `MSR 0xC0011029` whose *transitions* on C_a repeatedly caused architectural deviations in stack-centric snippets running on C_{sev}. We consistently observe wrong returns on the guest, indicating a corruption of the stack pointer. As these effects vanish when the sibling thread idles or the guest avoids stack-modifying instructions, we flagged the bit as a cross-hyperthread control for the frontend stack machinery. As flipping the bit allows modifying the stack pointer, we refer to this primitive as StackWarp. We further confirm that this bit has the same effect on all Zen 1-5 generation CPUs, as listed in Table 1. Section 4 attributes the phenomenon to the stack engine and quantifies the induced, deterministic shift of the guest `rsp`. In contrast, on Intel processors, the scan results only in crashes without exploitable findings (Section 8.1).

Crashes Some flips (across unrelated MSRs) stall the core or raise a machine error check. We provide a list of such MSR bits in Section A. In such cases, we reboot the machine and continue scanning. We explicitly ignore such cases in the discovery pipeline, as they are not directly exploitable under our threat model and would only distort the signal of guest-visible architectural changes we seek.

4 Root Cause Analysis

In this section, we analyze the root cause of StackWarp. We attribute StackWarp to a frontend bug in the Zen *stack engine* that can be toggled via an undocumented, per-core MSR control. We present the positive findings that establish a deterministic, controllable shift of `rsp`, characterize its behavior on the μop level, show cross-hyperthread controllability, and bound the offset. We also report the negative results that rule out alternative explanations and scope the effect.

4.1 Deterministic Shift and Freeze–Release Model

We observe that transitions of the per-core control in MSR (`0xC0011029`) cause a repeatable, attacker-chosen shift of the victim's `rsp`. If flipping the bit from 0→1 causes a

stack corruption, flipping it back from 1→0 produces a corruption of the *same magnitude* with the *opposite sign* (Section B). This behavior matches a simple model. While enabled, the stack engine tracks a hidden running delta Δ for `rsp`. In contrast, when the stack engine is disabled, every stack-manipulating instruction has to adjust the `rsp`. However, due to the incorrect synchronization, stack-manipulating instructions that are already decoded *assume* that they do not have to adjust the `rsp`. Thus, disabling the stack engine while stack instructions are in flight *freezes* Δ so memory updates commit but the architectural `rsp` is not advanced. Re-enabling *releases* the accumulated Δ in one step:

$$\texttt{rsp}' = \texttt{rsp} \pm \Delta,$$

with sign and magnitude determined by the in-flight mix of stack-manipulating instructions (i.e., `push`/`pop`/`call`/`ret`) at the transition. This freeze–release effect lets us prepare Δ on the host, then deterministically inject it into the guest as in Section 5.1. The behavior reproduces across all CPUs in Table 1, including fully patched Zen 5. While microarchitectural details may vary by stepping, the freeze–release model consistently explains the observed direction, magnitude, and determinism required for exploitation.

4.2 Instruction-Level Evidence via Performance Counters

To verify that stack-manipulating instructions do not correctly update the stack pointer, we analyze the number of μops with the μop cache [34] disabled. Disabling the μop cache ensures the CPU re-decodes instructions, preventing the observation of cached (stale) results. This serves primarily as a de-noising measure and does not fundamentally alter the experimental outcome. We use two performance counter events, issued μops from the decoder and retired μops, to analyze instructions that interact with the stack, including both explicit and implicit stack operations. We monitor these events while toggling *bit 19* of MSR (`0xC0011029`).

Table 2 summarizes the results. When the bit indicates "stack engine enabled", i.e., the default setting, stack instructions use fewer μops and explicit `rsp` arithmetic (`add/sub/mov rsp,*`) incurs additional synchronization μops, consistent with periodically committing a hidden delta [17]. When disabling the stack engine, `push`/`pop`/`call`/`ret` expand to multiple μops reflecting their true complexity, i.e., they require additional μops to update the stack pointer.

For verification, we analyze the μops for unrelated arithmetic on other registers (e.g., `sub rdi, imm8`), including the base pointer. In all of the tested cases, the number of μops does not change, indicating that this MSR bit indeed controls the stack engine. These effects are expected if the stack engine injects a sync μop that behaves like an ALU add to `rsp` and is dispatched at the frontend.

Table 2: Performance counter analysis on selected instructions. The table records the counter values when bit 19 of MSR (`0xc0011029`) is set to 0 vs. 1. The experiment is conducted on an AMD EPYC 9124 CPU.

	Instructions	Ops from Decoder		Retired Ops	
		Bit=0	Bit=1	Bit=0	Bit=1
Stack (impl.)	`push REG`	1	2	1	2
	`pop REG`	1	2	1	2
	`call+ret`	5	11	5	11
	`mfence`	6	5	6	5
Stack (expl.)	`mov rsp, REG`	3	1	2	1
	`add rsp, IMM8`	3	1	2	1
	`sub rsp, IMM8`	3	1	2	1
No stack	`sub rbp, IMM8`	1	1	1	1
	`sub rdi, IMM8`	1	1	1	1

4.3 Cross-Hyperthread Control from a Per-Core MSR

The control in MSR (`0xC0011029`) is per-core. When we idle the sibling logical thread, effects disappear. When the sibling flips the bit while the victim executes stack instructions, the victim's `rsp` shifts according to our model.This establishes a cross-hyperthread control path: the enable state is not correctly synchronized across logical threads sharing the core frontend. That a stack-engine sync is essentially an ALU μop also clarifies why cross-thread influence is observable in principle, ALU resources are shared across hyperthreads [35].

4.4 Bounds and Accumulation of Δ

Two factors bound |Δ|: the maximum value of the hidden delta tracked by the stack engine, and the number of stack-modifying instructions decoded but not yet retired at the transition. Prior work on Pentium M documents an 8-bit signed counter with periodic sync [28]. On AMD EPYC 9124, we measure single-shot |Δ| up to 640 B in either direction using sequences of `push` or `pop`. While this offset seemingly exceeds the range of an 8-bit signed counter, we expect that the stack engine tracks the Δ in multiples of the stack-pointer alignment, i.e., 8 bytes. Thus, we expect the theoretical maximum shift to be 1016 B, which seems to be practically limited by number of in-flight stack operations in the speculation window. Fences and explicit `rsp` arithmetic introduce backend synchronization (Table 2) and thus cap |Δ|. Still, the effective limit can be extended by relying on multiple injections. Several small, precisely placed shifts are often easier to achieve than a single large one and are sufficient for our exploits in Sections 5 and 6.

4.5 Rejecting Alternative Hypotheses

To further validate that StackWarp exploits the stack engine's optimization for the stack-pointer offset, we disprove alternative hypotheses.

Instruction Skipping. We design a test program that first pushes eight zeros, resets `rsp`, and then pushes eight non-zero values with `mfence` padding. For all executions, we find all eight non-zero values in memory, even after a crash. Thus, `push` reaches memory, and only the architectural `rsp` update is dropped. Hence, there is no instruction skipping.

Decode Failure. The decoder and retire counts change exactly for stack instructions and explicit `rsp` arithmetic when toggling bit 19. Unrelated GPR arithmetic remains stable (Table 2), which is inconsistent with general decoding failure. Moreover, when using this control with an idle hyperthread, programs work as expected.

Generic Register Optimization. We observe no hidden-delta behavior for `rbp` or other GPRs. Arithmetic on `rbp` (e.g., `sub rbp, imm8`) shows identical μop counts regardless of the control state (Table 2). Moreover, in our tests, flipping the control caused crashes only in code containing stack-manipulation operations.

5 Attack Primitives

In this section, we detail what an attacker can do with StackWarp, and how the resulting attack primitives can be used for end-to-end exploits. First, we show how a malicious hypervisor prepares and triggers a precise stack-pointer shift against a victim vCPU (Section 5.1). Second, we present control-flow and data-flow manipulation primitives that enable attacks, such as the examples given in Section 6. Eventually, we evaluate StackWarp's stability and reliability by a series of experiments with fine-granular synchronization between attacker and victim in Section 5.4.

5.1 Attack Flow

Figure 2 summarizes the attack, which proceeds in two phases on two sibling logical threads C_{sev} (victim vCPU) and C_a (attacker host thread) that share one physical core. The *profiling phase* disables the stack engine while concurrently executing stack instructions to configure an attacker-chosen Δ. The *injection phase* injects that exact Δ into the victim so it deterministically shifts the guest `rsp`, leading to a controlled stack manipulation of the victim application.

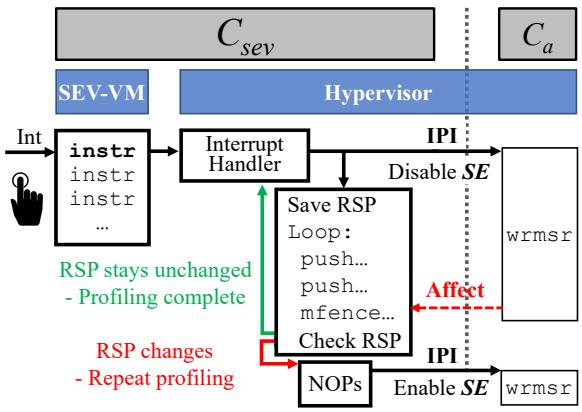
5.1.1 Profiling Phase: Generate and Verify Δ

The goal of this phase is to create and *freeze* a desired offset Δ in the stack engine before injecting it into the guest. For this phase (cf. Figure 2a), the victim is paused, i.e., not running, and the attacker runs code on both sibling cores, i.e., C_a and C_{sev}. On sibling core C_{sev}, the host executes a fenced sequence of `push` or `pop` to accumulate a hidden offset in the stack engine while it is enabled. Concurrently, on C_a, the host issues a `wrmsr` to the undocumented, per-core control in MSR (`0xC0011029`) to *disable* the stack engine. If the transition hits while the stack sequence on C_{sev} is decoded but not yet retired, memory updates commit but the architectural update of `rsp` is withheld, i.e., the hidden offset Δ is *frozen* (Section 4.1). The sign of Δ is controlled by using `push` or `pop` instructions, for a negative and positive sign, respectively. The absolute value of Δ depends on the exact number of `push` and `pop` instructions. We *verify* the freeze locally on C_{sev}: if `rsp` remains unchanged after the sequence, the accumulated value equals the generated Δ, as the expected stack change is withheld. If `rsp` does not contain the expected value, we restore `rsp`, re-enable the stack engine from C_a, and repeat. This closed-loop procedure yields a Δ with known sign and value and leaves it frozen in the core frontend until released. To reduce noise, we fence the sequence, pin threads, and keep the pipeline clean when re-enabling the stack engine. If the desired Δ is successfully frozen in the stack engine, the attack moves to the *injection phase*. This phase is deterministic and completes within tens of thousands of cycles (cf. Section 5.4).

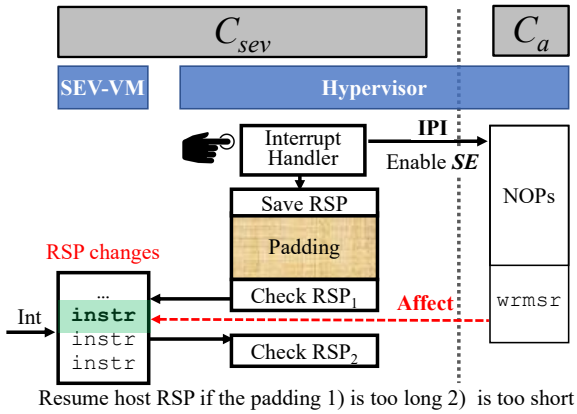
5.1.2 Injection Phase: Inject Δ into the Victim

With a verified Δ frozen, we schedule the victim on C_{sev} at a chosen snippet and *release* Δ by re-*enabling* the stack engine from C_a. The goal is for the `wrmsr` on C_a to take effect after VM-entry loads the guest state on C_{sev} but before the next retirement, so the guest observes `rsp` $\leftarrow$ `rsp` $\pm\Delta$ at the next instruction. We achieve this with a short, calibrated padding loop on C_{sev} followed by an inter-processor interrupt (IPI) to C_a that performs the `wrmsr`. The calibration of the padding sequence is required only once. As illustrated in Figure 2b, the CPU loads the guest state before resuming the VM and saves it upon exiting. The attacker synchronizes the `wrmsr` instruction on C_a to execute precisely within this window, ensuring it targets the guest `rsp`. Calibration relies on checking the host `rsp` before and after the context switch. These checks indicate whether the padding is too short or too long, namely if the Δ is injected to the host `rsp`.

To reduce jitter, we take several measures. We fix the core frequency. Additionally, a userspace thread is pinned to C_a to keep it in the P0 (active) state, thus preventing the core from entering the idle state. As the IPI has the highest priority except for NMI or SMI, C_a immediately executes the function specified by the IPI. The attacker also flushes the cache using `wbinvd` before resuming the VM, extending the duration of each state in the context switch and thus widening the timing window. Lastly, to further reduce noise in the IPI routine, the attacker sends an IPI from C_{sev} to C_a tasking C_a to execute a warm-up function and bringing the relevant code path into the cache. This warm-up step is performed before resuming



(a) Profiling phase. The attacker disables the stack engine on C_a while executing a fenced `push`/`pop` sequence on C_{sev} to freeze a chosen Δ.



(b) Injection phase. The attacker re-enables the stack engine from C_a so the release lands the next retirement on C_{sev}. The green region marks the effective injection window.

Figure 2: Overview of StackWarp. Freeze–release of the stack-engine delta yields a deterministic shift of the victim's `rsp` (Section 4).

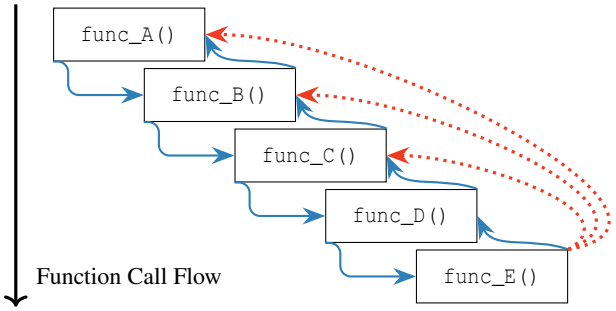


Figure 3: Function call flow. By shifting `rsp` to a previous stack frame before a `ret`, an attacker can skip function epilogues or entire functions.

the VM and issuing the IPI that triggers the `wrmsr` on C_a. As also shown in previous work [4], single-stepping yields consistent interruption points where we can manipulate the stack pointer. The same process supports multi-shot injections by regenerating and reusing additional Δ values as needed for larger effective shifts.

5.2 Control-Flow Primitives

We describe two control-flow building blocks allowing to hijack the control flow using StackWarp. These building blocks do not rely on gadgets as in traditional exploits.

(C1) Frame Rewind. As return addresses are placed right at the beginning of each function's stack frame, moving `rsp` to a the beginning of a *previous* instead of the current stack frame, right before a `ret` instruction causes a return to an older call site without altering any other architectural state. This lets an attacker skip epilogues or entire functions and is effective even without attacker-controlled stack data. This primitive is illustrated in Figure 3. For example, an attacker exploiting StackWarp right before the end of the execution of `func_E` can skip the remaining execution of `func_D` and instead directly return to `func_C`, `func_B`, or `func_A`. Assuming that the attacker aims to skip the remaining code of `func_D` and instead return to `func_C` directly, the attacker mounts StackWarp right before the return from `func_E`. The attacker then shifts the stack pointer to point to the stored return address of `func_C`, instead of the saved return address within `func_D`. The attacker resumes the victim application, which skips the remaining code of `func_D`, while returning the value produced by `func_E`.

This building block can also be used to "tunnel" return values. Consider a simple string comparison as shown in Listing 1. It uses C's `strcmp` function to compare a user-provided password against the actual password. The C standard defines `strcmp` to return `0` on equality and *non-zero* on mismatch [36]. Glibc's implementation internally calls one of many different implementations of `strcmp`, e.g., `__strcmp_avx2`. By injecting Δ right before `__strcmp_avx2` returns and rewinding by one frame, control returns *directly* to `authenticate_user`, bypassing `check_password`'s wrapper that translates `strcmp == 0` to a boolean `true`. Since C treats any non-zero as `true` [36], a mismatching `strcmp` result in `RAX` is interpreted as success by `authenticate_user`, granting access without modifying code or data in the guest.

(C2) Stack Pivot to Attacker-controlled Data. If the victim has an in-stack buffer that the attacker can influence (e.g., via syscalls that copy user data onto the kernel stack, see Section 6.4 for an example), moving `rsp` to that buffer imme-

```c
int check_password(const char* user_input) {
  if (strcmp(user_input, correct_passwd) == 0)
    return 1; // Password match
  return 0; // Password mismatch
}

void authenticate_user() {
  const char* user_pwd = read_user_input();
  if (check_password(user_pwd))
      allow_access();
  else
      deny_access();
}
```

Listing 1: Return-value tunneling: an attacker can shift `rsp` by $+8$ just before `strcmp` returns so control hops over `check_password`'s boolean wrapper and resumes in `authenticate_user` with a non-zero `RAX`, which is treated as `true`.

diately before a `ret` pivots control flow to an attacker-chosen address. This can be exploited ROP-style for arbitrary code execution in the victim. Unlike classical overflows, no memory corruption is required. The stack pointer alone is redirected, and stack depth can later be restored with a second injection.

5.3 Data-Flow Primitives

As StackWarp allows for controlled bidirectional stack pointer manipulation, an attacker can use it to manipulate a victim application's data flow. More precisely, an attacker gains the ability to manipulate memory accesses to stack-based variables. We describe two generic building blocks that leverage StackWarp for data-flow attacks.

(D1) Balanced Two-shot Manipulation within a Frame. With small $|\Delta|$ that stays inside the current frame, we can transiently relocate `rsp` to change which slots are read or written between two injections. This enables targeted corruption of security-critical metadata such as stack canaries, potentially altering control flow without triggering immediate faults.

Figure 4 shows a toy example with a stack-based buffer overflow, and the compiler-inserted stack-canary protection. During the time window marked by the red rectangle, the attacker uses StackWarp to shift the stack pointer upward by 0x18 bytes, allowing input to overwrite the original return address. After the canary value is read from the stack, during the time window marked by the blue rectangle, the attacker shifts the `rsp` back to its original position to maintain stack balance. This sequence enables overwriting the return address with attacker-controlled data while still passing the canary check, demonstrating that compiler-inserted mitigations can be bypassed without disrupting normal control flow. Note that this serves as an illustration and the access to other stack-based variables can be manipulated as well.

```c
// Compile with
  -O2 and
// -fstack-
  protector

int main()
{
  char buf[16];
  scanf("\%s",
  buf);

  if (!strcmp(
  buf, "AAA"))
    puts("Win");

  return 0;
}
```

```
00000000000010a0 <main>:
10a4:   sub     $0x28,%rsp
10a8:   lea     0xf55(%rip),%rdi
# Stack Canary
10af:   mov     %fs:0x28,%rax
10b8:   mov     %rax,0x18(%rsp)
10bd:   xor     %eax,%eax
... ...
# Check stack smashing
10dc:   mov     0x18(%rsp),%rax
10e1:   sub     %fs:0x28,%rax
10ea:   jne     10f3 <main+0x53>
10ec:   xor     %eax,%eax
10ee:   add     $0x28,%rsp
10f2:   ret
# goto stack_chk_fail
```

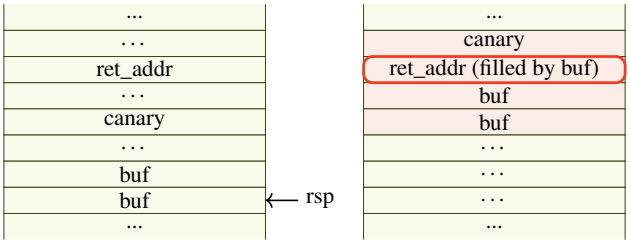


Figure 4: Bypassing stack canary via data flow manipulation. The attacker shifts the stack pointer upward to overwrite the return address, then restores it after the canary check, preserving stack balance while bypassing the protection.

(D2) Drop Writes / Read Stale. With a larger $|\Delta|$, stores intended for the current frame can land in an unused stack region, and the resulting memory accesses target addresses outside any active stack frame. In this case, writes are effectively discarded, and reads return stale or uninitialized values. Once the stack pointer is restored, the program resumes execution with this stale state, enabling stealthy state injection into subsequent computation. To demonstrate the drop-write capability, we provide a toy example Listing 5 in Appendix.

5.4 Evaluation

We use three practical ways to time the injection window. First, single-stepping via VM exits results in the highest precision. Second, if single-stepping is not feasible or necessary, we rely on a controlled-channel attack [37]. We simply unmap a code page that is close to the point at which we want to manipulate the stack pointer. This coarse-grained synchronization is easy to use and in many cases sufficient. Third, as a fallback when neither controlled-channel attacks nor single stepping is possible, we rely on coarse-grained timing with retries. This is sufficient if the point of the injection is not critical, as e.g., in our RSA key-recovery attack (cf. Section 6.1).

In this section, we use the single-stepping approach, as it provides the finest granularity and best illustrates the stability of each fault injection and their reliability with our attack primitives. For each injection, the profiling phase takes less than 100 000 CPU cycles to obtain the attacker-chosen offset. In the attack phase, our experiments show that a padding range of 4000 to 10 000 iterations on C_a allows the `wrmsr` to reliably affect the guest VM state. For our evaluation on an AMD EPYC 9124, we use 7000 iterations and apply single-stepping to interrupt the VM, ensuring that the injection occurred at a precise location. To confirm that fault injection works with single-stepping, we inspect the VM exit reason and the number of guest-retired instructions using performance counters. Across 100 trials, the attack successfully injects faults in 92 cases with single-stepping and in 3 cases with multi-stepping, all within the current guest code page. Only 4 trials result in a stack shift on the host, which resumes without any effect, and no trial causes a crash on either the host or the guest. Interestingly, we are not able to inject a fault with zero-stepping. However, this is not a limitation, as an attacker can simply repeat the profiling and retry the attack.

Next, we evaluate our attack primitives with single-stepping. Using the primitive of C1, we bypass the `strcmp` check in 44 out of 50 trials, yielding in a success rate of 88 %. With the toy example of D2, we shift the stack twice to balance it, and the offset is `-0x80` to an unused frame. The distance between two injections is only one single step. Out of 45 out of 50 trials, the injection successfully points the password to NULL and bypasses the check, yielding a success rate of 90 %. While all failing attempts lead to segmentation faults, they do not destabilize the guest VM or host system, thus enabling retries for an attacker.

6 Case Studies

In this section, we demonstrate StackWarp in 4 case studies. Section 6.1 demonstrates an RSA key-recovery attack. Section 6.2 leverages StackWarp to log into a guest VM via SSH without requiring a password. Section 6.3 uses `sudo` to gain root privileges. Similarly, Section 6.4 shows how StackWarp can be used on the kernel stack to escalate user privileges to kernel-mode arbitrary code execution.

6.1 RSA Key Recovery

In this case study, we demonstrate how an attacker can use StackWarp to recover the private key of an RSA-CRT implementation. Specifically, we use StackWarp to mount a Bellcore attack on an RSA signature [19, 20].

Background An RSA signatures compute $S = x^d \pmod N$, with x the (hash of the) message, d the private exponent, and N the public modulus. As the involved numbers are large, the operation is costly. An optimization, RSA-CRT, speeds up the computation using the Chinese Remainder Theorem (CRT) [38]. It splits the exponentiation into $S_1 = x^d \pmod p$ and $S_2 = x^d \pmod q$, where $N = p \cdot q$, and then recombines them into S.

However, this optimization creates an attack surface if any of the computations can be corrupted, and the same message can be signed twice. For example, if an attacker corrupts only S_1 while S_2 stays correct, the faulty signature $\hat{S}$ leaks the factors of N. Given S, i.e., the non-corrupted signature, and $\hat{S}$ on the same message, the attacker computes $q = \gcd(S - \hat{S}, N)$ and consequently also $p = N/q$. Thus, with a single fault, an attacker can trivially reconstruct the private key.

Threat Model We adopt the AMD SEV model: the victim is a protected VM guest on a malicious hypervisor. The VM offers a signing API that allows an attacker to sign messages with the private key inaccessible to the attacker. We assume only that the same message can be signed twice. The attacker neither chooses nor knows the data. The attacker only relies on StackWarp, and no other hardware or software vulnerability, or any side channel. The victim uses Intel's IPP RSA-CRT implementation, which is the same implementation that has previously been used in fault attacks [22, 39, 40]. The attacker's goal is to recover the private signing key.

Attack Flow The attacker first collects a correct signature S. After that, the attacker triggers another signing run and injects a fault using StackWarp during the computation of S_1, producing $\hat{S}$. We induce this fault by shifting the stack during the CRT step via StackWarp. While the type of fault does not matter much, it must affect only S_1, and the victim must not crash. Both are achievable by injecting at a calibrated cycle offset, requiring only coarse timer-based synchronization.

As the attacker runs on the same system and knows the library in use, the attacker can benchmark the victim's code and pick the right injection point. The Bellcore attack is robust enough to handle noise from cache activity or frequency scaling, so timing jitter does not break the exploit.

Results We successfully mount the Bellcore attack against an AMD SEV-SNP-protected VM using Intel's IPP RSA-CRT implementation. Our evaluation follows the 'blind' fault injection primitive described by Zhang et al. [4] where any faulted signature can lead to key recovery, regardless of the specific fault. We add the offset `+0x18` to the `rsp` after a randomly chosen 1111 single-steps. Out of 100 trials, 49 attempts produced a usable faulty signature, whereas 51 attempts result in a segmentation fault or no observable effect. None of the attempts crash the VM or the host system. Note that, the reliability could be improved with program analysis to find a stable gadget, but the blind attack itself is a valid threat.

```
1  // return 0 on failture,
2  // else authentication succeeds
3  int sys_auth_passwd(struct ssh* state,
4      const char* password) {
5   pw_ptr = retrieve_user_pw_entry(state)
6   char *hash_real_pw = shadow_pw(pw_entry_ptr);
7   if (hash_real_pw == NULL) return 0;
8   salt = hash_real_pw; // salt from stored hash
9   hashed_provided_pw = xcrypt(password, salt);
10  return strcmp(hash_real_pw, hash_provided_pw) == 0;
11 }
```

Listing 2: Simplified version of OpenSSH's `sys_auth_passwd`. The function is used to authenticate users using their password

Each of the faulted signatures enables full recovery of the RSA private key, yielding an effective success rate of 100 %. The average time per attempt is below 0.1 s, making this a realistic attack. In all successful cases, key recovery is immediate, 0.1 s once a faulty signature is obtained.

These experiments confirm that the Bellcore attack remains practical in modern SEV-SNP environments. The robustness of the attack model—requiring only a single exploitable fault—means that even imprecise injections are sufficient. Compared to other software-based fault mechanisms (e.g., undervolting [22] or Rowhammer), our primitive does not rely on hardware-specific behavior and works reliably across different test systems, including Zen 4 and Zen 5 CPUs.

6.2 OpenSSH

In this case study, we demonstrate how StackWarp can be used to gain remote code execution on an AMD SEV-SNP protected VM running an OpenSSH server. We use StackWarp to bypass the password authentication check in OpenSSH 8.9p1, allowing an attacker to log into an account without knowing the password. The attack is based on the Frame Rewind primitive introduced in Section 5.

Attack Flow. When a user connects to an OpenSSH server, the user's client is prompted for an authentication method, such as a password or a public key. Passwords are verified in OpenSSH's function `sys_auth_passwd` (illustrated in Listing 2), which we manipulate using StackWarp to always return success. The function `sys_auth_passwd` gets the session state and user-provided password as arguments and returns `0` on failure. Note that due to how C handles boolean values, any non-zero return value is treated as `true`, thus indicating successful authentication, as the return value of `sys_auth_passwd` is only used in boolean contexts.

We redirect the control flow by changing the stack during the call to `shadow_pw`, the function responsible for retrieving the entry of the user in Linux's shadow password file. Right before the end of the function, between the last stack-accessing instruction and the final `ret`, we use StackWarp to shift the stack pointer to the frame of `sys_auth_passwd`. For this, a `RSP` modification of 32 bytes is sufficient. This causes `shadow_pw` to return directly to the caller of `sys_auth_passwd`. As `shadow_pw` returns with a non-zero value, it indicates success to `sys_auth_passwd`'s caller, effectively bypassing the password check.

Results. To evaluate the attack, we mount it against OpenSSH 8.9p1 running on an AMD SEV-SNP-protected VM on AMD EPYC 9124 running Linux kernel 6.11.0. We use page-fault sequences and PMCs to identify the target page, followed by single-stepping to pinpoint the instruction within that page [41]. The attack works to successfully bypass the password check in 70 % of the attempts ($n = 10$), with each completing within 8 seconds. In all other attempts, the OpenSSH server rejected the password due to occasional inaccuracies in PMC, which leads to pattern mismatches. None of the attempts crashed the system, as the attacker deliberately avoid injecting faults. This allows an attacker to retry until success, after, an expected amount of 2 attempts. We conclude that StackWarp can be used to exploit OpenSSH and thus, allowing malicious hypervisors to gain arbitrary code execution in AMD SEV-SNP VMs running OpenSSH servers.

6.3 Privilege Escalation with sudo

Finally, we use StackWarp to manipulate the return value of the `getuid` system call in Linux, allowing for privilege escalation with sudo. In contrast to the previous case studies, this attack does not manipulate control flow directly, targeting data in the stack frame of Linux's low-level system call handler function instead.

We assume unprivileged code execution in the AMD SEV VM (cf. Section 6.2). They can invoke sudo, but do not possess the privileges to execute commands as root (i.e., sudo denies these privileges). The attack does not require a vulnerability in either Linux or sudo.

Attack Flow. As a usability feature, sudo does not query for credentials when invoked by the root user, as root does not gain additional privileges with sudo. To determine this, it invokes the `getuid` system call. If the return value in `rax` is 0 (i.e., root's user ID), sudo immediately grants elevated privileges; If it is not, it queries the user for credentials or denies privileges. Hence, by manipulating the return value of `getuid` with StackWarp, we can escalate our privilege level.

We achieve this with a two-shot stack pointer manipulation (cf. D1 in Section 5.3) within Linux's system call handler. Before returning control to the user again, this handler restores the user's register state with a series of *pop* instructions. This includes the return value in the `rax` register, which the kernel injects by manipulating the register state in memory. We apply StackWarp to inject a stack offset of -8 before `pop rax` executes, and then resume stack balance with another injection on the next single step. This causes the instruction to restore the value of the previously restored register for a

```
1 ud->cred.uid = getuid();   // mov rax, <NR>
2 ud->cred.euid = geteuid(); // syscall
3 ud->cred.gid = getgid();   // ret
4 ud->cred.egid = getegid(); // mov [MEM], rax
```

Listing 3: Target code of `sudo`. The four assignment have the same pattern of fingerprints of performance counter.

second time, which for sudo fills `rax` with 0. When resuming user-mode execution, the `getuid` system call's return value is 0, regardless of the actual user ID. By mounting this attack on a command like `sudo -i`, we can obtain a root shell.

To locate the exact point at runtime, we combine single-stepping, controlled channel, and performance counter fingerprints. Listing 3 shows the target code of `getuid` inside sudo, followed with three additional syscalls. We select the *Retired Instruction*, and *Retired Far Control Transfers* performance counter events to track `syscall` and `sysret`, i.e., the return from a syscall. Once the syscall returns, the subsequent `ret` executes as a single retired instruction on the next code page. With the controlled channel, the following instruction triggers a data access page fault. We identify the target by detecting this distinctive pattern, which appears four times in the trace.

Results. We mount this attack on an AMD SEV-SNP-protected VM on AMD EPYC 9124 running Linux kernel 6.11.0. The guest is running Linux kernel 6.10.0 and uses sudo 1.9.9. The attack succeeds in 16 out of 20 attempts, each completing within 8 seconds. None of these instances crashes the host or the VM, thus making the attack highly practical, as an attacker can repeat the attack until it eventually succeeds.

6.4 Ring 0 Code Execution

Prior case studies show how to achieve unprivileged code execution, and privilege escalation with data-flow primitive in a AMD SEV guest. In this section we demonstrate privilege escalation to ring 0 (guest kernel) using the second control-flow primitive to pivot control flow via stack injection. We use return-oriented programming (ROP) techniques [42] to execute arbitrary code in the kernel. We assume that the attacker can break KASLR as demonstrated in previous work [5, 43–45]. We do not rely on any vulnerability in the Linux kernel nor a specific version.

Attack Flow. The attack flow is similar to classical ROP attacks [42] exploiting a stack buffer overflow. However, in contrast to these classical attacks, we do not overflow a data structure overwriting the return address. Instead, we use StackWarp to directly move the stack pointer to attacker-controlled stack data right before a `ret` instruction, thus redirecting control flow to an attacker-controlled address.

To mount this attack we need to bring attacker-controlled data onto the kernel stack. We manually inspect common Linux syscalls for stack data structures and buffers that are copied to by `copy_to_user`. We find the `select` syscall to be a good candidate as it allows for arbitrary 120 B to be copied directly to the kernel stack.

The Linux kernel `select` implementation uses a small buffer on the stack. This buffer holds 6 file descriptor sets, 3 of which are attacker-controlled and can contain arbitrary data. The buffer is 256 B in size by default which gives a maximum size of 40 B per set, as the used set size is rounded down to multiples of 8 B. Above that set size, `kvmalloc` is used, as the sets are too large for the limited stack buffer. To allow for 40 B sets, totalling 320 file descriptors, the userspace attacker process actually has to own more than 320 file descriptors, which we achieve by copying the `stdin` file descriptor 512 times using the `fcntl` syscall. Finally, we call `select` with 120 B of payload data split over the 3 input sets which are continuously copied to the stack.

Before returning to userspace from the `select` syscall, we use StackWarp to offset the stack pointer to the stack buffer holding the attacker data. Note that this offset is fixed in the compiled kernel and can therefore be probed offline. Shifting the stack pointer results in the kernel using the attacker-controlled bytes as return address. We craft a classical ROP chain that executes the `kernel_halt` guest kernel function. We shift the stack pointer by the correct offset within the kernel `select` function and observe a kernel guest halt on return.

Results. We evaluate the attack on an AMD Ryzen 7 PRO 7840U. In all of our experiments (10 out of 10 runs), the attack completed successfully without failure. We conclude that StackWarp allows for escalating privileges to ring 0 in the guest kernel, allowing an unprivileged attacker in the VM to arbitrarily take over the VM.

7 Mitigations

In this section, we discuss potential mitigations for StackWarp on the hardware, software, hypervisor, and firmware level.

7.1 Hardware and Microcode and Fixes

The clean solution is to ensure in firmware or hardware that the MSR exploited by StackWarp cannot be used. CPUs should either (i) remove software write access to bit 19 of MSR `0xC0011029`, or (ii) clear the stack-engine delta on every context switch even if the stack engine is disabled. Both strategies are low-risk firmware updates: the bit is undocumented, so no ABI breakage is expected. Moreover, this should be easily doable, as AMD also used such a microcode update that prevents modifying a specific bit inside an MSR for mitigating CacheWarp [4]. An even more conservative approach would be to introduce an additional lock bit for the MSR bit. Then, the UEFI could configure the bit and lock it to prevent further modifications. Additionally, it should only

be possible to start SEV-SNP guests if the lock bit is set. All these mitigation would result in no performance overhead.

7.2 Hypervisor

AMD announces an SEV feature called SMT protection, which forces the sibling thread to remain idle while an SEV-SNP VM is running [46].This feature is enabled by setting a specific bit in the Virtual Machine Save Area (VMSA) before launching the SEV-SNP VM. Its status is included in the attestation report, allowing an SEV-SNP user to verify whether the feature is enabled. In principle, SMT protection could mitigate our attack, as the MSR write has to be executed on the sibling thread. However, the feature is not yet supported in the upstream Linux KVM hypervisor by AMD, and we were therefore unable to test it.

7.3 Guest Software

While software countermeasures cannot stop the stack-pointer fault, they can turn it into a non-exploitable crash instead of an exploitable vulnerability:

Shadow Stack / CET. AMD Shadow Stack [47] is an ISA extension available on Zen CPUs that helps protecting against control-flow attacks. With Shadow Stack, the hardware compares the return address stored on the normal stack against a copy stored in the hardware. On a mismatch, an exception is raised and the application is stopped. While this does not prevent StackWarp, it prevents exploitation techniques where StackWarp is used to change the control flow. Thus, Shadow Stack would be effective against the ROP chain we demonstrate in Section 6.4. Unfortunately, Shadow Stack is ineffective against any other modification, such as the case studies in Section 5.3 and Section 6.3.

Frame-pointer Checks. As a heuristic countermeasure, a compiler pass can randomly insert checks that the difference between stack pointer and base pointer is not bigger than expected for the current function. This is similar to mitigations against undervolting-based mitigations [48], for which such a mitigation has been shown to be effective. Future work can investigate whether such a mitigation is also practical for such stack-based attacks, regardless of whether the stack pointer is modified via StackWarp or via memory corruption.

8 Discussion

In this section, we discuss the applicability of our systematic core-scoped MSR analysis approach Section 3 to other registers, and related work.

8.1 Intel TDX

To check whether an analogue of StackWarp exists under Intel TDX, we use our setup from Section 3 on an Intel Xeon Gold 6526Y. We observe no cross-hyperthread effects on the TD's architectural state.

Still, we observe faults and system crashes in several test cases. However, most are trivial. For instance, we observe system crashes when disabling the `syscall` instruction via `IA32_EFER`. In all cases, we expect that crashing behavior is not security relevant, given that access to these MSRs is restricted to privileged users. Crucially, we find no unexpected effects on the architectural state of the hyperthread.

8.2 Control Registers

Control registers (CRn) are another type of system register, similar to MSRs, used to configure the CPU. For example, setting bit 30 in CR0 disables the CPU cache, and modifying the extended control register (XCR0) can disable ISA extensions.

An SEV-SNP VM stores its own guest control registers in the VMSA, protected by integrity checks. We attempted to modify the host XCR0 to disable ISA extensions, such as x87 and AVX, and injected faults into a running SEV-SNP VM. However, this did not appear to affect the guest XCR0, and the VM continued to operate normally.

8.3 Related Work

Hardware-related Integrity Attacks on AMD SEV. CacheWarp [4] demonstrates that the `invd` instruction can be turned into a precise software fault primitive. A malicious hypervisor interrupts a confidential VM (AMD SEV, SEV-ES, or SEV-SNP), lets it keep only the cache lines it wants, then invalidates the rest so the guest continues with deliberately *stale memory*, i.e., cache-line content before the write. CacheWarp is, therefore, the first pure software technique that subverts SEV-SNP integrity without needing to read encrypted memory or know guest-physical addresses.

In contrast, StackWarp never touches the cache state. Instead, it relies on an MSR write that shifts the guest stack pointer. This means that the primitive acts immediately inside the running guest rather than between two VM exits, and the attacker gets more control for exploitation as the modification of the stack pointer can be controlled. Moreover, StackWarp affects SEV-SNP on Zen 5, where CacheWarp is fixed.

Software-related Integrity Attacks on AMD SEV. Heckler [6] exploits the hypervisor's ability to inject any external interrupt. This leads to unexpected control flow, e.g., when injecting a legacy syscall (`int 0x80`), that modifies register values inside the guest VM. WeSee [5] generalises this idea to the SEV-SNP–specific `#VC` vector 29: by forging the `exit_reason` field, the attacker convinces the guest's VC handler to copy arbitrary data between the GHCB and guest context, achieving register leaks, register clobbers, and code injection. Both attacks group such techniques under *Ahoi attacks*, meaning notifications that directly change architectural state instead of merely exposing a side channel.

Both Heckler and WeSee rely on injected *interrupts* whose delivery path is visible in the guest's control flow and can be filtered by future interrupt-filter hardware. StackWarp, in comparison, requires no interrupt or VM exit: the sibling core writes to an undocumented MSR bit, immediately relocating the victim's stack without any guest-visible event.

Side-channel Attacks. There is a long line of research on AMD SEV that abuses page-table remapping [49], encryption oracles [50], or fault oracles [51] to read or modify memory contents. Most of these attacks have been mitigated with SEV-SNP, leaving only microarchitectural side-channel attacks, such as Prime+Probe [52] as a known attack surface. Such microarchitectural side channels are limited to leaking meta data, and have strong assumptions on the victim, i.e., the victim requires secret-dependent memory accesses. In contrast, StackWarp makes no such assumptions and exploitation cannot be prevented with constant-time programming techniques [53]. Concurrent work [54] presents side-channel attacks on the stack engine and identifies the same MSR bit as a mitigation by disabling the stack engine. Their findings are complementary and can assist in adapting StackWarp to other microarchitectures. Notably, their discovery of immediate-based operations on Zen 5 simplifies the profiling of the desired shift offset required for our attack.

9 Conclusion

We discovered StackWarp, a frontend-based stack synchronization vulnerability enabling attacks to fully compromise AMD SEV-SNP VMs. Our templating-based sytematic analysis of undocumented MSR bits on AMD CPUs discovered stack corruptions leading to the discovery of StackWarp. Our root-cause analysis showed that StackWarp enables an attacker to deterministically shift the stack pointer on the sibling logical core. We demonstrated the security impact of StackWarp by mounting several attacks against AMD SEV-SNP running on fully patched Zen 5 hardware. Using StackWarp, we demonstrated RSA key recovery and fully bypass OpenSSH authentication. Furthermore, we showed two ways to achieve privilege escalation via sudo and using a kernel-level ROP chain, leading to a full system compromise. We discussed potential mitigations for StackWarp. We conclude that StackWarp shows that SMT undermines the security SEV-SNP integrity guarantees.

Acknowledgment

We want to thank our shepherd and the anonymous reviewers for their guidance, comments and valuable suggestions, as well as Lukas Gerlach and Youheng Lü for helpful feedback on this work. This work was partly supported by the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS).

Ethical Considerations

This work exposes a previously unknown vulnerability in AMD Zen CPUs that undermines the integrity guarantees of SEV-SNP, a technology widely deployed in confidential virtual machines. Such findings inherently carry ethical implications, as they demonstrate practical attacks that could be misused by malicious actors if prematurely disclosed. To mitigate these risks, we followed responsible disclosure practices. We reported our findings to AMD on March 24, 2025, received acknowledgment on March 25, 2025, and coordinated with the vendor regarding CVE assignment and embargo timelines. AMD has confirmed that hot-loadable microcode patches have already been released to their customers. No details beyond what is described in this paper were shared publicly prior to vendor acknowledgment, and proof-of-concept code will only be released publicly after vendor patches or mitigations are available.

Our experiments were conducted exclusively on hardware owned and controlled by the authors in a laboratory environment. No experiments involved third-party systems, production cloud infrastructure, or workloads belonging to others. The attack demonstrations targeted only test virtual machines we deployed ourselves, ensuring that no external users or data were put at risk. Furthermore, the study does not involve human subjects, nor does it collect or process personal data, so no institutional ethics board review was required.

We are aware that publishing attack techniques may enable adversaries to replicate them. To balance scientific transparency with security concerns, we limited technical detail in ways that ensure reproducibility for the research community while avoiding step-by-step exploit recipes that could facilitate immediate misuse. The released artifacts will include microbenchmarks and measurement scripts sufficient for validation, but will not include weaponized exploits. By coordinating disclosure, restricting experiments to safe environments, and carefully controlling publication of supporting materials, we aim to maximize the positive impact of our work for improving hardware security while minimizing the potential for abuse.

Stakeholder Analysis

- **Cloud Tenants**: This group is the primary party at risk, as the vulnerability directly breaks the TEE security promise. The main positive impact of our paper is to protect these users by discovering the Stack Warp vulnerability and working together with the vendor to provide a long-term fix.
- **Cloud Service Providers (CSPs)**: While a malicious insider is the threat actor, benign CSPs benefit from our work. It allows them to understand the true security posture of their services and apply vendor patches to protect their customers.
- **CPU Vendor (AMD)**: As the hardware vendor, AMD is responsible for the fix, and our findings contribute directly to their product security.

Open Science

Our artifacts can be viewed at: https://zenodo.org/records/17898697. We provide the experiment code for the proof-of-concept, the performance counter analysis, the profiling and injection phase in the KVM, and the case studies.

- **simple-poc**: A poc to verify if StackWarp works on the machine.
- **ucode_instr_num**: μops PMC analysis (Section 4.2).
- **architectural_tests**: Architectural behavior testing for StackWarp discovery (Section 3).
- **kvm**: Modified KVM for profiling and injecting phase (Section 5.1).
- **rsa_crt**: Artifacts of the RSA Key Recovery case study (Section 6.1).
- **openssh-exploit**: Artifacts of the Openssh case study (Section 6.2).
- **getuid-exploit**: Artifacts of the Sudo case study (Section 6.3).
- **kernel_rce**: Artifacts of the Ring 0 Code Execution case study (Section 6.4).

References

[1] AMD, "Amd invd instruction security vulnerability," 2023. [Online]. Available: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3005.html

[2] 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 *USENIX Security*, 2021.

[3] 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 *S&P*, 2022.

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

[5] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde, "Wesee: Using malicious# vc interrupts to break amd sev-snp," in *S&P*, 2024.

[6] B. Schlüter, S. Sridhara, M. Kuhne, A. Bertschi, and S. Shinde, "HECKLER: Breaking confidential vms with malicious interrupts," in *USENIX Security*, 2024.

[7] B. Schlüter, C. Wech, and S. Shinde, "Heracles: Chosen plaintext attack on amd sev-snp," in *CCS*, 2025.

[8] Y. Yan, W. Huang, I. Grishchenko, G. Saileshwar, A. Mehta, and D. Lie, "Relocate-vote: Using sparsity information to exploit ciphertext side-channels," in *USENIX Security*, 2025.

[9] L.-C. Chiang and S.-W. Li, "Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV," in *ASPLOS*, 2025.

[10] L. Giner, S. R. Neela, and D. Gruss, "Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP," in *DIMVA*, 2025.

[11] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *S&P*, 2019.

[12] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port contention goes portable: Port contention side channels in web browsers," in *AsiaCCS*, 2022.

[13] T. Hornetz and M. Schwarz, "PortPrint: Identifying Inaccessible Code with Port Contention," in *uASC*, 2025.

[14] S. Deng, M. Li, Y. Tang, S. Wang, S. Yan, and Y. Zhang, "CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations," in *USENIX Security Symposium*, 2023.

[15] J. Wichelmann, A. Pätschke, L. Wilke, and T. Eisenbarth, "Cipherfix: Mitigating ciphertext side-channel attacks in software," in *USENIX Security*, 2023.

[16] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen, "Early load address resolution via register tracking," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, 2000.

[17] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," 2024.

[18] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "Sev-step: A single-stepping framework for amd-sev," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024.

[19] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on RSA with CRT: Concrete results and practical countermeasures," in *CHES*, 2002.

[20] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," 2001.

[21] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[22] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *S&P*, 2020.

[23] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies," in *CCS*, 2019.

[24] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A. Sadeghi, "V0LTpwn: Attacking x86 Processor Integrity from Software," in *USENIX Security Symposium*, 2020.

[25] Intel Corporation, "Hyper-Threading Technology: Architecture and Microarchitecture," 2002.

[26] AMD, "Technology Brief: AMD EPYC and SMT," 2025.

[27] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "SecSMT: Securing SMT processors against Contention-Based covert channels," in *USENIX Security*, 2022.

[28] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. D. Valentine, "The intel pentium m processor: Microarchitecture and performance," 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:51799543

[29] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface," in *USENIX Security*, 2021.

[30] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization," in *CCS*, 2021.

[31] 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 *S&P*, 2025.

[32] A. Kogler, D. Weber, M. Haubenwallner, M. Lipp, D. Gruss, and M. Schwarz, "Finding and Exploiting CPU Features using MSR Templating," in *S&P*, 2022.

[33] Google, "Msrscan," 2024. [Online]. Available: https://github.com/google/security-research/tree/master/pocs/cpus/misc/msrscan

[34] C. Lam, "Turning off zen 4's op cache for curiosity and giggles," 2024. [Online]. Available: https://chipsandcheese.com/p/turning-off-zen-4s-op-cache-for-curiosity

[35] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss, "Squip: Exploiting the scheduler queue contention side channel," in *S&P*, 2023.

[36] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, 2011.

[37] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *S&P*, 2015.

[38] D. Pei, A. Salomaa, and C. Ding, "Chinese remainder theorem: applications in computing, coding, cryptography," in *World Scientific*, 1996.

[39] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "MicroWalk: A Framework for Finding Side Channels in Binaries," in *ACSAC*, 2018.

[40] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2020.

[41] R. Zhang, A. Cheu, A. Gascon, D. Moghimi, P. Schoppmann, M. Schwarz, and O. Suciu, "SNPeek: Side-Channel Analysis for Privacy Applications on Confidential VMs," in *NDSS*, 2026.

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

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

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

[45] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," *arXiv:1905.05725*, 2019.

[46] "AMD64 Architecture Programmer's Manual," Advanced Micro Devices Inc., 2025.

[47] AMD, "AMD PRO Technologies Security White Paper," 2025. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/products/processors/technologies/amd-pro-technologies-security-white-paper.pdf

[48] A. Kogler, D. Gruss, and M. Schwarz, "Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks," in *USENIX Security*, 2022.

[49] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "Severed: Subverting amd's virtual machine encryption," in *EuroSec*, 2018.

[50] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "Sevurity: No security without integrity–breaking integrity-free memory encryption with minimal assumptions," in *S&P*, 2020.

[51] M. Li, Y. Zhang, and Z. Lin, "CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV," in *SIGSAC*, 2021.

[52] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[53] Cauligi, Sunjay and Disselkoen, Craig and Gleissenthall, Klaus v and Tullsen, Dean and Stefan, Deian and Rezk, Tamara and Barthe, Gilles, "Constant-time foundations for the new spectre era," in *SIGPLAN*, 2020.

[54] S. Niederer, S. Rüegge, A. Hajiabadi, and K. Razavi, "One Flew over the Stack Engine's Nest: Practical Microarchitectural Attacks on the Stack Engine," in *MICRO*, 2025.

[55] O. C. Company, "illumos," 2025. [Online]. Available: https://github.com/oxidecomputer/illumos-gate/blob/stlouis/usr/src/uts/intel/sys/amdzen/ccx.h

A Crashing MSR Bits

MSR `0xC0011092` and `0xC0011093` are partially documented as L3 cache controller configuration registers on Zen 3 and Zen 4 [55]. We find that repeatedly flipping bit 1 of `0xC0011092` triggers a Machine Check Error (MCE) classified as a microarchitectural error. Similarly, repeatedly flipping bit 24 of `0xC0011093` triggers another MCE. The reported MCE indicates a parity check error in the L1 cache. However, we are not able to find an exploitable use for these behaviors.

MSR `0xC0011097` is not officially documented but is referenced as part of the L3 XI complex interface by Oxide [55]. Notably, bit 15 is only allowed to be set once. Once set, the hypervisor can no longer launch a new SEV-SNP VM, as it fails to create an encryption context according to the error log. If an SEV-SNP VM is already running, the hypervisor can still set this bit. The VM times out on shutdown, the CPU disables the PSP, and the hypervisor is unable to resume the VM or launch a new one.

B Stack Pointer Analysis

We use the code in Listing 4 (eight `push` with `mfence` between them) to analyze the effect of toggling bit 19 of MSR (`0xC0011029`). Setting the bit 0→1 yields a *negative* offset (missed increments of `rsp`). Resetting 1→0 yields the same magnitude but *positive*. Conversely, with sequences of `pop`, the signs invert.

C Toy Example of Dropping Write

Listing 5 illustrates a toy checker for a root login. The function stores the password pointer from `pw->pw_passwd` in the local variable `correct`. If `getuid() == 0`, or `correct` is `NULL`, or `correct[0] == '\0'`, the function returns success. An attacker can shift `rsp` immediately before the assignment to `correct`, causing the store to miss its intended stack slot. Consequently, the password variable remains uninitialized, and the subsequent check may observe `correct == NULL`, thereby granting access.

```c
void uf1() {
  printf("Unreachable 1!!!\n");
}
void uf2() {
  printf("Unreachable 2!!!\n");
}

__attribute__((naked)) void push_stack(void *p) {
  asm volatile(
    ".rept 8\n\t"
    "push %0\n\tmfence\n\t" // address of uf2()
    ".endr\n\t"
    "add rsp, 0x40\n\t" // stack balance
    "ret\n\t"
    :: "r"(p) : "memory");
}

int main() {
  void (*fp[8])(void) = {
    uf1, uf1, uf1, uf1, // address of uf1()
    uf1, uf1, uf1, uf1, // on the stack
  };
  for (;;) {
    push_stack((void *)uf2); // save ret address
  }
}
```

Listing 4: Program of stack pointer analysis.

```c
bool correct_password(const struct passwd *pw) {
  // addressed from RSP
  char *correct;
  // shift RSP so this store misses
  correct = pw->pw_passwd;
  // observed as NULL -> success
  if (getuid() == 0 || !correct || correct[0] == '\0')
    return true;
  ...
}
```

Listing 5: Drop-write primitive: a large $|\Delta|$ moves the target slot out of the active frame, such that the store is "lost". On restore of the stack pointer, later reads see a stale/uninitialized value.