CacheWarp: Software-based Fault Injection using Selective State Reset

Ruiyi Zhang
CISPA Helmholtz Center for Information Security

Lukas Gerlach
CISPA Helmholtz Center for Information Security

Daniel Weber
CISPA Helmholtz Center for Information Security

Lorenz Hetterich
CISPA Helmholtz Center for Information Security

Youheng Lü
Independent

Andreas Kogler
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

AMD SEV is a trusted-execution environment (TEE), providing confidentiality and integrity for virtual machines (VMs). With AMD SEV, it is possible to securely run VMs on an untrusted hypervisor. While previous attacks demonstrated architectural shortcomings of earlier SEV versions, AMD claims that SEV-SNP prevents all attacks on the integrity.

In this paper, we introduce CacheWarp, a new software-based fault attack on AMD SEV-ES and SEV-SNP, exploiting the possibility to architecturally revert modified cache lines of guest VMs to their previous (stale) state. Unlike previous attacks on the integrity, CacheWarp is not mitigated on the newest SEV-SNP implementation, and it does not rely on specifics of the guest VM. CacheWarp only has to interrupt the VM at an attacker-chosen point to invalidate modified cache lines without them being written back to memory. Consequently, the VM continues with architecturally stale data. In 3 case studies, we demonstrate an attack on RSA in the Intel IPP crypto library, recovering the entire private key, logging into an OpenSSH server without authentication, and escalating privileges to root via the `sudo` binary. While we implement a software-based mitigation proof-of-concept, we argue that mitigations are difficult, as the root cause is in the hardware.

1 Introduction

In recent years, trusted execution environments (TEEs) have become available on many x86 desktop and server CPUs. While Intel SGX was the first widely-available TEE, it is limited to smaller microservice workloads and requires significant changes to the code running inside the TEE. The next generation of TEEs, specifically AMD SEV and Intel TDX, secure entire virtual machines (VMs). In this threat model, trusted VMs can run on an untrusted hypervisor. The hypervisor can ideally neither read (confidentiality) nor modify (integrity) any content of the VM. Cloud providers, such as Microsoft Azure, Google Cloud, and Amazon Web Services already support AMD SEV [4, 21, 40]. With AMD SEV-SNP, AMD extended SEV to support full integrity protection [6], mitigating previous attacks on the integrity of SEV and SEV-ES VMs [15, 35, 42, 47, 57]. AMD also acknowledges that well-known software-based side-channel attacks, such as Prime+Probe are possible against all versions of AMD SEV [6].

In this paper, we present CacheWarp, a new software-based microarchitectural fault attack on AMD SEV-ES and SEV-SNP, targeting memory writes. We show that creating an incoherent view of the memory is possible, causing inconsistencies between the cache and the memory. Specifically, we commit a subset of data writes to memory while dropping changes to other memory locations using the `invd` instruction. Consequently, the VM memory state is only partially updated and the VM uses stale memory. CacheWarp is fully deterministic and allows fine-grained dropping of memory writes from the hypervisor. In contrast to previous attacks [34, 36], CacheWarp is independent of the weaknesses of the AES-XEX encryption used in AMD SEV. Moreover, the hypervisor does not need read or write access to any encrypted VM page. In addition to dropping arbitrary writes, we introduce 2 generic techniques to exploit such dropped writes in a generic fashion. With *Dropforge*, dropping implicit and explicit writes of functions can change function parameters and behavior. With *Timewarp*, the implicit push of the return address caused by a function call can be dropped to divert the control flow to the call site of an *earlier* function call with the return value of the *current* function.

CacheWarp has several advantages over previous attacks that violated the integrity of the guest VM. Most importantly, CacheWarp can be mounted against SEV-SNP. It is, therefore, to the best of our knowledge, the only software-based attack violating the integrity of SEV-SNP. Moreover, even for SEV-ES, CacheWarp has advantages over previous attacks. It does not have to change any memory mappings [35, 42, 47] add malicious I/O devices [47], or build encryption oracles [57]. CacheWarp only needs to evict modified data from the cache that should not be dropped. This can either be done in parallel to the execution of the VM using targeted eviction sets, or by interrupting the VM at two attacker-chosen points. On the first interrupt, the attacker commits all outstanding memory writes to memory, e.g., using the `wbnoinvd` instruction. On the second interrupt, the attacker reverts a subset of the outstanding memory writes to old, now stale, values using the `invd` instruction.

While CacheWarp can be mounted "blindly", precise control over the time when it is mounted improves the reliability. In contrast to SGX [52] where single-stepping an SGX enclave has been well explored, AMD SEV runs an entire operating system beneath the victim code, making it difficult to precisely target a specific instruction at which the VM should be interrupted. Interrupting an AMD SEV VM requires storing and loading an entire VM state, resulting in many unrelated instructions with unpredictable timings, making triggering interruptions after a specific instruction difficult. Hence, we introduce a robust execution-control framework for VMs running on AMD SEV that is based on page faults and timer interrupts. Conceptually, it is similar to SGX-Step [52] and SEV-Step [56], i.e., using the APIC timer to generate interrupts constantly. However, we introduce additional techniques for reliable stepping to account for the unpredictable runtimes of saving and restoring the VM state. We rely on uncacheable memory for the VM save area and the `wbinvd` instruction for bringing the cache into a known state, resulting in highly-predictable runtimes for stopping and resuming VMs, significantly improving stepping. Additionally, for SEV-ES, we introduce a new technique using register-state feature arrays to reliably detect progress in the VM despite the encrypted register state and to synchronize our attack with the victim. For SEV-SNP, we rely on the progress detection of SEV-Step [56].

We demonstrate CacheWarp and our stepping framework in 3 case studies. In the first case study, we mount a Bellcore attack [10, 12] on the RSA-CRT implementation of the Intel IPP crypto library. Our attack recovers the full private RSA key from an AMD SEV-protected VM within 6 s. Even without single stepping, we achieve a success rate of 90 %. In the second case study, we show that CacheWarp diverts the control flow to the call site of a previous function with the return value of a previously called function. We use this capability to break the authentication of an OpenSSH server running inside a guest VM, allowing an attacker to log into the VM. In the final case study, we demonstrate that an attacker with unprivileged code execution in the VM, i.e., after exploiting the OpenSSH server, can perform privilege escalation by using CacheWarp on the `sudo` binary. Although SEV-ES is already known to provide incomplete protection, we rely on SEV-ES as a "prototyping platform" for the attacks, as reliable stepping is simpler due to the visibility of the register state. Hence, the exploit-development time, including debugging, is significantly reduced. However, as the underlying vulnerability is the same for SEV-SNP, the resulting exploits can be mounted on SEV-SNP as well, as we demonstrate for the Bellcore attack.

Mitigating CacheWarp purely in software is difficult as it exploits a memory-coherence problem that a malicious hypervisor can create. AMD could update the microcode, disabling the `invd` instruction if AMD SEV is active, or change its behavior to first write back any modified data to the memory, similar to the `wbinvd` instruction. Such behavior is in line with the behavior on Intel CPUs, which disable `invd` if SGX is active. Additionally, we propose a software-only compiler-based mitigation that prevents exploiting CacheWarp. Programs compiled with this mitigation ensure that every write is committed to the memory, making it impossible for the attacker to drop writes without being detected.

CacheWarp demonstrates an efficient software-based method of violating the integrity and, consequently, the confidentiality guarantees of AMD SEV. As the root cause is in hardware, fixing CacheWarp requires firmware or hardware changes. Moreover, further research is required for effective and efficient software mitigations.

Contributions. The main contributions of this paper are:

1. We present CacheWarp, a software-based fault attack on AMD SEV-ES and SEV-SNP that allows dropping arbitrary writes, reverting selected cache lines to a previous (stale) value.
2. We introduce two new attack techniques to change the behavior of function calls and to return to a previous call site with a different return value.
3. We demonstrate CacheWarp in 3 case studies: Extracting the private key from the IPP RSA implementation, logging into an OpenSSH server without credentials, and escalating privileges to root using the `sudo` tool.
4. We propose a compiler-based software-only mitigation that reliably prevents exploitation of CacheWarp.

Outline The remainder of this paper is organized as follows. We provide the required background information in Section 2. Section 3 analyzes the behavior of cache invalidation instructions on Intel and AMD. Section 4 introduces our software-based fault injection attack and the techniques used. Section 5 presents the building blocks used in our case studies in Section 6. Section 7 discusses countermeasures. Section 8 discusses limitations and related work. Section 9 concludes.

Responsible Disclosure We responsibly disclosed our findings to AMD on April 25th, 2023. AMD acknowledged our findings on June 27th, 2023 and assigned CVE-2023-20592. AMD confirmed that they will mitigate CacheWarp for SEV-SNP via a microcode patch and an SEV firmware update for Zen 3 EPYC Milan CPUs. The source code of the framework and the proof-of-concepts are open-sourced at `https://github.com/cispa/CacheWarp`

2 Background

This section provides the background information to complement the rest of the paper.

2.1 Virtual Memory

CPUs support virtual memory to isolate processes and translate virtual addresses to physical addresses, i.e., locations in DRAM. The prevalent implementation of virtual memory uses multi-level page tables. The translation unit of page tables is a page and is usually 4 kB in size. While the CPU handles address translation, the operating system configures multi-level page tables to define each process's virtual memory mapping. Alongside the mapping, the operating system stores access permissions and additional meta information for each page mapping in the page table. To extend virtual memory to VMs, AMD-V (AMD Virtualization) supports nested page tables [2]. With nested page tables, the guest VM configures a page table translating from Guest Virtual Address (gVA) to Guest Physical Address (gPA). Furthermore, the hypervisor provides an additional page table mapping from gPA to Host Physical Address (hPA). On each memory access, the composition of those page tables is used to translate from gVA to hPA. Nested paging allows running multiple VMs and isolating their memory without requiring any modifications to the operating systems used by the VMs.

2.2 AMD Cache Hierarchy

Most modern CPUs use a cache hierarchy to reduce the latency of memory accesses. AMD and Intel CPUs have a per-core L1 and L2 cache and a shared last-level cache (LLC), partitioned into slices and Core Complexes on AMD. Usually, the L1 is split into data- and instruction-caches [2, 25]. If a data load can be served from any level of the cache, it has a lower latency. Otherwise, it is fetched from DRAM and usually stored in the cache hierarchy for subsequent accesses. Data may not be written to DRAM directly on write accesses, but an existing cache line may be modified and marked as dirty in the metadata.

Eviction. As caches are limited in size, loading a cache line may require evicting another line based on the eviction strategy. Dirty cache lines must be written back to DRAM if evicted to keep a coherent memory view.

Core Complex (CCX). Starting with the Zen microarchitecture, AMD relies on a modular chiplet design for their multi-core CPUs [7] that are divided into one or more Core Complexes (CCXs) [14]. A CCX consists of up to 8 CPU cores alongside private caches and an LLC cache shared amongst cores in the same CCX [3]. Additionally, AMD relies on non-inclusive cache hierarchies [28]. The CCXs are interconnected with AMD's proprietary infinity fabric and interface with a shared I/O die, responsible for handling DRAM accesses, PCIe communication, and other I/O operations [3,5].

2.3 Software-based Fault Attacks

A fault attack involves intentionally introducing errors or faults into a system to observe its behavior and extract sensitive information. While fault attacks have been well studied in the context of a physical attacker [11, 20], two prominent examples of software-based fault attacks are Rowhammer [30] and software-based undervolting [29, 43, 46]. Both of these attacks rely on the physical properties of the underlying system. Rowhammer exploits disturbances in DRAM memory created by specific access patterns to induce bit flips. Undervolting brings the CPU into an unstable state where specific computations (most prominently multiplications) lead to incorrect results. These attack primitives lead to powerful attacks undermining the security guarantees of the victim system. However, as they both depend on the hardware and environmental properties of the system under attack, the extent to which they can be exploited is highly dependent on the specific DRAM module (Rowhammer) or CPU (undervolting) of the victim.

2.4 Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (AMD SEV) protects VMs running under an untrusted hypervisor [2]. All versions of AMD SEV support encrypting guest memory using AES-XEX [49], an AES mode of operation tailored towards full disk encryption. The per-guest AES keys are generated and stored on a secure co-processor and are not accessible. The CPU handles encryption and decryption when a guest accesses encrypted memory. Building on AMD-V [2], SEV also uses nested paging.

With AMD SEV-ES (Encrypted State) [9], AMD introduced protection for the saved VM state on transitions to the hypervisor. The VM State Save Area (VMSA), which contains the guest's state, is no longer part of the Virtual Machine Control Block (VMCB) but rather stored in encrypted memory, guaranteeing confidentiality. Additionally, a checksum over the VMSA is stored in a protected memory area, so malicious hypervisors cannot modify information such as register contents. Whenever the CPU transitions from hypervisor to guest, the checksum is verified, and if the checksum is invalid, an exception is raised, ensuring the integrity of the VM state.

Finally, AMD SEV-SNP (Secure Nested Paging) [6] adds a Reverse Map Table (RMP) to track the ownership of each guest page. The RMP verifies the gPA to hPA mapping on each memory access, blocks hypervisor writes to guest pages and ensures that each page is mapped to only one gPA. Thus, AMD SEV-SNP mitigates memory-mapping attacks by a malicious hypervisor [57]. For instance, an attacker can no longer revert a page to a previous version, as is possible with AMD SEV-ES, by reading the encrypted page content and writing it back later. Although AMD SEV-SNP protects integrity and confidentiality, a malicious hypervisor can still read the encrypted memory and distinguish unique ciphertexts, except for the VMSA, which is protected with *freshness* [34]. Therefore, Wichelmann et al. [55] propose a software-based mitigation to include freshness in each memory encryption block.

3 Cache Invalidation

In this section, we analyze the behavior of the root mechanism in CacheWarp on Intel and AMD CPUs, the cache-invalidation instructions `invd` and `wbinvd`. We show that these privileged instructions have undesirable effects when used by an attacker, which is possible in the threat model of TEEs (Section 3.1). We demonstrate previously undocumented behavior that allows violating the consistency of the memory view, leading to architectural data changes inside TEEs (Section 3.2). Finally, we show that non-coherent memory types are allowed on some AMD client CPUs (Section 3.3).

Setup. For our evaluation, we test on different microarchitectures, namely AMD Rome (AMD EPYC 7252, microcode `0x8301055`), Zen (AMD Ryzen 5 2500U, microcode `0x810100b`), Zen+ (AMD Ryzen 5 3550H, microcode `0x8108102`), and Zen 3 (AMD Ryzen 9 5900HX, microcode `0xa50000c`). For the analysis of the instructions independent of the TEE, we rely on Ubuntu 20.04 with kernel 5.15.0. For the experiments on AMD SEV, we use Ubuntu 22.04 with kernel 6.1.0 on the hypervisor, and Ubuntu 20.04 with kernel 5.15.0 inside the VM.

3.1 Availability

Both Intel and AMD CPUs [2, 25] support the `invd` instruction to invalidate all levels of the internal cache, including the data and instruction cache. Crucially, invalidating the cache containing modified data irreversibly destroys these changes, essentially reverting data modifications to the previous stale state. In contrast, the `wbinvd` instruction writes back any modified cached content to memory before invalidating the cache. In addition, there is the `wbnoinvd` instruction that only writes back modified cache contents to the main memory but does not invalidate the cache. Unlike `wbinvd` and `wbnoinvd`, `invd` has a very limited use case and should only be used if memory consistency is not required and caches are not shared between threads or cores [1]. However, in the threat model of TEEs, the `invd` instruction can be abused by a privileged attacker.

On AMD, `invd` is, by default, converted to `wbinvd`. This behavior can be changed by clearing bit 4 of MSR (`0xc0010015`) [1]. We additionally check the availability of `invd` instructions inside VMs with QEMU v7.2.0. KVM intercepts all invalidation instructions. Hence, CacheWarp can only be mounted via the hypervisor but not from within other VMs.

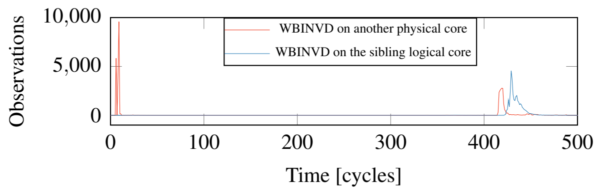


Figure 1: Access time for reloading data from different domains after executing `wbinvd`/`invd`.

Intel explicitly documents that the `invd` instruction is unavailable if Intel SGX is enabled [27, §3.6.5]. More precisely, this instruction raises a general protection fault if processor-reserved memory protection is activated. Consequently, this is also the case for Intel TDX, as TDX crucially relies on the Intel SGX-based quoting enclave as the root of trust [26]. We experimentally verify that the behavior of the `invd` instruction matches the documentation. Thus, CacheWarp is not possible on Intel CPUs.

3.2 Scope

In addition to the availability of the instructions, we analyze their scope, i.e., *which* cache lines are affected, to understand the impact of these instructions on AMD. For the private caches, i.e., L1 and L2, the `invd` and `wbinvd` instructions are limited to a physical core. Hence, they only invalidate data of the current logical and sibling logical core. Although not documented, our experiments show that the LLC part within a CCX is invalidated by the `invd` and `wbinvd` instructions.

Private Caches To measure the effect of the `invd` instruction on different cache levels, we rely on eviction to control the cache state. We rely on eviction sets for the L1 cache to reliably bring data into a specific cache set of the L2. To ensure that data resides in the L2 but not in the L1, we rely on an L1 eviction set that evicts the data from the L1 but not from the L2. Accessing the eviction set is fast as long as the eviction set stays in the respective cache. Hence, if data is invalidated, we observe an increased eviction time. For each cache level, we measure the average access time while executing the `invd` instruction versus executing no instruction on the sibling core. As a second experiment, we measure the eviction timing after executing the `invd` instruction on a different physical core within the CCX. To get precise timing measurements, we rely on the `rdpru` instruction [37]. As shown in Figure 1, executing `wbinvd` or `invd` on another physical core does not affect data within the private caches. However, data in the shared cache is invalidated to memory, exhibiting a red peak on the right side. To conclude the first analysis, `invd` invalidates the private caches on the physical core.

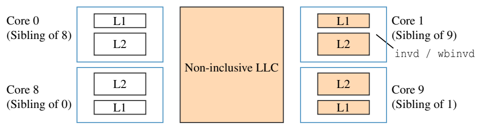


Figure 2: The scope of `wbinvd` and `invd` within a CCX that consists of 2 physical CPUs with 4 CPU cores. The boxes stand for different cache domains. The orange-shaded areas represent the scope of the `invd` instruction.

Shared Cache To understand the scope of the `invd` instruction on the shared non-inclusive LLC, we rely on L2 eviction. The LLC eviction is independent of the L2 eviction, as the LLC is non-inclusive. However, the addresses in the LLC eviction set share their cache set index with those in the L2 eviction set. We can extend an eviction set only for the LLC into one that additionally works for the L2 by accessing 8 more addresses mapping to the same set, as an L2 cache set has 8 ways. As bits 6-8 of the physical address determine the slice [18], a minimum of 28 physical addresses with the same bits 6 to 19 is required to achieve effective LLC eviction. Additionally, Figure 1 shows that the data in the LLC is invalidated, independent of the core within the CCX where the `invd` instruction is executed. Hence, this shows that the `invd` instruction invalidates the entire LLC shared within a CCX but does not affect other CCXs. Figure 2 summarizes our results.

Cache-line State The execution time of the `invd` and `wbinvd` depends on the number of modified, i.e., dirty, cache lines of the affected cache. We experimentally evaluate that by varying the number of dirty cache lines from 0 to 512 and measuring the average execution time over 10 000 repetitions. We observe a clear linear relationship between the number of dirty cache lines and the timing of the instruction.

3.3 Non-coherent Memory

In addition to manually triggering inconsistencies using the `invd` instruction, memory type range registers (MTRRs) on AMD can also violate memory consistency. Both on Intel and AMD, each physical core has its own MTRRs to configure the memory type of physical address ranges. The main idea is to have different memory types on different CPU cores for the same physical address. While Intel does not have non-coherent memory types due to the self-snoop feature, the uncacheable (UC) and write-combining (WC) memory types are documented as non-coherent on AMD [2].

Experimental Setup We map two virtual addresses to the same physical address on two different cores. As the default memory type is write-back (WB), we mark the memory range as UC on one of the cores by using MTRRs. We experimentally confirm that writing to uncacheable memory does not trigger cache updates or invalidations. If two cores write to the same memory and both writes occur on the WB memory type prior to writes on the UC memory type, the writes stored in the cache may later overwrite the writes to the UC memory if a write-back of the cache line is triggered. Our experiments show that this coherency violation also applies to virtualization on AMD client CPUs, ranging from Zen to Zen 3 microarchitectures. In this context, if the guest VM writes to the memory, the hypervisor can discard or "drop" those writes. However, when performing this experiment on an AMD server CPU that supports AMD SEV, the hardware enforces the use of coherent memory types. As a result, the non-coherent memory type, i.e., UC, is converted to the forced coherent memory type, cache disable (CD). If the cache line is in a modified state, writes on CD memory cause the cache line to be written back before being invalidated. Moreover, accesses to a dirty cache line are performed after the invalidation is finished. Hence, a simple attack using a non-coherent memory type for guest access is impossible for AMD SEV.

4 CacheWarp

In this section, we introduce CacheWarp, a software-based fault-injection attack on AMD SEV-ES and SEV-SNP. CacheWarp is based on our analysis of the `invd` instruction, which shows that invalidating dirty cache lines without triggering a write-back is feasible on AMD CPUs, even if AMD SEV is enabled. Specifically, a malicious hypervisor can selectively drop any writes of an AMD SEV-ES and SEV-SNP guest that occurred at an attacker-chosen point. The consequence of such a drop is that the VM architecturally uses stale data.

4.1 Threat Model

In line with previous work [35,42,47,57] and AMD's whitepaper [6], we assume a privileged attacker, i.e., a malicious hypervisor for AMD SEV. Such an attacker executes privileged code outside the VM. The attacker has full control of the scheduling and can pin the virtual CPUs to a logical CPU and offline other cores within the same CCX. Moreover, the attacker can influence the allocation of memory. Lastly, as the attacker can execute privileged code, the attacker can modify model-specific registers (MSRs) and write to APIC registers. However, the attacker has no control over the code or data inside the VM. These capabilities are all in the threat model of AMD SEV and we do not assume knowledge of the guest physical address (gPA) of the target function in the AMD SEV VM [36].

We evaluate the attack on an 8-core AMD EPYC 7252 CPU for SEV-ES and AMD EPYC 7313P and 7443 CPUs for SEV-SNP. As suggested by AMD [8], on AMD EPYC 7252, the host OS, QEMU, and OVMF are built with the master branch (Linux kernel 6.1.0, QEMU v7.2.0-2-g5204b499a6, OVMF

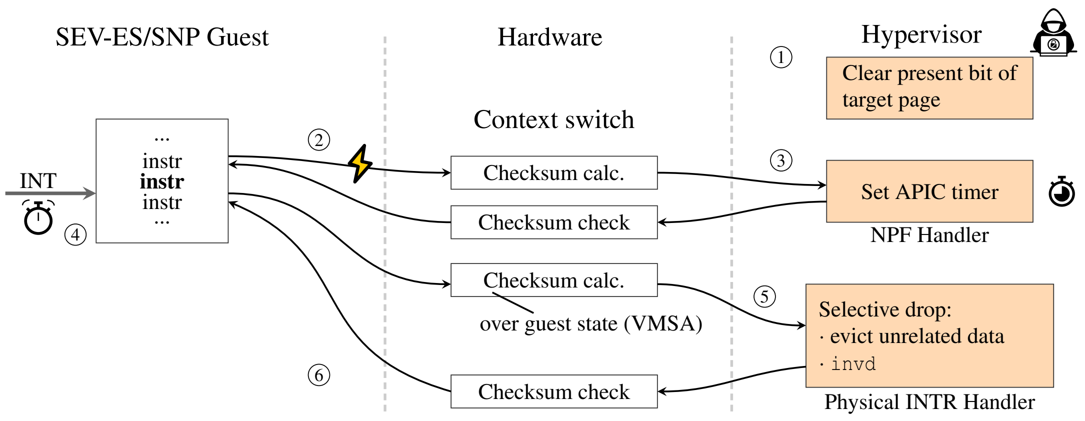


Figure 3: Overview of CacheWarp. The attacker clears the present bit (①) of the page containing the targeted store that should be dropped. Thus, the CPU induces a fault (②) when the guest execution reaches this page. In the fault handler, the attacker sets the APIC timer (③) to let the guest continue execution until the target instruction. When the timeout is reached, the guest is interrupted (④) and the attacker can selectively drop writes of the guest (⑤). The attacker evicts unrelated data from the cache, and drops all other modified data using the `invd` instruction. The guest then continues (⑥) with the dropped stores.

commit `cda98df`, firmware 0.24.15). For AMD SEV-SNP, we use the *snp-latest* branch (commit `ad91624`, firmware 1.54.01). The victim VMs are configured with a single virtual CPU and 4 GB of main memory. The victim system is running Ubuntu 20.04 LTS (Linux kernel 5.15.0).

4.2 Attack Overview

In this section, we present a high-level overview of CacheWarp. The goal of an attacker is to drop a guest write at a precise attacker-chosen point so that the guest architecturally uses stale data in the subsequent execution. Such writes can be explicit, e.g., local variables on the stack, but also implicit, e.g., return addresses of a `call` instruction. We show that both cases lead to powerful attacks. With Dropforge (Section 4.3), we change the behavior of functions. With Timewarp, (Section 4.4) we redirect return values to an earlier point in the program's control flow.

To locate a potential target, we introduce our framework for execution control. We rely on page faults and timer interrupts to interrupt guests at a specific state, dropping all writes not committed to the memory. Additionally, on AMD SEV-ES, we achieve reliable single-stepping using an uncacheable VMSA and register-state tracking (Section 5.1). To maintain the memory consistency of unrelated dirty cache lines, the hypervisor can selectively evict them into the memory (Section 5.2). This step guarantees that CacheWarp does not lead to a failed integrity check for an SEV guest. Figure 3 outlines each step of CacheWarp.

4.3 Dropforge

The main idea of Dropforge is to selectively drop writes during a function call to modify the behavior or result of a target

```
int victim(int a, int b) {
    int ret = 0;
    ret += a * 10 + b;
    return ret;
}

int main() {
    ...
    do {
        ret = victim(1, 1);
        reset = victim(2, 5);
    }
    while(ret == 11);
    ...
}
```

```
victim:
    push   %rbp
    mov    %rsp,%rbp
    mov    %edi,-0x14(%rbp) ; param
    mov    %esi,-0x18(%rbp) ; param
    movl  $0x0,-0x4(%rbp)   ; var
    mov    -0x14(%rbp),%edx
    mov    %edx,%eax
    shl    $0x2,%eax
    add    %edx,%eax
    add    %eax,%eax
    mov    %eax,%edx
    mov    -0x18(%rbp),%eax
    add    %edx,%eax
    add    %eax,-0x4(%rbp)  ; var
    mov    -0x4(%rbp),%eax  ; ret
    pop    %rbp
    retq
```

Figure 4: The toy example for dropping implicit writes introduced by the compiler. Depending on which write we drop, i.e., passed parameters, the initialization of local variables, or the assignment of local variables, we observe different return values for the program.

function. If an attacker drops a write from a function, the value is effectively reverted to the previous state, and the function works with a stale value. Writes that can be dropped are not only explicit writes but also writes introduced by the compiler. Compilers commonly introduce writes to the stack for exchanging register contents, i.e., the register content is written to the stack and then read into another register.

To illustrate the power of this attack primitive, Figure 4 shows a toy example containing both explicit and compiler-introduced writes. In this toy example, we show that passing parameters, the initialization of local variables, and the assignment of local variables can all be dropped. The guest VM keeps calling the function `victim` in a loop as long as the return value is correct. The second invocation of `victim` resets the stack frame with different values. Hence, once a

```
int ret1{
    return 1;
}
int ret0{
    return 0;
}
int main() {
    while(1){
        if (ret1() == 0){
            puts("WIN");
        }
        ret0(); // <- Victim
    }
}
```

```
1 main:
2       push    %rbp
3       mov     %rsp,%rbp
4       mov     $0x0,%eax
5       call    1149 <ret1>  ; <- push rip
6       test    %eax,%eax     ; <- old
        retaddr
7       jne     118c <main+0x25>
8       lea     0xe80(%rip),%rax
9       mov     %rax,%rdi
10      call    1050 <puts@plt>
11      mov     $0x0,%eax
12      call    1158 <ret0>  ; <- victim
13      jmp     116f <main+0x8>
```

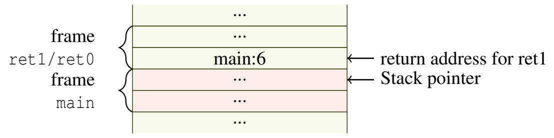


Figure 5: A toy example illustrating the Timewarp primitive. By dropping the implicit write of the `call ret0` instruction, we jump back to the call site of `ret1`. This effectively looks like `ret1` returned '0' and the program outputs "WIN".

drop occurs in the first invocation of `victim`, some different stale values are used. This reset also ensures that the observed return value is not a result of multiple dropped writes across multiple function invocations. In this toy example, most writes fall into the same cache line. Hence, when "blindly" dropping writes, it is highly likely that all writes are dropped. However, an attacker can use single-stepping to target individual writes, even if they fall into the same cache line.

In our example (compiled with GCC 9.4.0), an attacker can drop the writes from the compiler-generated prologue of the function, and the explicit write to the variable, which is used as the return value. We confirm that each instruction that performs a write operation (marked with a comment) can be skipped. Depending on the exact write that is dropped, we observe different return values in the calling function. While we can also drop the value that is pushed onto the stack by the `push %rbp` instruction, the next invocation of `victim` has the same stack frame. Hence, using the old value of `rbp` does not make a functional difference.

4.4 Timewarp

The main idea of Timewarp is to reuse a stale return address of a previous function to redirect control flow. Timewarp essentially jumps back to the call site of this previous function with the return value of the current function. Timewarp exploits that on x86, the return address is implicitly written to the stack on a function call and that stack frames are reused across function calls.

To illustrate Timewarp, Figure 5 shows a toy example. The main function calls two functions, `ret0` and `ret1`. Their expected return values are '0' and '1', respectively. Note that the loop is not strictly necessary. It only enables an attacker to repeat the attack arbitrarily often without having to restart the application and without requiring single stepping. For a realistic attack (cf. Section 6.2), we do not rely on such a loop. In Line 5, `ret1` is called (`call ret1`), which pushes the return address `main:6`, i.e., the address of the instruction following the call, onto the stack. When returning from `ret1`, the `ret` instruction at the end of the function reads the saved return address from the stack. The CPU jumps to this location by setting the instruction pointer to the address.

To modify the return value of `ret1`, we target the call to `ret0` (`call ret0` instruction) at Line 12. A `call` instruction consists of an implicit `push` and a `jmp` instruction. The `push` instruction writes the return address, i.e., the address of the instruction following the call, onto the stack. In this example, the return address is `main:13`. After the call, and before the function returns, the attacker can invalidate the write of the return address. Thus, as the stack frame is reused from the previous call to `ret0`, the old return address `main:6` is still on the stack. As a result, `ret0` returns to `main:6` instead of `main:13`, effectively resetting the instruction pointer to an earlier state. However, the return value stored in `rax` is defined by the function `ret1`. Consequently, the call site of the previously-invoked function `ret1` is re-executed, but this time with the return value of `ret0` instead of `ret1`. For the program, it looks like `ret1` returned the value '0' and it continues execution from there. As the stack-related registers `rbp` and `rsp` are callee-saved [39], this re-use of the stale return address does not have any side effects on the stack layout.

5 CacheWarp Building Blocks

In this section, we introduce two building blocks to provide precise control over which writes to drop (cf. Section 4.2). First, we present our page-fault and interrupt-based framework for execution control. For AMD SEV-ES guest VMs, the framework additionally allows single-stepping the VM reliably. Unlike prior work [36], we rely on MTRRs to assign uncacheable memory for VMSA and `wbinvd` to clean the cache before interrupts, resulting in a stable cache state during a context switch. As a result, the runtime of resuming guest VM demonstrates negligible fluctuations, allowing reliable single stepping. Second, based on the result of constructing eviction sets (cf. Section 3.2), the irrelevant dirty data that resides in other cache sets can be evicted proactively. These two building blocks ensure the reliability of CacheWarp.

5.1 Stepping

CacheWarp needs precise control over guest execution, as dropping irrelevant data can have disastrous side effects for the attacker, such as freezing the entire system. Hence, the main challenge of our framework is to minimize the noise of stepping, achieving reliable single-stepping. Previous work did not have to deal with this problem, as attacks were often repeatable without the potential to crash the system. For Intel

SGX, SGX-Step [52] configures the APIC timer such that a malicious OS can single-step the execution of victim enclaves to leak secret information [38, 41, 45, 48, 50, 53]. Compared to SEV, the context switch to and from SGX is relatively fast and nearly constant time, leading to reliable stepping. For AMD SEV, a malicious hypervisor can also hijack the APIC timer, forcing the guest VM to interrupt its execution. While Li et al. [36] show that they can achieve the execution control of AMD SEV-ES VM on a near instruction level, the internal call flow of each stepping is still unclear. Moreover, the malicious hypervisor needs to dynamically calibrate the interval of APIC interrupts, as the context switch from guest to host is not constant-time, resulting in imprecise control. As a result, multi-stepping may occur frequently instead of single-stepping. While a multi-step is endurable in ciphertext side-channel attacks by repeating measurement, this is not the case for CacheWarp.

Overview The framework is implemented as a modified hypervisor that is controlled by user-space controller in the host. The user-space controller communicates the fixed APIC interval, the number of steps to be performed, and a flag to signal the initiation of stepping. After this flag is set, the hypervisor clears the present bit of all guest pages. As a result, any attempt to fetch code from the guest triggers a nested page fault (NPF) due to the missing present bit of the code page. By reading the *exit_info* in the unencrypted VMCB, the hypervisor gets the faulting gPA. The hypervisor's NPF handler then iterates over the nested page table and resets the present bit in the page-table entry for this faulted page. Before resuming guest execution, the hypervisor sets an APIC interval to determine how much progress the guest can make during this stepping. Since the hardware disables interrupts during the context switch [2], an interval lower than a threshold always results in zero-steppings, i.e., the APIC timer interrupt arrives before the guest commits the first instruction. Conversely, an interval exceeding the threshold ensures that the guest retires one or more instructions, known as single-stepping and multi-stepping [52]. To ensure consistent conditions for every stepping, the hypervisor always clears the present bit of the last fault page when handling the APIC timer interrupt, i.e., an NPF is always triggered before the next APIC interrupt.

Similar to the NPF handler, the hardware takes over the context switch when the local APIC timer interrupt arrives. Since AMD SEV-ES, the CPU encrypts the sensitive guest register values before storing them in the VMSA. In addition, guest-state consistency checks are performed before resuming guest execution. While the guest registers are encrypted, the hypervisor can observe the ciphertext of the guest's RIP register to differentiate between zero-step and non-zero-step, i.e., the hypervisor can determine if the guest made progress. Besides, observing the change of other registers helps to infer whether a non-zero-step is single-step or multi-step. Section 8 discusses alternatives to the RIP register as AMD SEV-SNP obfuscates the register ciphertexts preventing localization over the VMSA.

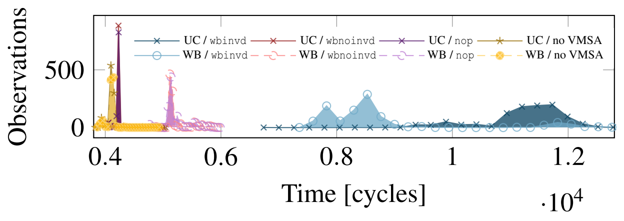


Figure 6: Timing of context switch (zero-step) with different strategies. 'UC' means the hypervisor marks VMSA as uncacheable, and 'WB' marks VMSA as write-back. 'no VMSA' means the hypervisor does not observe the guest `RIP` in VMSA but only sets a tiny APIC interval to guarantee zero-step.

Table 1: The threshold of APIC interval to get single-step.

Uncachable			Write Back		
`nop`	`wbinvd`	`wbnoinvd`	`nop`	`wbinvd`	`wbnoinvd`
74 cycles	146 cycles	78 cycles	114 cycles	146 cycles	114 cycles

Reliability Improvements As shown by Li et al. [36], single stepping is not reliable on AMD SEV-ES due to the non-constant time context switch. The main reason for the observed timing variations are cache effects when loading and storing values to the VMSA memory. Hence, to eliminate the effect of the cache state, we mark the VMSA memory as uncacheable using MTRRs. While we expect a longer time window for a context switch, the consistency check and guest state loading are actually faster, around 900 cycles. Besides, the hypervisor always observes the same VMSA encrypted state even with a large APIC interval that is more than 100 000 cycles. This behavior strongly resembles the observed memory inconsistencies on AMD client CPUs, as shown in Section 3.3.

According to the AMD manual [2], the hypervisor should flush the guest data from all CPU caches before it reads the ciphertext. We hypothesize that when the VMSA page is write-back, reads from the hypervisor force the cached guest data to be encrypted and written back to the memory. However, when the VMSA page is uncacheable, memory coherency is not enforced between the hardware and the hypervisor for this page. The secure processor always saves the plaintext of guest registers into the cache, while the hypervisor reads the stale encrypted VMSA value from memory and no longer flushes the dirty cache lines. The hypervisor always observes the same VMSA value, which leads to a "zero-step".

To experimentally verify this, the hypervisor performs the `wbnoinvd` instruction before reading the encrypted guest-register state to obtain the latest view of the VMSA. We successfully demonstrate the capability of the hypervisor to observe the variations in the ciphertext of the VMSA. To further investigate the effect of cache state on the stepping, we separately perform `nop`, `wbinvd`, and `wbnoinvd` before resuming VMs while marking the VMSA page as uncacheable or write-back. For each stepping, the hypervisor reads the guest RIP in VMSA after timing the context switch (zero-step). Figure 6 shows the timing of the context switch with different strategies. The threshold of APIC interval to get a single-step is shown in Table 1. When the VMSA is write-back, the timing of the context switch is short only if there is no access from the hypervisor. Moreover, if the hypervisor ensures a clean cache state by executing `wbinvd` before the context switch, the stepping takes longer with an uncacheable VMSA. Thus, by relying on uncacheable memory, we can avoid additional flushes caused by reads from the hypervisor before the context switch. As a result, the runtimes of `VMRUN`, i.e., loading guest state and checksum calculation, is highly-predictable, resulting in a significant improvement in stepping. Additionally, the window of APIC intervals that exclusively results in zero- and single-step is extended to dozens of cycles.

Note that before resuming the execution of the guest, a field in VMCB indicates whether the host has passed an interrupt to the guest. Specifically, the guest OS can handle some specific virtual interrupts, e.g., the APIC timer interrupt, before executing the first instruction. However, to ensure that the attacker always zero/single-steps on the target instruction, the malicious hypervisor can choose not to inject such events.

Evaluation. To evaluate the reliability of single-stepping, Figure 8a in Appendix A shows a benchmark program that modifies a different register for each instruction. The `nop` instruction is also used to demonstrate the performance of the precise control of our framework, as it does not have any dependency. For each stepping, we use a bit vector of changes to guest registers as a fingerprint of the current progress of the victim. According to the layout of VMSA, certain registers are grouped in pairs and coexist within a single 16 byte block, which is the encryption unit. Figure 8b illustrates a subset of the registers we include. Each bit represents a modification of this block in the current stepping. Depending on the value of the vector, we can confirm that a multi-step occurs when the changing pattern does not match the expected sequence and ideally recover how many instructions are executed in the multi-step as well. Note that in realistic scenarios, we cannot directly recognize a multi-step as the guest instruction sequence is unknown to the attacker. While fingerprinting could work, here we just use the benchmark program to evaluate the reliability of single-steps. We observe 10 000 single-steps with 1951 zero-steps and 0 multi-steps, using a fixed interval of 220 cycles as the APIC interval. Concurrent work [58] uses performance counters to distinguish zero-, single-, and multi-steps on SEV-SNP, guaranteeing reliable single-stepping as well. This technique could also be added to our framework.

5.2 Selective State Drop

As the second building block, we show how the hypervisor can selectively drop dirty cache lines, i.e., partially writing back the cache before its invalidation. This step is indispensable, ensuring the passing of the integrity check of the VMSA. Our experiments show that the checksum calculation is always finished before the hypervisor takes over the execution. When the hypervisor executes the `invd` instruction after zero-stepping, the AMD SEV VM can still resume its execution afterward. However, if the chosen interval leads to a single-step or multi-step, the register state of the guest within the VMSA (at least the block containing the guest RIP) changes and is in the modified cache state. Thus, the hypervisor cannot simply execute the `invd` instruction after single/multi-steps, as it immediately leads to a crash of the guest VM.

To achieve a selective state drop, the attacker must keep the dirty cache lines that the attacker intends to discard in the cache but evict all other cached data back to memory. Hence, the attacker first allocates the eviction buffer for all cache sets. Note that this step only needs to be performed once, e.g., before the VM creation. If the write the attacker wants to drop resides in the private cache, i.e., L1 or L2 cache, an eviction on the L2 cache must first be performed due to the non-inclusive shared LLC. To ensure all irrelevant dirty cached data within the scope of `invd` is written back to memory in advance, the attacker performs an L2 eviction for all L2 sets except for the target set. Next, as the unrelated data is now evicted into the LLC cache, the attacker performs an LLC eviction to write back these modified values to the memory.

While the exact target address is unknown to the hypervisor, the hypervisor can track it on a page level. Similar to how the page address for a code fetch is recorded, the hypervisor can retrieve the gPA of an NPF triggered by a write operation. By iterating over the nested page table, the hPA for the fault page, which determines the cache index, can be read to refine the potential cache indexes of the target address.

Finally, to avoid a race condition between writes from other physical cores and cache invalidation, the attacker can offline the other physical cores within the CCX. To ensure there is no other dirty data in the internal cache of the logical sibling core, the attacker can disable hyperthreading. Moreover, the attacker can also disable hardware prefetchers and interrupts as well as fix the CPU frequency during eviction to obtain a denoised environment. After evicting all the dirty cached values the attacker wants to keep, the attacker executes the `invd` instruction to drop the remaining writes.

Evaluation. We evaluate the accuracy of selectively dropping writes on a single L2 cache set both on AMD SEV-ES and SEV-SNP. The test program in the VM first allocates a 2MB huge page to ensure that it occupies memory belonging to all L2 sets. We randomly choose an address within the range and pass it to the for-loop code as listed in Figure 7. The code executes a read-add-write pair for the given address

```
1   for (int i = 0; i < 10000000; i++) {
2       asm volatile (
3       ".rept 4000\n"
4       "movq (%0), %%r11\n"
5       "addq $1, %%r11\n"
6       "movq %%r11, (%0)\n"
7       ".endr\n"
8       :: "r"(addr) : "memory");
9   }
```

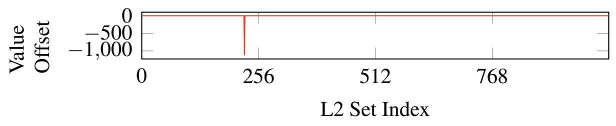


Figure 7: Evaluation for selectively dropping memory writes. For each L2 set, 1000 steppings are repeated separately, followed by invalidation on the entire L2 set per step. The L2 set index of the target address is 225. Only when the correct L2 set remains in the cache, the writes are dropped.

in each iteration, iterating a total of 4000 times. We use the `O2` optimization flag, such that the loop counter resides in a register instead of utilizing stack memory. Hence, the only point where we can drop the modified cache state is `movq %%r11, (%0)` (Line 6).

We choose the APIC interval that leads to multi-stepping. For each stepping, the guest can approximately complete one iteration of memory read and write before the context switch caused by the interrupt. For example, on AMD EPYC 7252, the L2 cache entry of the target can potentially be stored in one of the 16 LLC sets following L2-prime. Hence, to drop the dirty cached data in a different L2 cache set each time, the hypervisor performs a complete LLC eviction for all LLC sets except for 16 specific LLC sets in each measurement. Note that the LLC eviction needs to contain the step of L2 eviction, due to the non-inclusive LLC cache. The hypervisor only requires more LLC eviction addresses as they have the same bits with L2 eviction addresses for the L2 cache index. As there are 1024 L2 cache sets, we assume that the hypervisor can observe the value offset only in one measurement when the target data resides in that L2 cache set that is dropped. Figure 7 shows that only when the correct L2 set remains in the cache, the writes are dropped. Otherwise, the data is written back and updated in the memory before the cache invalidation.

6 Case Studies

In this section, we present 3 end-to-end attacks using CacheWarp as our case studies. In Section 6.1, we demonstrate an attack on the RSA-CRT implementation of the Intel IPP cryptographic library, extracting the full private key within 10 s on average. In Section 6.2, we show that CacheWarp can break the password authentication of the OpenSSH server, resulting in full (unprivileged) access to the guest VM. In Section 6.3, we show that the `sudo` binary can further be exploited using CacheWarp to escalate the privileges of the attacker.

6.1 Breaking RSA-CRT via Bellcore Attack

In the first case study, we show that CacheWarp can perform a Bellcore attack [10, 12] on RSA-CRT, fully recovering the private key given a single faulty signature.

RSA and Bellcore Attack In an RSA signing scheme, the signature S is computed via the modular exponentiation $S = x^d \pmod N$, with x the message, d the secret signing exponent, and N the public modulus. As the involved numbers typically have multiple thousand bits, this exponentiation is computationally expensive. To improve performance, the computation can be split into two parts $S_1 = x^d \pmod p$ and $S_2 = x^d \pmod q$, with p and q the secret prime factors of $N = p \times q$. The final signature S can then be constructed from S_1 and S_2 via the Chinese Remainder Theorem (CRT) [44].

While more efficient than a standard RSA implementation, RSA-CRT is especially vulnerable to fault attacks [10, 12]. A single fault that occurred during the computation of S_1 but not during the computation of S_2 is sufficient to recover the private key (p, q). Given two signatures, one such faulty signature $\hat{S}$ and one correctly-computed signature S over the same message, an attacker can compute the private-key part q as $gcd(S - \hat{S}, N) = q$ and subsequently $p = \frac{N}{q}$. This equation holds as $S = \hat{S} \pmod q$ but $S \neq \hat{S} \pmod p$. Hence, such a fault allows an attacker to forge arbitrary signatures.

Threat Model In line with the AMD SEV threat model, our victim is a VM guest protected by AMD SEV running on an attacker-controlled hypervisor. The victim provides an interface to get the signature of data. We do not make any assumptions on the signed data; we only require that the same data can be signed twice. The attacker does not have to choose the data or know the data that is signed. For our attack, the victim uses the RSA-CRT implementation of the Intel IPP cryptography library, as also shown in previous microarchitectural attacks [38,43,54]. The goal of the attacker is to extract the private signing key of the victim.

Attack Flow The attacker queries a non-faulty signature S for an arbitrary message, which is used later to recover the private key. For the attack, the attacker uses CacheWarp to drop the write to a cache line containing the RSA private key during the RSA-CRT computation. This dropped write has the effect that the computation uses stale data instead of the private key. Note that for the Bellcore attack, it does not matter how the computation is faulted. Hence, any stale data is sufficient for getting a faulty signature. Still, the attacker has to ensure that only one of the partial signatures S_1 and S_2 contains a fault. An attacker has 2 options to achieve that. The clean way is to rely on our single-stepping framework (cf. Section 5.1). With single-stepping, the attacker has fine-grained control over which write is dropped. As a result, an attacker can reliably induce the desired fault. However, in practice, it is sufficient to "blindly" drop (multiple) cache lines. Due to the robustness of the Bellcore attack, such a non-synchronized attack is possible. As the attacker only has to induce a single fault and can retry the attack arbitrarily often, only limited precision is needed. Therefore, targeted eviction strategies for unrelated data are unnecessary.

Results and Implications We successfully mount the Bellcore attack on an SEV-ES machine and an up-to-date SEV-SNP machine (AMD EPYC 7313P), both running Ubuntu 22.04 with Linux kernel 6.1.0. Note that no extra steps are needed to adapt the Bellcore attack to SEV-SNP, as it does not rely on single-stepping. The victim VM runs Ubuntu 20.04 and the newest Intel IPP library 2021.8 from July 27, 2023 (commit `36e76e2`). For SEV-ES, we mount the attack 10 times and check whether the resulting signature has changed.[1] In 9 tries, we drop a write that leads to a faulty signature. 100 % of these faulty signatures allow us to correctly recover the private key using the Bellcore attack. Hence, the average overall success rate is 90 %. As one try takes on average 6 s, we get an expected success rate of 99 % after 12 s. The recovery time using the Bellcore attack is negligible, with on average below 0.1 s. For SEV-SNP, we perform 100 "blind" drops, i.e., we do not use single-stepping but mount CacheWarp at randomes times. Of these tries, 28 result in an exploitable faulty signature and only one causes a system crash. The remaining 71 tries have no effect. Given that each attempt takes less than 2 s, an attacker can easily replicate this attack.

In contrast to other software-based fault attacks, such as Rowhammer [30] and undervolting [29, 43, 46], CacheWarp does not rely on manufacturing differences of hardware. Moreover, CacheWarp can accurately choose the time of the fault as well as the location of the fault. Thus, CacheWarp deterministically works on all vulnerable host machines. Similarly to this attack on RSA-CRT, all cryptographic schemes vulnerable to a fault injection during memory load can be attacked using our primitive. As related work indicates, fault attacks can be a powerful primitive against symmetric [20] and asymmetric [11, 12] cryptographic implementations. Therefore, we highly encourage exploring the impact of our primitive in the context of fault attacks, especially as faulting to stale data is typically not a fault model that is considered [17, 23, 31].

[1]The attacker randomly chooses one or multiple cache lines to drop after a 500 cycles multi-step.

6.2 Bypassing OpenSSH Authentication

In this second case study, we use CacheWarp to log into a password-protected OpenSSH server in a VM guest protected by AMD SEV without knowing the password.

Threat Model In line with the AMD SEV threat model, our victim is a VM guest protected by AMD SEV running on an attacker-controlled hypervisor. The attacker does not have code execution in the victim VM. We assume that an OpenSSH server runs in the victim VM. We assume that the SSH server allows authentication via a password, although we strongly suspect public-key authentication to be vulnerable as well. The attacker runs as the VM host with hypervisor privileges and can request an SSH connection to the victim, either directly from the hypervisor or from a different machine. The goal of the attacker is to connect to the VM without possessing valid SSH credentials.

Attack Flow To bypass the SSH authentication, the attacker mounts CacheWarp on the `sys_auth_passwd` function in OpenSSH. If this function returns '1', the user is authenticated, and the SSH connection is established. More precisely, the attacker uses the Timewarp primitive from Section 4.4 to force a password comparison to always succeed. Appendix D shows the relevant parts of the code. By invoking a Timewarp at the `xcrypt` call (Line 9), the control flow is transferred to the return point of the last function call, which is the `shadow_pw` call (Line 5). The `xcrypt` call returns a pointer to the hashed user input on success, and the attacker can provide an arbitrary nonempty input to `xcrypt` by using a password of their choice. Using the Timewarp primitive, the attacker jumps to the return of the `shadow_pw` call overwriting the `pw_password` variable with the return value from the previous xcrypt call. Therefore, from this point on, it holds that `pw_password` = `encrypted_password` as only the first two characters of the salt are relevant which are always the same. At this point, an attacker is authenticated with a wrong password and can execute arbitrary code in the VM.

Challenges While the basic flow of the attack is simple to understand, its execution remains challenging. Most importantly, the Timewarp primitive must be invoked precisely at the `call` instruction responsible for calling the `xcrypt` function. To single-step to this exact location, we utilize a combination of a page-fault attack and register-usage detection (cf. Section 5.1) to find the correct trigger point. By performing a page-fault attack, we can identify when the control flow reaches the code page containing `sys_auth_passwd`. As this code page can also contain other code, we utilize register-usage detection to identify a register usage pattern that only occurs before the `xcrypt` call. This combination of page-fault attack and register-usage detection allows us to precisely identify the point at which the `xcrypt` function is called.

Results We perform the end-to-end attack on an AMD EPYC 7252, running Ubuntu 22.04. The victim VM runs Ubuntu 20.04 and the OpenSSH server version 8.2p1 which is currently the default version. Our attack succeeds in 10 of 10 test runs. A theoretical failed attack means locating the target instruction fails due to potential multi-steps occurring in the locating period. Thus, the cache is not invalidated, resulting in no crash and the possibility for the attacker to retry the attack. In addition, the entire attack takes less than 10 s, making it highly practical. As VM guests typically expose an SSH connection to allow users to connect to them, this attack presents a high-risk factor for AMD SEV-enabled VMs and completely breaks their security guarantees.

6.3 Bypassing Sudo Authentication

In this third case study, we demonstrate how CacheWarp can be exploited to escalate unprivileged code execution in a guest VM to privileged code execution. Such a scenario is realistic after exploiting, e.g., an OpenSSH server using CacheWarp (cf. Section 6.2) or any other attack. In this scenario, a typical next step for an attacker is to escalate privileges on the machine, thus gaining full control over the system. We demonstrate that an attacker can reliably exploit `setuid` binaries using CacheWarp, and, by that, elevate privileges to the superuser. We demonstrate such an exploit on `sudo` version 1.8.31 as found on Ubuntu 20.04.

Threat Model We assume that the attacker has unprivileged native code execution in the AMD SEV VM. Additionally, in line with the AMD SEV threat model, the attacker controls the hypervisor. We assume that the `sudo` Linux utility is installed in the VM, which is common for Linux distribution.

Attack Flow The general idea is to trick the user check in the `sudo` utility, such that the executing user seems to be the root user, skipping any other checks. On a high level, the `sudo` utility works as follows: The program first queries information about the currently executing user, e.g., their EUID, RUID, and GUID. The `sudo` program uses the stored information to look up the user's permission in the sudoers file. If the permission checks based on the retrieved information pass, `sudo` executes the command specified by the user as root.

For the exploit, we target the code that retrieves the information about the currently executing user. Listing 1 in Appendix B shows the assembly code responsible for retrieving and storing this information. After each query of a user-related ID, the information is stored in a zero-initialized C struct. This struct is accessed by a single write to the corresponding offset for each entry. In our exploit, we drop the write for the real user ID (RUID). As the memory is zero-initialized, this results in an RUID of zero, which reflects the RUID of the root user. The `sudo` program does not require additional checks if it is already running as the root user. Thus, `sudo` allows the user to execute arbitrary commands with the highest privileges.

Challenges Compared to the other case studies (cf. Section 6.1 and Section 6.2), the primitive required for this exploit, including the setup, is much simpler. The main challenge is finding the exact point in time to drop the write. For this, we inspect the register usage of the glibc implementation of `getuid` and search for exactly this pattern using our single-stepping framework. Once we identify the pattern, we can count the remaining few instructions before the desired write instruction to drop it.

Results We evaluate our exploit on an AMD EPYC 7252 with Ubuntu 20.04 running Linux kernel 5.15.0 in AMD SEV. We observe a success rate of 99 % for 100 executions. If the exploit fails, the attacker can just mount the exploit again by restarting the `sudo` binary. We observe that our exploit finishes within 20 seconds. Compared to the Rowhammer-based `sudo` exploit from Gruss et al. [22], which takes more than 44.4 h, our attack is practical and achieves a speedup of 4 orders of magnitude.

7 Countermeasures

In this section, we discuss potential mitigations for CacheWarp on the hardware, firmware, and software layer.

Hardware Ultimately, CacheWarp has to be fixed on the hardware level. One solution is to prevent the `invd` instruction from being used if AMD SEV is enabled. CacheWarp requires this specific privileged instruction that is only necessary for a few very specific scenarios [25]. However, none of these scenarios should occur under the regular use of AMD SEV, and we did not find any use of this instruction in the Linux kernel or Xen hypervisor. Hence, making the use of `invd` and AMD SEV mutually exclusive would prevent CacheWarp. On Intel CPUs, the `invd` instruction is mutually exclusive with Intel SGX, i.e., if SGX is enabled, `invd` simply raises an exception. Such a hardware change also ensures backward compatibility, as `invd` is still available to legacy systems that do not support SGX.

Alternatively, if the `invd` instruction should still be available with AMD SEV, the CPU could automatically write back the cache state on VM interrupt. Similar to TLB flushes, the AMD SEV VM could set a bit in VMSA to indicate the hardware needs to write all modified cache lines back to the main memory. While the writes to the VMSA could still be dropped, there is an integrity check over the whole VMSA during the `vmrun` instruction, already preventing this case.

Firmware In contrast to hardware fixes, firmware workarounds can act as short-term mitigations against

CacheWarp. Such firmware updates can be done using microcode updates via the BIOS or the operating system. Microcode updates can hook instructions to change their behavior [13, 33]. Such a hook can be used to prevent the `invd` function from invalidating the cache if AMD SEV is active. While such a hook leads to a slight performance penalty for the instruction, we expect this to not be a problem, as this instruction is nearly never used.

Alternatively, a microcode update could hook the access to bit 4 of MSR (`0xc0010015`). This bit controls whether the `invd` actually invalidates cache content or acts the same as the not-exploitable `wbinvd` instruction (cf. Section 3.1). As microcode can introduce new MSRs [32], we expect such a change to be possible. Hence, a microcode update could prevent changing the `invd` behavior if AMD SEV is used.

Software We implement a proof-of-concept software-only mitigation for guest VMs in LLVM's *clang*. Appendix C details the compiler's implementation and security analysis. The main idea is to ensure that writes end up in the main memory before they are used again. This can be ensured by forcing a write-back of a modified value using the unprivileged and untrapable `clwb` instruction. In contrast to `clflush`, this instruction writes back the modified cache line without flushing it from the cache. However, there is still a race condition if the hypervisor interrupts the guest VM between the write and the `clwb` instruction. Hence, the compiler must add a read-back of the value *after* the `clwb` to ensure that the write was not dropped before the `clwb` instruction. If the read value is not the same as the written value, the VM can abort, as this is not possible under normal bug-free execution.

Our mitigation is inspired by Intel's compiler-based LVI mitigation for SGX [31, 51] that had to solve a similar problem. While the LVI mitigation has to prevent speculative execution after any load, our mitigation has to write back and verify every write. Still, with this strong attacker model of AMD SEV, we do not see any other software-only mitigation that is effective in the presence of a malicious hypervisor. We evaluate the overhead of our proof-of-concept mitigation on *nbench* [16, 19, 31] compiled with the `O0` optimization flag so that most writes are explicit. An average overhead of factor 193.51 over the benchmarks is introduced by protecting explicit writes and `push` instructions in the prolog. We leave it to future work to implement a feature-complete version of the compiler (Appendix C), as this engineering effort is out of the scope of this paper. We experimentally verify our compiler mitigation against *selective write dropping* (Section 5.2) and no longer observe write drops. Finally, the presence of the `clwb` instruction is already drastically reducing the probability of injecting a fault.

8 Discussion

Limitation AMD addressed the Cipherleaks side channel [36] in AMD SEV-SNP by adding freshness to the VMSA when storing the register context in memory, obfuscating the ciphertext. Thus, we can no longer rely on *register changing* patterns (Section 5.1) to evaluate steppings and identify target instructions as they are no longer constant based on the registers. However, the performance counters are accessible to track the runtime information for each stepping [58]. Other patterns with some chosen events can be used to identify target instructions, which only requires engineering effort as reliable single-step was demonstrated on SEV-SNP [58].

Differences between SEV-ES and SEV-SNP for CacheWarp The underlying vulnerability of CacheWarp is the same for SEV-ES and SEV-SNP. While SEV-SNP introduces new hardware-based security features, most side channels, including page faults and Prime+Probe [6], are not mitigated. As a result, on SEV-SNP, an attacker can still selectively drop dirty cache lines for each stepping (Section 5.2) using eviction. Similarly, since the `wbinvd` instruction and MTRRs are available on SEV-SNP, an attacker can adopt the same strategy as used on SEV-ES to improve stepping. The only difference between CacheWarp on SEV-ES and SEV-SNP is the method for precisely locating the drop target, i.e., the technique for single-stepping.

Related Work Before the introduction of AMD SEV-SNP, several attacks have been proposed. Hetzelt and Buhren [24] showed the first flaws in the design of AMD SEV already before it was widely available in CPUs. They demonstrated control-flow modifications via the back-then unencrypted VM control block (VMCB). Morbitzer et al. [42] exploit the fact that before the introduction of SEV-SNP, a malicious hypervisor could control page mappings in the VM. They tricked the VM into decrypting sensitive memory and returning it to the attacker as the content of a legitimate request posted to a web or SSH server in the victim VM. As the underlying victim operating system relies on the correctness of the provided address mapping, the content of the sent memory pages is not further checked before transmitting them to the attacker. Radev and Morbitzer [47] manipulate external hypervisor-controlled interfaces to an AMD SEV VM degrading hardware randomness in the victim machine, injecting and exfiltrating data, and finally gaining code execution in the victim VM. Wilke et al. [57] show that using an encryption oracle attacker-controlled data can be injected into the victim VM, again leading to code execution for the attacker. Li et al. [35] abuse the information leakage on address translation faults to leak pages from inside of the AMD SEV-protected VM to an attacker. In contrast to our work, these discussed attacks either break with the introduction of AMD SEV-SNP [24, 42, 57] or only theoretically discuss applications on SEV-SNP [35, 47].

Li et al. [36] showed that even with confidentiality and integrity guarantees, the confidentiality of VMs can be violated using side-channel attacks on the ciphertext. While their work violated the confidentiality of the AMD SEV-SNP VMs, our attack additionally breaks the integrity of the VMs as we achieve arbitrary code execution. With hardware access to the hypervisor, Buhren et al. [15] showed that the endorsement keys can be extracted, allowing an attacker to fake remote attestation reports. In contrast to them, we target the software in the VM directly and do not require hardware access.

9 Conclusion

In this paper, we presented CacheWarp, a new software-based fault attack on AMD SEV-ES and SEV-SNP that enables architectural rollbacks to previous (stale) states. Unlike previous attacks, CacheWarp is not mitigated on the newest SEV-SNP implementation, and it does not rely on the specifics of the guest VM. We introduced a robust single- and zero-stepping framework for synchronizing attacker and victim, which we use in 3 case studies. We demonstrated a full key recovery of RSA-CRT in the Intel IPP crypto library, authentication-less login to an OpenSSH server, and privilege escalation via the `sudo` binary. Although we devised a proof-of-concept compiler-based software mitigation, we argue that only hardware, and possibly firmware, mitigation can efficiently mitigate CacheWarp.

Acknowledgment

We want to thank our shepherd and the anonymous reviewers for their guidance, comments and valuable suggestions, as well as Leon Trampert and Moritz Lipp for helpful feedback on this work. Ruiyi Zhang thanks Xinyue Shen for fruitful discussions and suggestions that helped improving the paper. This work was supported in part by Semiconductor Research Corporation (SRC) Hardware Security Program (HWS). This research is supported in part by the European Research Council (ERC project FSSec 101076409). Additional funding was provided by a generous gift from AWS. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] *Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh*, 3rd ed., Advanced Micro Devices Inc., 7 2018.

[2] "AMD64 Architecture Programmer's Manual," Advanced Micro Devices Inc., 2023.

[3] *Processor Programming Reference (PPR) for AMD Family 19h Model 11h, Revision B1 Processors*, 0th ed., Advanced Micro Devices Inc., 5 2023.

[4] "AMD SEV-SNP," Amazon Web Services, 2023. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html

[5] "AMD Infinity Architecture Technology," AMD, 2023. [Online]. Available: https://www.amd.com/en/technologies/infinity-architecture

[6] "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," AMD, 2023. [Online]. Available: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[7] "AMD "ZEN" Core Architecture," AMD, 2023. [Online]. Available: https://www.amd.com/en/technologies/zen-core

[8] AMD, "AMDSEV - Software for SEV on GitHub," 2023. [Online]. Available: https://github.com/AMDESE/AMDSEV

[9] "Protecting VM Register State With SEV-ES," AMD, 2023. [Online]. Available: https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf

[10] 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.

[11] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, 2006.

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

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

[14] N. Brookwood, "EPYC: A Study in Energy Efficient CPU Design," Insight 64, 2018. [Online]. Available: https://www.amd.com/system/files/documents/The-Energy-Efficient-AMD-EPYC-Design.pdf

[15] 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.

[16] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, “Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017.

[17] T. Fuhr, É. Jaulmes, V. Lomné, and A. Thillard, “Fault attacks on aes with faulty ciphertexts only,” in *FDTC*. IEEE, 2013.

[18] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, “Efficient and Generic Microarchitectural Hash-Function Recovery,” in *S&P*, 2024.

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

[20] C. Giraud and H. Thiebeauld, “A survey on fault attacks,” in *CARDIS*, 2004.

[21] “Confidential VM documentation,” Google Cloud, 2023. [Online]. Available: https://cloud.google.com/compute/confidential-vm/docs

[22] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O’Connell, W. Schoechl, and Y. Yarom, “Another Flip in the Wall of Rowhammer Defenses,” in *S&P*, 2018.

[23] L. Hemme, “A differential fault attack against early rounds of (triple-) des,” in *CHES*, 2004.

[24] F. Hetzelt and R. Buhren, “Security analysis of encrypted virtual machines,” *ACM SIGPLAN Notices*, 2017.

[25] Intel, “Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide,” 2023.

[26] Intel, “Intel Trust Domain Extensions,” 2023. [Online]. Available: https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf

[27] Intel Corporation, “Software Guard Extensions Programming Reference, Rev. 2.” 2014.

[28] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Cross processor cache attacks,” in *AsiaCCS*, 2016.

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

[30] 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.

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

[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] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, “Reverse engineering x86 processor microcode.” in *USENIX Security Symposium*, 2017.

[34] 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.

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

[36] 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 Symposium*, 2021.

[37] 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 *AsiaCCS*, 2020.

[38] 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.

[39] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, “System V Application Binary Interface,” 2013.

[40] “Azure confidential computing,” Microsoft Azure, 2023. [Online]. Available: https://learn.microsoft.com/azure/confidential-computing/

[41] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, “CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction,” in *USENIX Security Symposium*, 2020.

[42] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, “Severed: Subverting amd’s virtual machine encryption,” in *EuroSec*, 2018.

[43] 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.

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

[45] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend," in *USENIX Security Symposium*, 2021.

[46] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults," in *AsianHOST*, 2019.

[47] M. Radev and M. Morbitzer, "Exploiting interfaces of secure encrypted virtual machines," in *Reversing and Offensive-oriented Trends Symposium*, 2020.

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

[49] P. Rogaway, "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC," in *ASIACRYPT*, 2004.

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

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

[52] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Workshop on System Software for Trusted Execution*, 2017.

[53] ——, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *CCS*, 2018.

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

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

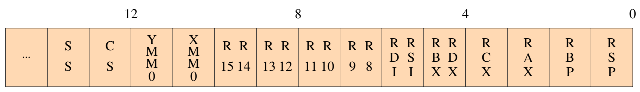
[56] L. Wilke, "SEV Step," 2023. [Online]. Available: https://github.com/sev-step/sev-step

[57] 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.

[58] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "Sev-step: A single-stepping framework for amd-sev," 2023.

```
300f: nop                  ; 0x0
3010: push   %r11          ; 0x1
3012: add    $0x1,%r11     ; 0x80
3016: add    $0x1,%r9      ; 0x40
301a: add    $0x1,%rdx     ; 0x10
301e: add    $0x1,%rcx     ; 0x8
3022: add    $0x1,%rax     ; 0x4
3026: sub    $0x1,%rax     ; 0x4
302a: sub    $0x1,%rcx     ; 0x8
302e: sub    $0x1,%rdx     ; 0x10
3032: sub    $0x1,%r9      ; 0x40
3036: sub    $0x1,%r11     ; 0x80
303a: pop    %r11          ; 0x1
303c: jmp    300f          ; 0x0
```

(a) Register-changing pattern used in the benchmark program



(b) The vector to record register changes

Figure 8: Register-changing observation

A Register-changing Observation

Figure 8a shows a benchmark program that modifies a different register for each instruction. Figure 8b illustrates a subset of the registers we included. We evaluate the reliability of single-stepping on SEV-ES by observing the register-changing pattern of the benchmark program.

B Sudo Gadget

Listing 1 shows the exploited assembly code in the `sudo` case study Section 6.3.

```
[...]
xor  %edi, %edi
call getsid
mov  %eax, (user_details_struct.sid)
call getuid
mov  %eax, (user_details_struct.ruid) ; <-- drop
call geteuid
mov  %eax, (user_details_struct.euid)
call getgid
mov  %eax, (user_details_struct.rgid)
[...]
```

Listing 1: Assembly code responsible for retrieving information about the currently executing user in the `sudo` program.

C Compiler Mitigation

In this section, we detail the proof-of-concept compiler to mitigate CacheWarp. We implement the compiler in the LLVM framework as a machine function pass and iterate over all assembly instructions to check if the instruction is an explicit store or push. Figure 9a shows the emitted instructions to protect a store instruction. First, the compiler inserts a clwb instruction to write back the dirty cache line. Second, we emit a comparison instruction between the memory operand and the original register. Finally, if the values are not equal, we are under attack and abort further execution. We focus only on the store instructions in the proof-of-concept. However, the compiler can be extended to additional instructions like call but this engineering effort is out-of-scope for this work.

We assume an attacker with single-stepping capabilities can inject the invd instruction after each instruction in the victim. Figure 9b shows the first case when the invd arrives after the store and drops the dirty cache line. However, the comparison detects this due to the mismatch of the memory and the desired register value and aborts further execution. Figure 9c shows the second case when the invd arrives after the clwb instruction. Here the invd has no effect as the cache line's content is already written back. In this case, the victim cannot detect that the attacker tried to fault attack the program since no fault was injected. We experimentally verify these two cases and confirm that the first is detected by the compare instruction and the second is ineffective.

```
1     mov    %rcx,(%rdi)
2     clwb   (%rdi)
3     cmp    %rcx,(%rdi)
4     jne    _abort
5
```

(a) The compiler protects the store in Line 1 by inserting instruction in Lines 2 to 4.

```
1     mov    %rcx,(%rdi)
2     invd   // HV injected
3     clwb   (%rdi)
4     cmp    %rcx,(%rdi)
5     jne    _abort
6
```

(b) The injected invd in Line 2 effectively drops the current dirty cache line. However, the comparison with the original register value detects the fault attack.

```
1     mov    %rcx,(%rdi)
2     clwb   (%rdi)
3     invd   // HV injected
4     cmp    %rcx,(%rdi)
5     jne    _abort
6
```

(c) The injected invd in Line 3 is ineffective as the content is already written back to the main memory.

Figure 9: The compiler-generated code to protect a store instruction and the possible attack scenarios a malicious hypervisor could exploit in the AMD SEV threat model.

D OpenSSH Gadget

Listing 2 shows the exploited source code in the OpenSSH case study Section 6.2.

```
// Returns 1 if user authenticated 0 else
int sys_auth_passwd(struct ssh *ssh, const char *password) {
  [...]
  char *encrypted_password, *salt = NULL;
  char *pw_password =  shadow_pw(pw);
  [...]
  if (authctxt->valid && pw_password[0] && pw_password[1])
    salt = pw_password;
  encrypted_password = xcrypt(password, salt);
  return encrypted_password != NULL &&
      strcmp(encrypted_password, pw_password) == 0;
}
```

Listing 2: Simplified source code of the sys_auth_passwd function from OpenSSH.