TLB Poisoning Attacks on AMD Secure Encrypted Virtualization

Mengyuan Li
The Ohio State University
Columbus, Ohio, USA
li.7533@osu.edu

Yinqian Zhang ✉
Southern University of Science & Technology
Shenzhen, Guangdong, China
yinqianz@acm.org

Huibo Wang
Baidu Security
Sunnyvale, California, USA
wanghuibo01@baidu.com

Kang Li
Baidu Security
Sunnyvale, California, USA
kangli01@baidu.com

Yueqiang Cheng
NIO Security Research
San Jose, California, USA
yueqiang.cheng@nio.io

ABSTRACT

AMD's Secure Encrypted Virtualization (SEV) is an emerging technology of AMD server processors, which provides transparent memory encryption and key management for virtual machines (VM) without trusting the underlying hypervisor. Like Intel Software Guard Extension (SGX), SEV forms a foundation for confidential computing on untrusted machines; unlike SGX, SEV supports full VM encryption and thus makes porting applications straightforward. To date, many mainstream cloud service providers, including Microsoft Azure and Google Cloud, have already adopted (or are planning to adopt) SEV for confidential cloud services.

In this paper, we provide the first exploration of the security issues of TLB management on SEV processors and demonstrate a novel class of TLB Poisoning attacks against SEV VMs. We first demystify how SEV extends the TLB implementation atop AMD Virtualization (AMD-V) and show that the TLB management is no longer secure under SEV's threat model, which allows the hypervisor to poison TLB entries between two processes of a SEV VM. We then present TLB Poisoning Attacks, a class of attacks that break the integrity and confidentiality of the SEV VM by poisoning its TLB entries. Two variants of TLB Poisoning Attacks are described in the paper; and two end-to-end attacks are performed successfully on both AMD SEV and SEV-ES.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**; **Side-channel analysis and countermeasures**; **Trusted computing**; **Virtualization and security**.

KEYWORDS

Trusted execution environments; Secure Encrypted Virtualization; Cloud security; TLB management

✉Corresponding authors

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485876

ACM Reference Format:
Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3485832.3485876

1 INTRODUCTION

AMD's Secure Encrypted Virtualization (SEV) is a new security feature of AMD Virtualization (AMD-V) [5] that prevents the privileged cloud providers from manipulating or inspecting the data and applications of cloud tenants. It supports full virtual machine encryption through a hardware memory encryption engine and a secure co-processor (*i.e.*, AMD-SP) that transparently manages the hardware encryption keys. Compared to similar technology like Intel Software Guard Extension (SGX) [8], SEV is more advantageous in its ease of adoption without the need of altering software to be protected. So far, SEV has been adopted in Azure Cloud and Google Cloud as a backend of confidential cloud computing services [10, 26].

Nonetheless, numerous works have shown that SEV is vulnerable in several aspects: First, the VM control block (VMCB) used to store register values and control information is not encrypted during context switch, allowing a malicious hypervisor to manipulate or inspect the register values of guest VMs, which would lead to a complete breach of confidentiality or integrity of guest VMs [12, 30]. To counter these threats, AMD released SEV Encrypted State (SEV-ES) [16]. In SEV-ES, the register states in VMCB would be encrypted and saved in the VM Save Area (VMSA) during the world switch, leaving no chance of these attacks for the malicious hypervisor. Necessary register values are passed through a Guest-Hypervisor Communication Block (GHCB), which is not encrypted.

Second, neither SEV nor SEV-ES protects the integrity of encrypted memory and nested page tables (NPT). Therefore, the malicious hypervisor could replay the encrypted memory blocks or change the mapping of guest memory pages in the nested page tables to breach the security of SEV and SEV machines [7, 9, 22, 27, 28, 32]. To mitigate these attacks, AMD recently released the third generation of SEV—SEV Secure Nested Paging (SEV-SNP) [3], which introduces a Reverse Map Table (RMP) and a mechanism of page validation to prevent malicious modification of nested page tables by tracking the ownership of the memory. As AMD claims,

SEV-SNP provides strong integrity protection for the guest VMs, and hence mitigates all these attacks.

Third, SEV, including SEV-ES and SEV-SNP, allows the untrusted hypervisor to manage the address space identifier (ASID), which is used to control the VM's accesses to the encrypted memory. The principle adopted by AMD is a "security-by-crash" design, which assumes that mismatch of ASID could lead to VM crashes and hence guarantee the security of the guest VMs. However, the abuse of ASIDs has been exploited by the Crossline attacks, which leverage the short window before VM crashes to leak secret data through page faults or to execute instructions that form decryption and encryption oracles [21].

This paper outlines a new category of security attacks against SEV, namely TLB Poisoning Attacks, which enable the adversary who controls the hypervisor to poison the TLB entries shared between two processes of the same SEV VM. The root cause of TLB Poisoning Attacks is that the hypervisor is in control of the TLB flushes by the design of AMD SEV. Specifically, because TLB is tagged with ASIDs to distinguish the TLB entries used by different entities, unnecessary TLB flushes can be avoided during the world switches (VMEXIT and VMRUN between the guest VM and the hypervisor) or the context switches (context switches between the process hosting the guest VM's current virtual CPU (vCPU) and other processes). As it is difficult for the CPU hardware to determine whether to flush the entire TLB or only TLB entries with certain ASIDs, the TLB flush is solely controlled by the hypervisor. The hypervisor can inform the CPU hardware to fully or partially flush the TLB by setting the TLB control field in the VMCB, which will take effect after VMRUN. As such, the adversary can intentional skip TLB flushes, such that a victim process of the victim SEV VM may unwillingly use the TLB entries injected by another process of the same VM.

Two attack scenarios of TLB Poisoning attacks are considered in this paper: (1) With the help of an unprivileged attacker process running in the targeted SEV VM, the adversary is able to poison the TLB entries used by a privileged process and alter its execution. (2) Without the help of a process directly controlled by the adversary, the adversary could still exploit the misuse of TLB entries on a network-facing process (not in his control) that share the same (or similar) virtual address space with the targeted process and bypass authentication checks. We have demonstrated two end-to-end attacks against two SSH servers to show the feasibility of the two attack scenarios, respectively, on an AMD EPYC Zen processor that supports SEV-ES.

Responsible disclosure. We have disclosed the vulnerability that enables TLB Poisoning Attacks to AMD via emails in December 2019. After an in-depth teleconference discussion with the SEV team, we have been confirmed that the vulnerability exists on SEV and SEV processors, but the upcoming SEV-SNP has a new feature that prevents the attack. Therefore, AMD will not release a patch for the discovered vulnerability but will rely on the new SEV-SNP processor as a line of defense.

Contributions. The paper makes the following contributions to field of study.

- It demystifies AMD SEV's TLB management mechanisms, which have never been studied and reported in-depth, and identifies a severe flaw of its design of TLB isolation that leads to misuse of TLBs under the assumption of a malicious or compromised hypervisor.
- It presents a novel category of attacks against SEV, namely TLB Poisoning Attacks, which manipulate the TLB entries shared by two processes within the same SEV VM and breach the integrity and confidentiality of one of the processes. To the best of our knowledge, it is the first TLB poisoning attack demonstrated in any context.
- It demonstrates two end-to-end TLB Poisoning Attacks against SEV-ES-protected VMs. In one attack, it shows the feasibility of poisoning TLB entries to change the code execution of the victim process; in the other, it provides an example of stealing secret data from the victim process by a process (not controlled by the adversary) through shared TLB entries.

2 BACKGROUND

In this section, we present some background information about SEV's memory and TLB isolation.

Secure Encrypted Virtualization (SEV). As AMD's new memory encryption feature for AMD-V [5], SEV aims to produce a confidential VM environment in the public cloud and protect VMs from the privileged but untrustworthy cloud host (*e.g.*, the hypervisor). SEV is built atop an on-chip encryption system composed of an ARM Cortex-A5 co-processor [17] and AES encryption engines. The co-processor, also known as AMD-SP, stores and maintains a set of VM encryption keys (K_{vek}) which is uniquely assigned to each SEV-enabled VM. The K_{vek} in the co-processor could not be accessed by either the privileged hypervisor or the guest VM itself. The AES encryption engine automatically encrypts all data in the memory, and decrypts them in the CPU by using the correct K_{vek}.

Nested Page Tables. AMD adopts two-level of page tables to help the hypervisor manage the SEV VM's memory mapping. The upper-level page table, also called the guest page table (gPT), is part of the guest VM's encrypted memory and is maintained by the guest VM, and is usually a 4-level page table that translates the guest virtual address (gVA) to the guest physical address (gPA). Moreover, Guest Page Fault (gPF) caused by the gPT walk is trapped and handled by the guest VM. The lower-level page table is also called NPT or host page table (hPT), which translates gPA to system physical address (sPA), and is maintained by the hypervisor. The NPT structure gives the SEV VM the ability to configure the memory pages' encryption states. By changing the C-bit (Bit 47 in the page table entry) to be 1 or 0, the states of the guest VM's memory page can either be private (encrypted with his K_{vek}) or shared (encrypted with the hypervisor's K_{vek}). The gPT and all instruction pages are forced to be private states no matter of the value of C-bit.

Moreover, Nested Page Faults (NPF) may be triggered by the hardware during the NPT walk. According to the NPF event, the hypervisor can grab useful information that could reflect the behavior of a program, and therefore leak sensitive information, including the gPA of the NPT and the NPF error code [2]. This forms a well-known controlled-channel attack [12, 22, 30], which compromises SEV's confidentiality and integrity.

Address Space Layout Randomization (ASLR). ASLR is a widely used spectrum protection technique that randomizes the virtual memory areas of a process to defend against memory corruption attacks. This defense mechanism prevents attackers from directly learning the pointer's virtual address and forces them to rely on software vulnerabilities or side-channel attacks [6, 13, 14, 18] to locate the randomized virtual address. Different operating systems have different ASLR implementations. For example, a 64-bit Linux system usually exhibits 28-bit of ASLR entropy for executable [11] while Windows 10 exhibits only 17-19 bits of ASLR entropy for executables [31].

Translation Lookaside Buffer (TLB) and Address Space Identifier (ASID). TLB is a caching hardware inside the chip's memory-management unit (MMU). After a successful page table walk, the mapping from the virtual address to the system address is cached in TLB. For a nested page table on SEV, the mapping of the gVA and the sPA is cached in the TLB. During a page table walk, given a guest CR3 (gCR3) and a host CR3 (hCR3), the hardware automatically translate a gVA to a sPA using the two-level page tables despite the gPT and the NPT are encrypted by different K_{vek}s. AMD-SP uses ASID to uniquely identify the SEV-enabled VM and its K_{vek}. ASID is also part of the tag for both cache lines and TLB entries [17].

3 UNDERSTANDING AND DEMYSTIFYING SEV'S TLB ISOLATION MECHANISMS

In this section, we briefly sketch our understanding of TLB isolation mechanisms used in AMD Virtualization for both non-SEV VMs and SEV-enabled VMs. For some of the mechanisms that are not documented, we experimentally validated our conjectures.

3.1 TLB Management for Non-SEV VMs

To avoid frequent TLB flushes during VM world switches, AMD introduced ASID in TLB entries [1]. ASID 0 is reserved for the hypervisor and the rest of the ASID are used by the VM. The range of the ASID pool can be determined by CPUID 0x8000000a[EBX]. TLB is tagged with the ASIDs of each VM and the hypervisor, which avoids flushing the entire TLB at the world switch and also prevents misuses of the TLB entries belonging to other entities.

We explore the TLB management algorithm for non-SEV VMs by diving into the source code of AMD SVM [4]. Specifically, the hypervisor is responsible for maintaining the uniqueness and the freshness of the ASID in each logical core of the machine. For each logical core, the hypervisor stores the most recently used ASID in the `svm_cpu_data` data structure. Before each VMRUN of a vCPU of a non-SEV VM, the hypervisor checks whether the CPU affinity of the vCPU has changed by comparing the ASID stored in its VMCB with the most recently used ASID of this logical core. If a mismatch is observed, which means either the vCPU was not running on this logical core before the current VMEXIT or more than one vCPUs sharing the same logical core concurrently, the hypervisor assigns an incremental and unused ASID to this vCPU. In either of these cases, the increment of the ASID ensures the residual TLB entries cannot be reused. Otherwise, no TLB flushing is needed and the vCPU can keep its ASID and reuse its TLB entries after VMRUN.

The hypervisor is in charge of enforcing TLB flushes under certain conditions. For example, when the recently used ASID exceeds the max ASID range on the logical core, a complete TLB flush for all ASIDs is required. To flush TLBs, the hypervisor sets the `TLB_CONTROL` bits in `TLB_CONTROL` filed (058h) of the VMCB during VMEXITs. With different values of bits 39:32 of `TLB_CONTROL`, the hardware will perform the different operation on the TLB:

- `TLB_CONTROL_DO_NOTHING` (00h). The hardware does nothing.
- `TLB_CONTROL_FLUSH_ALL_ASID` (01h). The hardware flushes the entire TLB.
- `TLB_CONTROL_FLUSH_ASID (03h)`. The hardware flushes all TLB entries whose ASID is equal to the ASID in the VMCB.
- `TLB_CONTROL_FLUSH_ASID_LOCAL` (07h). The hardware flushes this guest VM's non-global TLB entries.
- Other values. All other values are reserved, so other values may cause problems when resuming guest VMs.

After each VMRUN, hardware checks these bits and performs the corresponding actions. The hypervisor is in charge of informing the hardware to flushes TLBs and maintain TLB isolation. Hardware may also automatically perform a partial TLB flush without triggering a special VMEXIT when observing context switches or MOV-to-CR3 instructions. In such cases, only the TLB entries tagged with the current ASID (either in guest ASID or the hypervisor ASID) are flushed [2].

3.2 Demystifying SEV's TLB management

The TLB management for SEV VMs and non-SEV VMs is slightly different. The ASIDs of SEV VMs remain the same in their lifetime. Therefore, instead of dynamically assigning an ASID to a vCPU, all vCPUs of the same SEV VM have the same ASID. At runtime, TLB flush is still controlled by the hypervisor. Especially, KVM records the last resident CPU core of each vCPU. For each CPU logical core, it also records the VMCB of the last running vCPU ($sev\_vmcbs[asid]$) for each ASID. Before the hypervisor resumes a vCPU via VMRUN, it sets the TLB control field in the VMCB to the value of TLB_CONTROL_FLUSH_ASID when (1) this vCPU was not run on this core before or (2) the last VMCB running on this core with the same ASID is not the current VMCB. This enforces the isolation between two vCPUs of the same SEV VM. The code is listed in Listing 1. However, if the hypervisor chooses not to set the TLB control field, no TLB entries will be flushed.

```
1  struct svm_cpu_data *sd = per_cpu(svm_data, cpu);
2  int asid = sev_get_asid(svm->vcpu.kvm);
3  pre_sev_es_run(svm);
4  svm->vmcb->control.asid = asid;
5 // No CPU affinity change and No VMCB change
6  if (sd->sev_vmcbs[asid] == svm->vmcb &&
7      svm->vcpu.arch.last_vmentry_cpu == cpu)
8    return;
9 //Otherwise, flush the TLB tagged with the ASID
10 sd->sev_vmcbs[asid] = svm->vmcb;
11 svm->vmcb->control.tlb_ctl = TLB_CONTROL_FLUSH_ASID;
12 vmcb_mark_dirty(svm->vmcb, VMCB_ASID);
13 }
```

Listing 1: Code snippet of `pre_sev_run()`.

Experiments to demystify TLB tags. According to AMD manual [2], ASID is part of TLB tag. But is unclear what are the remaining parts of the tag. We conducted some experiments to explore

Table 1: TLB flush rules. The *World* column indicates whether the event happens in host world or the guest world; *TLB tag* represents the TLB entry's ASID to be flushed—the host's ASID is 0 and the SEV VM's ASID is N; *Forced* indicates whether the TLB flush is forced by the hardware or controllable by the hypervisor. * highlights a special case, in which when the world switch happens between two vCPUs, the TLB tagged with 0 is forced to be flushed while the TLB tagged with N is flushed under the control of the hypervisor.

World	Events	TLB Tag	Forced
Host/Guest	MOV-to-CR3, Context-switch	0/N	✓
Host/Guest	Update Cr0.PG	0/N	✓
Host/Guest	Update CR4 (PGE, PAEm and PSE)	0/N	✓
Host	Address transliation Registers	All	✓
Host	Activate an ASID for SEV VM	N	✓
Host	Deactivate an ASID for SEV VM	N	✗
Host	ASID exceeds ASID pool range	All	✗
Host	Two vCPUs switch	0+N*	✓+✗*
Host	Change vCPU's CPU affinity	N	✗

the structure of TLB tags. Specifically, we checked whether vCPUs' TLB entries on the co-resident logical cores will influence each other and whether TLB entries from the different VM modes (non-SEV, SEV, or SEV-ES) will influence each other. The experiment settings are similar. To explore TLB isolation between co-resident logical cores, we manually set the ASID of two vCPUs to the two co-resident logical cores of the same physical core. To explore TLB isolation between VMs with different VM modes (*e.g.*, SEV and non-SEV), we configured a non-SEV VM and a SEV/SEV-ES VM on the same logical core and set the non-SEV VM's ASID to be identical to the SEV/SEV-ES VM's ASID. In both cases, we skipped the TLB flush to check whether the TLB poison is observed (using steps in Section 4.2.1). In neither of two cases, TLB poison is observed. Therefore, we conclude:

- **ASID.** ASID is part of the TLB tag, which provides TLB isolation for TLB entries with different ASID.
- **Logical Core ID.** The Logical Core ID is also part of the TLB tag, which provides TLB isolation for TLB entries on the same physical core but different logical cores.
- **VM mode.** VM mode is part of the TLB tag. Even a non-SEV VM may have the same ASID as a SEV or SEV-ES VM, however, the TLB tag field contains information about the VM's mode, which isolates TLB entries from VMs under different modes.

Besides these components, we have also conjectured that C-bits—the C-bit in the guest page table (gC-bit) and the C-bit in the nested page table (nC-bit)—are also part of the TLB tag. The reason is that when address translation bypasses the page table walk, the values of the gC-bit and nC-bit are still required for the processor to determine which ASID to present to AMD-SP if memory encryption is needed. However, there is no direct evidence for us to conclude the exact C-bit tag format in TLB entries. We have no way to empirically affirm that, for instance, whether both of the C-bits are in the TLB tag or only one C-bit is in the TLB tag.

3.3 TLB Flush Rules for SEV VMs

We summarize the TLB flush rules for SEV/SEV-ES VMs in both hardware-enforced TLB flush and the hypervisor-coordinated TLB flush in Table 1. The hardware-enforced TLB flush rules cannot be skipped, while the hypervisor-coordinated TLB flush can be skipped by a malicious hypervisor, which is the root cause of the TLB Poisoning Attack.

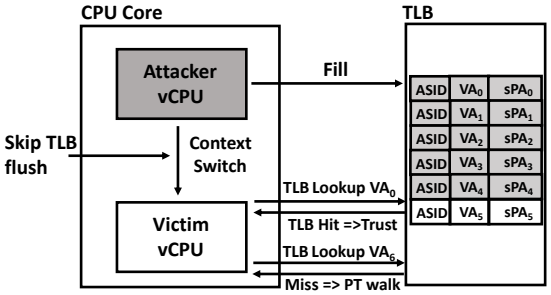


Figure 1: TLB misuses across vCPUs.

Hardware-enforced TLB flushes. All TLB entries are flushed when there is System Management Interrupt (SMI), Returning from System Management (RSM), Memory-Type Range Register (MTRR), and I/O Range Registers (IORR) modifications or MSRs access related to address translation, no matter their ASIDs. At the same time, hardware will automatically flush TLB tagged with the current ASID when observing activities like MOV-to-CR3, context switches, updates of CR0.PG, CR4.PGE, CR4.PAEm and CR4.PSE. Hardware will also force a TLB flush when the hypervisor wants to activate an ASID for a SEV VM.

Hypervisor-coordinated TLB flushes. There are mainly two cases where the hypervisor is coordinated in TLB management. (1) When different VMCB with the same ASID (different vCPUs of the same SEV VM) is to be run on the same logical core. (2) The VMCB to be run was executed on a different logical core prior to this VMRUN.

4 ATTACK PRIMITIVES

In this section, we discuss the threat models consider in this paper, and then introduce three attack primitives: TLB misuse across vCPUs (Section 4.2), TLB misuse within the same vCPU (Section 4.3), and a covert data transmission channel between the hypervisor and a process in the victim VM that is under the adversary's control (Section 4.4).

4.1 Threat Model

We consider a scenario where the platform is hosted by a hypervisor controlled by the adversary. The victim VM is a SEV-ES enabled VM and thus protected by all SEV-ES features. We assume the ASLR is enabled inside the victim VM.

There is an unprivileged attacker process controlled by the adversary running in the victim VM. The attacker process does not have access to the kernel or learn sensitive information from procfs. The attacker process does not need to have capabilities to perform network communication. We note that the assumption of having an attacker process running inside the victim VM can be weakened (see Section 6). The victim process can be any process in the victim VM other than the attacker processes. We assume the adversary can learn the virtual address range of the victim VM via other attacks, such as CrossLine attacks [21].

4.2 TLB Misuse across vCPUs

When the victim VM has more than one vCPU, the attacker process and the victim process can run on different vCPUs. We call the vCPU running the attacker process the attacker vCPU and the vCPU running the victim process the victim vCPU. The adversary can misuse TLB entries by skipping the TLB flush during the context switch of these two vCPUs. We use two examples to show how this may be exploited to breach the integrity and the confidentiality of the victim process.

4.2.1 TLB Poisoning. We first show that by poisoning TLB entries, the attacker process can alter the execution of the victim process. The attack is illustrated in Figure 1.

- **Step-I**: The victim process is suspended before executing an instruction at address VA_0. This can be achieved by manipulating PTEs to trigger NPFs. Note that the content of this instruction is not relevant to this attack.
- **Step-II**: The hypervisor schedules the attacker vCPU to the same logical core as the victim vCPU, and the TLB control field is set to TLB_CONTROL_FLUSH_ASID (03h) to flush the TLB entries with the SEV VM's ASID.
- **Step-III**: It then instructs the attacker process to run an instruction sequence "mov $0x2021, %rax; CPUID" also at address VA_0. The CPUID instruction will trigger a VMEXIT. During the VMEXIT, the attacker vCPU is paused, and the victim vCPU is scheduled to run without flushing the TLB entries.
- **Step-IV**: When the victim process executes the instruction at VA_0, a VMEXIT due to CPUID can be observed with the %rax value set to 0x2021 in the GHCB. This means the victim process has been successfully tricked to execute the same instruction as the attacker process at VA_0, because it reuses the TLB entry poisoned by the attacker process.

4.2.2 Secret Leaking. The second example shows that the attacker process can read the victim process's memory space directly.

- **Step-I**: The attacker process uses `mmap()` syscall to pre-map a data page such that the virtual address VA_0 points to a data region on this page.
- **Step-II**: The victim process is scheduled to run and accesses the memory at address VA_0, which can be either a instruction fetch or a data load. This step loads a TLB entry into the TLB.
- **Step-III**: The victim vCPU is de-scheduled by the hypervisor, and the attacker vCPU is scheduled to run on the same logical core. The hypervisor sets the TLB control field of the attacker's VMCB to TLB_CONTROL_DO_NOTHING (00h), such that no TLB entry is flushed.
- **Step-IV**: After being scheduled to run and loading data from VA_0, we observe that the attacker process successfully loads the data from the victim's address space, compromising the victim's confidentiality. This is because the TLB entries created by the victim process is reused by the attacker process.

4.3 TLB Misuse within the Same vCPU

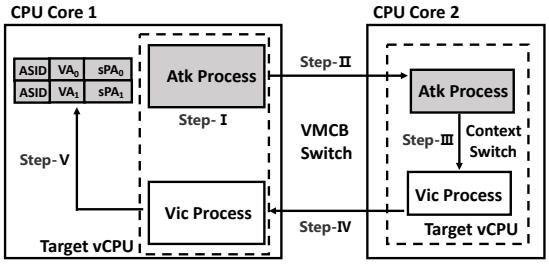
When the victim VM has only one vCPU, the attacker process shares the vCPU with the victim process. In this case, TLB misuse is less straightforward. The TLB flush rules we illustrated in Section 3.3 suggest that the hardware will automatically flush the entire TLB tagged by the victim VM vCPU's ASID when there is an internal context switch in the guest VM, which leaves no chance for the hypervisor to skip the TLB flush. As such, the hypervisor cannot directly misuse the TLB entries between two processes within the same vCPU. To address this challenge, we propose a novel VMCB-switching approach to bypass the hardware-enforced TLB flush during the internal context switch.

4.3.1 Bypassing Hardware-enforced TLB Flushes. The key to bypassing the hardware-enforced TLB flush is to reserve the attacker process's TLB entries on one CPU core and then migrate the vCPU to another CPU core. The internal context switch between the victim process and the attacker process is then performed on the second CPU core, which automatically flushes all TLB entries on the second logical core. Because the hypervisor isolates the first CPU core to prevent other processes from evicting its TLB entries, the TLB entries of the attacker processes are hence preserved. The hypervisor then migrates the vCPU back, with the victim process executing on it. The victim process will then misuse the TLB entries *poisoned* by the attacker process.

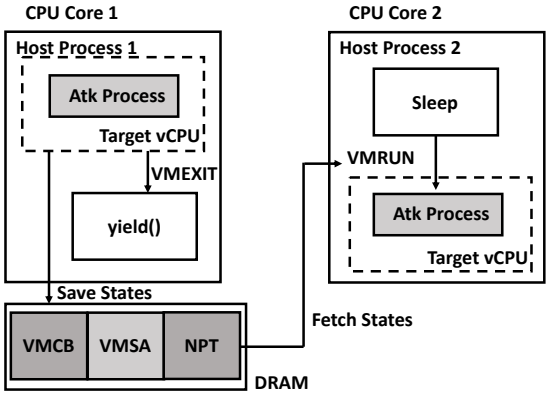
The challenges for bypassing the hardware-enforced TLB flush are two-fold: First, changing the vCPU affinity inside the victim VM leads to TLB flush for both the victim and attacker processes, which, nevertheless, can only be done by a privileged process. Secondly, changing the CPU affinity outside the victim VM—from the hypervisor side—may easily evict the reserved TLB entries. Thus, traditional CPU schedule methods like `taskset` or `sched_setaffinity` cannot work in our case.

4.3.2 VMCB Switching. The following VMCB-switching approach can be used to bypass the hardware-enforced TLB flushes (shown in Figure 2).

- **Step-I:** The hypervisor first isolates the target vCPU hosted in a hypervisor process HP_1 on logical core LC_1 and prevents other processes from accessing LC_1, as well as its co-resident logical core on the same physical core. The hypervisor also reserves another logical core LC_2 with an idle hypervisor process HP_2. This is to ensure irrelevant processes will not evict the reserved TLB entries.
- **Step-II:** After the attacker process poisons the targeted TLB entries, the hypervisor traps the vCPU into a yield() loop during one VMEXIT. Meanwhile, the hypervisor lets the idle process HP_2 on LC_2 to resume the attacker vCPU using its VMCB, VMSA pointer, and NPT structures. This is possible because all states of the attacker vCPU (*e.g.*, registers, ASID, Nested CR3) are stored in the DRAM, encrypted using either hypervisor's memory encryption key (*e.g.*, VMCB, NPT) or the guest VM's VM encryption key (*e.g.*, VMSA). After resuming the attacker vCPU on LC_2, there are no valid TLB entries on LC_2, but the attacker process inside the attacker vCPU can continue execution after page table walks.
- **Step-III:** The hypervisor traps and traces gCR3 changes to monitor the internal context switches on the attacker vCPU. Specifically, it intercepts TRAP_CR3_WRITE VMEXIT and extract the gCR3 value in the EXITINFO1 field of VMCB. Since the inner context



(a) VMCB switching steps.



(b) Step-II: Change vCPU's CPU affinity without TLB flush.

Figure 2: VMCB switching.

switch happens on LC_2, no hardware-enforced TLB flush is triggered on LC_1, and thus the attacker process's TLB entries are preserved on LC_1.

- **Step-IV:** After observing a context switch from the attacker process to the victim process is scheduled, the hypervisor switches the attacker vCPU back to LC_1 following a similar method described in Step-II. The hypervisor stops HP_2 on LC_2 and releases HP_1 on LC_1 from the empty loop.
- **Step-V:** After resuming execution on LC_1, the victim process first tries to execute its next instruction pointed by `RIP` in VMSA via a TLB lookup. The preserved TLB entries on LC_1 are unconditionally trusted by the hardware. After the victim process has used the attacker's TLB entries to execute instructions, some remaining TLB entries belonging to the attacker process may potentially disturb the execution of the victim process afterwards. Thus, the hypervisor can choose to perform a total TLB flush.

Note that the attacker process and the hypervisor can also breach the confidentiality of the victim process in a reversed way, where the hypervisor reserves the victim process's TLB entries and let the attacker process to reuse it to exfiltrate secrets from the victim's address space.

4.4 CPUID-based Covert Channel

The third primitive we build is for transmitting data between the hypervisor and the attacker process in the victim VM that is under the adversary's control. To do so, we build a CPUID-based covert channel so that network communication is not required. The adversary-controlled process may execute CPUID instructions to receive data or pass the data to the hypervisor. Specifically, to send data to the hypervisor, the attacker process may trigger a CPUID with a reserved RAX value (*e.g.*, 1234) to initiate data transfer. The attacker process then repeatedly triggers CPUID with RAX filled with the data to be transferred. Similarly, to receive data from the hypervisor, the attacker process can trigger a CPUID with another reserved RAX value (*e.g.*, 1235). The hypervisor retrieves the value of RAX and passes the data into GHCB's RAX field before VMRUN. The attacker process can then read the value of RAX after the CPUID instruction. Data received from the covert channel can use used as commands; the attacker process performs pre-defined actions (*e.g.*, mmap memory page and read certain virtual address) in accordance with the command received. On our testbed, the maximum transmission speed is 1.854MB/s when using the 8-byte RAX register for data transmission. Other covert channels that make use of cache timing [24, 25] or AMD's way predictor [23] can also be adopted as covert channels, but are less robust.

5 TLB POISONING WITH ASSISTING PROCESSES

In this section, we introduce the first variant of TLB Poisoning attacks, which is assisted by an unprivileged attacker process running in the victim VM. Following the threat model described in Section 4, we assume the attacker process is unprivileged with limited access to system resources, such as procfs, networking, or any privileged system capabilities. This is practical either when the adversary has an unprivileged user account on the victim VM or an application with security vulnerabilities remotely exploitable by the adversary. To simplify the attack, we assume the ASLR is disabled on the victim VM or the attacker process can learn the virtual memory area (VMA) of the victim process. In a real attack, the attacker process can break the ASLR either by CROSSLINE attack or other existing methods [6, 13, 21].

5.1 Case Study: OpenSSH

In this case study, we show that with the help of an unprivileged attacker process within a guest VM, the adversary can poison the TLB entries of a privileged victim process and then control its execution. The attack is applied to OpenSSH and used to bypass password authentication.

5.1.1 OpenSSH's Process Management. The sshd daemon process (denoted P_d) is launched during system boot. The daemon process runs in the background and listens to connections on SSH ports (*i.e.*, 22). Its address space is defined in the kernel by the VMA data structures. Upon receiving a connection, P_d forks a sshd child process P_c, which performs a privilege separation (or *privsep*) by spawning another unprivileged process P_n to deal with the network transmission and keeps the root privilege itself to act as a monitoring process. Once the user has successfully authenticated, P_n is terminated, and a new process P_u is created under the new user's username. In our TLB Poisoning Attack, the victim process is the privileged child sshd process P_c and the attacker process aims to poison the TLB entries of P_c.

5.1.2 Password Authentication Bypass. The adversary first initializes a SSH connection to the target VM and monitors gCR3 changes by setting the CR3_WRITE_TRAP intercept bit in its VMCB. When the SSH packet from the adversary is received by the SEV-ES VM, the adversary will immediately observe a context switch (*i.e.*, gCR3 change). The new process to run is the sshd child process P_c. In this way, the adversary can identify the gCR3 of P_c.

① **Locate the shared library.** The attacker process first helps the adversary to locate the gPA of the shared library. In our attack, we target at `pam_authenticate()`, which is a function of the shared library `libpam.so.0` and used by sshd for password authentication. `pam_authenticate()` returns 0 if the authentication succeeds. The adversary can use the attacker process to help locate the gPA of `pam_authenticate()` (denoted gPA_{pam}). He first synchronizes with the two colluding entities using the covert channel described in Section 4.4 and then calls `pam_authenticate()` from the attacker process. The hypervisor can learn gPA_{pam} by triggering NPFs.

② **Track the victim's execution.** The adversary clears the Present bit of all pages and monitors NPFs after intercepting his SSH packet with the incorrect password. If a NPF of gPA_{pam} is observed, the adversary knows the victim process is going to authenticate the password by calling `pam_authenticate()`. The adversary then pauses the victim process by trapping the victim in the gPA_{pam} NPF handler. This is used to provide a time window for the attacker process to poison the TLB entries. Note that this step is rather important in real attacks. The attacker process needs to poison the TLB entries right before the victim process accessing those poisoned TLB entries. Otherwise, the poisoned TLB entries may be evicted by other activities.

③ **Poison TLB entries.** The adversary can then poison the TLB entries of the victim. Let the virtual address of the instruction page containing `pam_authenticate()` in P_c be gVA_{pam}. We assume the adversary can learn gVA_{pam} in advance. gVA_{pam} is predictable if ASLR is disabled. The adversary can also learn gVA_{pam} using existing attack methods [6, 13, 21]. The adversary targets at poisoning the TLB entries indexed by gVA_{pam}. Specifically, the attacker process first mmap a page with the virtual address to be gVA_{pam}. Note that gVA_{pam} is only used in P_c and the attacker process can assign this virtual address to a new instruction page. The attacker process then copies the same instruction page as the victim into the new page, but replaces a few instructions of `pam_authenticate` (offset 0x5b0 - 0x65f of the binary, starting with `test %rdi %rdi`) with `mov $0 %eax` and `ret` (0xb8 0x00 0x00 0x00 0x00 0xc3). The adversary also schedules the attacker process to the same logical core as the victim process by changing the CPU affinity of the vCPU. The attacker process then repeatedly accesses this instruction page in a loop to preserved the TLB entries.

④ **Bypass authentication.** After the attacker process poisons the TLB entries of `pam_authenticate()`, the adversary directly resumes P_c without a TLB flush. Recall in step ②, P_c was paused before a page table walk to resolve gPA_{pam}. The adversary resumes P_c without handling this page table walk in order to force P_c to reuse the poisoned TLB entries. In this way, when P_c calls `pam_authenticate()`, it will execute the instruction in the attacker's address space. Therefore, the function will directly return with an 0 in `EAX` and thus allow arbitrary user to login.

5.2 Evaluation

The experiment settings are list below. The CPU we used is AMD EPYC 7251 with 8 physical cores. All the software needed to launch a SEV-ES VM is download from AMD SEV repository [4]. The host kernel version is sev-es-v3 . The QEMU version used was sev-es-v12 and the OVMF version was sev-es-v27 . The victim VM was a SEV-ES-enabled VMs with 4 vCPUs, 4 GB DRAM and 30 GB disk storage. The OpenSSH version is OpenSSH_7.6p1 and the OpenSSL version is 1.0.2n. We repeated the attack 20 times and evaluated the attacks in terms of successful rate: All the 20 attacks could successfully bypass the password authentication and logged in with incorrect passwords.

6 TLB POISONING WITHOUT ASSISTING PROCESSES

In this section, we show that TLB Poisoning attacks can work even without the help of an attacker process in the victim VM. The intuition is that when processes share similar virtual address spaces, TLB misuse may happen between these processes without direct control of either of them.

Specifically, we target at `fork()`, which is a system call used to create new processes. `fork()` is widely used in server-side applications, *e.g.*, OpenSSH, sftp, Nginx, and Apache web server, to serve requests from different clients. The forked child processes has a high probability to share a very similar virtual memory area with majority of their virtual address space layout overlapped. Even the VM's administrator chooses to enable ASLR, the same VMA randomization will be applied to the parent process and all child processes, which gives the adversary the chance to conduct TLB poisoning without concerning about the unpredictable VMA. This similarity of address spaces of forked processes has been exploited in memory hijacking attacks [19].

Attack scenarios. Similar to the previous case study, we choose to showcase our TLB Poisoning attack against an SSH server. But this time, we target Dropbear SSH [15], which is a lightweight open-source SSH server written in C and released frequently since 2003. We did not choose the more popular OpenSSH because it alters its memory address space in all its children processes that serve incoming connections (by calling `exec()`). However, this mechanism is only observed in OpenSSH and OpenBSD. Other network applications like Dropbear SSH and Nginx will not change their virtual memory layout for different connections.

We assume the targeted Dropbear SSH server application is free of memory safety vulnerabilities and timing channel vulnerabilities. We assume the binary of the Dropbear Server application is known by the adversary. We assume the username of a legitimate user is also known by the adversary; this is a practical assumption as usernames are not considered secrets. To simplify the attack, we also assume the two processes are scheduled on two different vCPUs, which makes the attack easier to perform; otherwise the VMCB-switching approach is required.

6.1 Poison TLB Entries between Connections

We consider two SSH connections: One is the connection from the adversary, which is served by the process P_{atk} that is forked from the DropSSH server daemon; the other is a connection from a legitimate user, which is served by the process P_{vic}. The attack goal is to allow the attacker process to temporarily use the victim process's TLB entries and circumvent the password authentication.

Regular login procedures. After the login password packet is received by the victim VM, P_{vic} calls `svr_auth_password()` to validate the password. As shown in Listing 2, the password encryption function in the POSIX C library `crypt()` is called to generate a hash of the user-provided password. The result is stored in a buffer called `testcrypt`. The buffer storing the plaintext of the password is freed immediately. After that, the hash of the user-provided password is compared with the stored value in the system file using `constant_time_strcmp()`, which returns 0 if these two strings are identical. If the user-provided password is correct, P_{vic} will take the correct-password branch, which calls `send_msg_userauth_success()`. Otherwise, the incorrect-password branch is taken.

```
void svr_auth_password(int valid_user) {
   char * passwdcrypt = NULL;
   // store the crypt from /etc/passwd
   char * testcrypt = NULL;
   // store the crypt generated from the password sent
   ...
   // **Execution Point 1 (NPF)
   if (constant_time_strcmp(testcrypt, passwdcrypt) == 0)
      {
   // successful authentication
   // **Execution Point 2 (NPF)
       send_msg_userauth_success();
      } ...
}
```

Listing 2: Code snippet of `svr_auth_password()`.

Attack overview. We show that by breaking the TLB isolation, the attacker process P_{atk} can bypass the password authentication even with an incorrect password. Specifically, the virtual addresses of the `testcrypt` buffer are usually the same for both P_{atk} and P_{vic} (this fact will be empirically evaluated later). We use <gVA$_{pwd}$, sPA$_{vic}$> to denote the TLB entry owned by P_{atk}, which caches the mapping from the virtual address of the `testcrypt` buffer to the system physical address that stores the hashed password used in P_{vic}. The goal here is to make sure the TLB entry <gVA$_{pwd}$, sPA$_{vic}$> is not flushed when P_{atk} executes `constant_time_strcmp()`. In this way, P_{atk} can re-use the `testcrypt` of P_{vic} to circumvent password authentication.

Key challenges. The key challenge in this attack is to ensure only necessary TLB entries are preserved. Otherwise, later TLB entries may flush those necessary TLB entries. To address the challenge, it is important to perform TLB poisoning at the proper execution point. As shown in Figure 3, the adversary needs to locate the execution points right before and after the password authentication (*e.g.*, `constant_time_strcmp()`), which can be done using the NPF controlled channels.

The attack overview is shown in Figure 3. Let the guest physical address of the instruction page where the `svr_auth_password()` and the `constant_time_strcmp()` are located be gPA$_1$ and gPA$_2$, respectively. The adversary first traps the attacker process in an empty loop when handling the NPF of gPA$_2$ (execution point 1), which means P_{atk} is about to call `constant_time_strcmp()`. Then the adversary will not interrupt P_{vic} until it also reaches the NPF of gPA$_2$ (execution point 1). When handling this NPF, the adversary triggers a complete TLB flush. P_{vic} then continues execution until it finishes the password authentication and tries to return to `svr_auth_password()`. A NPF of gPA$_1$ (execution point 2) will be observed and the adversary traps P_{vic}. Meanwhile, the adversary releases the attacker process and skips the TLB flush. All TLB entries used by P_{vic} during the execution of `constant_time_strcmp()` are thus preserved in the TLB, including TLB (gVA$_{pwd}$, sPA$_{vic}$). After the attacker process completes `constant_time_strcmp()`, passes the password check, and reaches execution point 2, the adversary triggers a complete TLB flush (to avoid unnecessary TLB misuses) and releases P_{vic}. Both P_{atk} and P_{vic} continue execution as normal afterwards and no traces will be left in the kernel message.

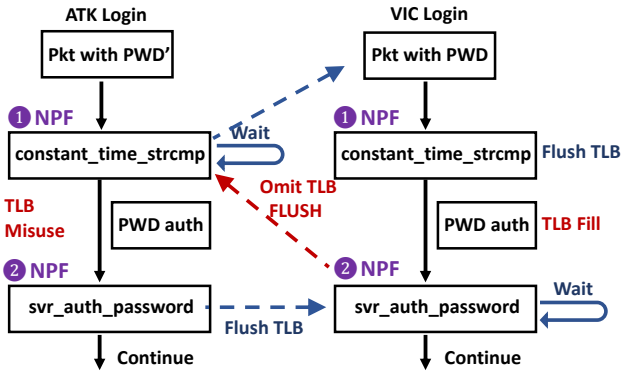


Figure 3: Attack steps to bypass password authentication.

6.2 An End-to-end Attack

The adversary follows these steps for an end-to-end attack:

① **Monitor network traffic.** Even the adversary cannot directly learn the content of encrypted network packets, the adversary can inspect incoming and outgoing network packets through the unencrypted metadata (*e.g.*, destination address, source address or the port number). The adversary continuously monitors network traffic to identify the SSH handshake procedure. Once the adversary identifies a *client_hello* packet sent from a legitimate user, the adversary traps that packet and sends a *client_hello* packet from a remote machine controlled by himself. Once this *client_hello* packet reaches the victim VM, the adversary resumes the processing of the *client_hello* packet from the legitimate user. Thus, the victim VM shall receive two connection requests, one from the adversary and another from a legitimate user.

② **Monitor fork() and gCR3 changes.** Next, the adversary locates the gCR3 of the forked child processes. During the victim VM's booting period, the adversary continuously monitors gCR3 changes by setting the CR3_WRITE_TRAP intercept bit in the VMCB. Afterwards, gCR3 changes will cause an automatic VMEXIT with the

new gCR3 value stored in VMEXIT EXITINFO. After receiving the two SSH connection packets, the Dropbear Daemon will fork twice to generate the child process for the adversary's connection and the legitimate user's connection. We call the forked child process for the adversary's connection P_{atk}, whose gCR3 is gCR3_{atk}. We call the forked child process for the legitimate user's connection P_{vic}, whose gCR3 is gCR3_{vic}. The adversary can identify the gCR3_{atk} and gCR3_{vic} by correlating them with the received *client_hello* packets.

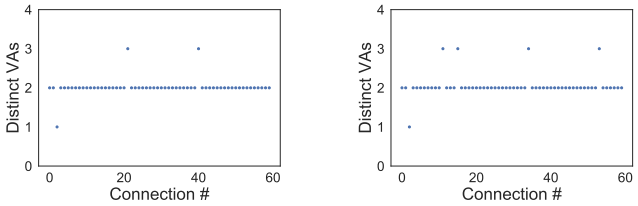
③ **Monitor NPFs to locate the target gPAs.** The adversary may try to log in by sending an arbitrary password. The legitimate user logs in by sending a correct password. The adversary triggers NPFs by clearing the Present bits in the NPT, when the encrypted SSH packets that contain the passwords are observed. A sequence of NPF for P_{atk} and a sequence of NPFs for P_{vic} will be observed. The adversary also collects additional information (*e.g.*, NPF EXITINFO2) along with the NPF VMEXITs, which reveals valuable information. For instance, the adversary can learn that the NPF is caused by write/read access, user/kernel access, code read, or page table walks. The adversary also periodically (*e.g.*, every 50 NPFs) clears all Present bits to fine tune the NPF sequence. Since the Dropbear's binary is known by the adversary, the adversary can learn the NPF patterns offline to locate the gPA of `svr_auth_password()` (denoted gPA_1) and the gPA of the first instruction in `constant_time_strcmp()` (denoted gPA_2). The features used in pattern recognition are the sequence of NPFs and their error code. During the attack, the adversary can use the recognized pattern to locate gPA_1 and gPA_2.

④ **Skip TLB flush.** The adversary continuously monitors P_{atk} and P_{vic}. When the adversary observes the NPF of gPA_2 in P_{atk}, he traps P_{atk} in an empty loop and clears the Present bit of all pages. When the adversary observes the NPF of gPA_2 in P_{vic}, he clears the Present bit for all memory pages and performs a complete TLB flush. The adversary traps P_{vic} when he observes the NPF of gPA_1. P_{atk} is then resumed and the adversary skips the TLB flush. P_{atk} will use the preserved TLB entries from P_{vic} to to read the password hash from the `testcrypto` in the address space of P_{vic}, which leads to a successful login with an incorrect password. To void further TLB pollution, the adversary then forces a complete TLB flush and resumes the victim process. Both P_{atk} and P_{vic} will continue their execution normally afterwards.

6.3 Evaluation.

All experiments were performed on a workstation whose CPU is AMD EPYC 7251 Processor (8 physical core with SMT enabled). The VMs (including victim VM and the training VMs) used in this section were SEV-ES-enabled VMs with four vCPUs, 4 GB DRAM, and 30 GB disk storage. The software of the OS, QEMU, and the UEFI image are the same in Section 5.2. ASLR is enabled in the SEV-ES-enabled VMs by setting the parameter in /proc/sys/kernel/randomize_va_space to 2. The source code of Dropbear is downloaded from Github [15][1]. The Dropbear SSH Server is configured as the default setting. The Dropbear SSH Server is bond to Port 22. One minor non-default setting to assist the attack is that we forced P_{atk} and P_{vic} to execute on different vCPUs of the victim VM. Note, this setting improves the success rate of the attack but is not necessary in practical attacks.

[1] commit:846d38fe4319c517683ac3df1796b3bc0180be14



(a) 10 connections/s. **(b) 1 connection/s.**

Figure 4: Variation of the virtual address of `testcrypto`.

Buffer address variation. We first evaluated the variation of the virtual address of `testcrypto` under different connection ratios. In the training VM, the Dropbear server is modified to print the virtual address of `testcrypto` to the console after each connection. Then we used a simple script to initiate new SSH connections, send the correct password to login, obtain the virtual address of `testcrypto`, and end the current SSH connection. In total, 120 connections were collected. For the first 60 connections, the time interval between two contiguous connections was set to 0.1 second. For the second 60 connections, the time interval was set to 1 second. As shown in Figure 4a, when the time interval is set to 0.1 second, although 3 different virtual addresses of the `testcrypto` are observed, the virtual address of `testcrypto` remains the same in 57 out of the total 60 connections. When the time interval is set to 1 second, the virtual address of `testcrypto` remains the same in 55 out of the total 60 connections. The experiment results show that the virtual addresses for `testcrypto` are relatively stable for different connections, which gives the adversary the chance to poison the TLB entries of the `testcrypto` buffer between two connections.

Pattern matching. We evaluated the performance of pattern matching. Specifically, we repeated the above attack steps 100 times and performed pattern matching on-the-fly each time. In 98 out of the 100 trials, the adversary is able to correctly recognize the pattern and locate the gPA. The average time used to locate the pattern is 0.10137 second with a standard deviation of 0.02460 second.

End-to-end attacks. We then evaluated the success rate of end-to-end attacks. The adversary conducted end-to-end attacks in the victim VM. An incorrect password is used by the adversary for his SSH connections. The adversary repeated the attacks 20 times. In 17 out of the 20 connections, the adversary is able to log in with the incorrect password. There are two reasons that might count for the 3 failed cases. The first reason is that the reserved TLB entries might be evicted before use. The second reason is that there are false positives in pattern matching. However, the adversary can always repeat the attacks the next time a legitimate user logs in.

7 DISCUSSION AND COUNTERMEASURE

In this section, we discuss applications of TLB Poisoning Attacks on SEV-SNP, their differences compared to known attacks, and their countermeasures.

7.1 TLB Poisoning on SEV-SNP

Although we have not tested TLB Poisoning Attacks on SEV-SNP processors, according to the feedback from the AMD team, SEV-SNP has fixed the TLB misuse problem. The latest AMD architecture programmer's manual [2] also shows some newly added fields in the VMSA: TLB_ID (offset 3d0h) and PCPU_ID (offset 3d8h). However, from the public documents, it is unclear how exactly these two fields enforce additional TLB flushes. We conjecture that the hardware use TLB_ID and PCPU_ID as parts of TLB tags to identify vCPU and TLB entry's ownership. We inspected the source code of software supports of SNP (branch: sev-snp-devel)[2] [4], and failed to locate any software function that controls these two VMCB fields. Therefore, we conjecture these two fields are managed solely by the hardware. The hypervisor can still use TLB_CONTROL field to enforce TLB flushes but has lost the capability to deliberately skipping TLB flushes.

7.2 Comparison with Known Attacks

Previous works break the confidentiality and/or the integrity of SEV by replacing unprotected I/O traffic [22], manipulating NPT mapping [27, 28] and unauthenticated encryption [7, 9, 32]. All of these previous works can be mitigated by SEV-SNP via the Reversed Map table (RMP), which establishes a unique mapping between each system physical address with either a guest physical address or a hypervisor physical address. The RMP also records the ownership of each system physical address (*e.g.*, a hypervisor page, a hardware page, or a SEV-SNP VM's page) as well as the ASID. For SEV-SNP VM, the RMP checks the correctness and the ownership after a nested page table walk. Only if the ownership is correct, will the mapping between the guest virtual address and the system physical address be cached in the TLB. This ownership check prevents the hypervisor from remapping the guest physical address to another system physical address and thus prevents attacks that require manipulation of the NPT. Meanwhile, the RMP restricts the hypervisor's ability to write to the guest VM's memory page, which mitigates attacks relying on unauthenticated encryption and unprotected I/O operations.

In contrast, this work is the first to demystify how TLB isolation is performed in SEV and the first to demonstrate the security risks caused by the hypervisor-controlled TLB flushes. TLB Poisoning Attacks by themselves do not rely on the known vulnerabilities of SEV and SEV-ES, such as the lack of authenticated memory encryption, the lack of NPT protection, and the lack of I/O protection, and RMP alone does not prevent TLB Poisoning Attacks.

7.3 Countermeasures

TLB Poisoning Attacks affect all SEV and SEV-ES servers, including all first and second generation EPYC server CPUs (*i.e.*, Zen 1 and Zen 2 architecture). Older processors may use a microcode patch to enforce a TLB flush during VMRUN for all SEV/SEV-ES vCPUs. From the software side, to mitigate TLB Poisoning Attacks, we recommend all network-related applications (*e.g.*, HTTPS, FTP, and SSH server) to use `exec()` to ensure a completely new address space for a new connection.

[2]Commit: 0965d085cd2453a3512c98924dac70e5cdf17402.

8 RELATED WORK

There have been several reported design flaws of AMD SEV since its debut in 2016, including unencrypted VMCB [29, 30], unprotected I/O interface [22], unprotected memory mapping [12, 27, 28], unauthenticated memory encryption [7, 9, 32], and most recently unauthenticated ASID [21].

Unencrypted VMCB. The unencrypted VMCB vulnerability only applies to SEV and is the key reason for AMD's release of SEV-ES. The VM's states (*e.g.*, registers) are saved in plaintext during a traditional world switch in AMD hardware-based Virtualization (AMD-V) [5] under the assumption that the hypervisor is trusted. However, with SEV, unencrypted VMCB leads to numerous attacks (*e.g.*, [30]). AMD released SEV-ES in February 2017.

Lack of memory integrity. Most of the rest attacks can work on SEV-ES. Among those attacks, Li *et al.* [22] studied unencrypted I/O operations on SEV and SEV-ES. On SEV, peripheral devices (*e.g.*, disk, the network interface card) are not supported to directly read-/write guest VMs' memory with the corresponding K_{vek}. Thus, an additional buffer area is reserved and maintained by the guest VM, which provides an interface for the hypervisor to generate encryption/decryption oracles during I/O transmission. Hetzelt *et al.* [12] first studied memory mapping problems caused by hypervisor-controlled nested page tables on SEV. These types of attacks are further explored by others [27, 28].

Lack of memory confidentiality. SEV (including SEV-ES and SEV-SNP) leaves the read access ability to the hypervisor for the performance concern, which, on the other hand, gives attackers the chance to steal secrets by monitoring ciphertext changes. Li *et al.* [20] studied an unexplored ciphertext side channel against all SEV, SEV-ES, and SEV-SNP. Attackers can intercept ciphertext changes inside the VMSA area and infer VM's internal register states. The authors then presented CIPHERLEAKs attack and showed that attackers can steal RSA's private key and ECDSA signature's nonce in the latest cryptography library by monitoring registers' ciphertext changes. CIPHERLEAKs attack is believed to be the first attack against SEV-SNP.

Unauthenticated ASID. Crossline attacks [21] studied the ASID misuse and the "Security-by-Crash" principle of AMD SEV and SEV-ES. ASID is used as tags in TLB entries and cache lines, and also the identifier of memory encryption keys in AMD-SP. However, the hypervisor is in charge of the ASID management. AMD relies on a "Security-by-Crash" principle to prevent ASID misuses; it is expected that an incorrect ASID will crash the VM immediately. However, the authors showed that by assigning the ASID of a victim VM to a helper VM, the adversary could extract the victim VM's arbitrary memory block with the PTE format. Crossline attacks are stealthy, but NPT page remapping is still required.

Page-fault side channels. Page-fault side channels are widely used in many prior SEV attacks [12, 21, 22, 27, 28, 30]. The guest VM maintains its own guest page table, which transfers guest virtual address to guest physical address and is encrypted and protected by SEV [1]. The lower nested page table is transparent to and maintained by an untrusted hypervisor. The hypervisor can easily track the victim VM's execution paths by clearing the Present bit in the

lower NPT pages. Moreover, NPFs also reveal valuable information to the hypervisor (*e.g.*, write/read access and user/privileged access). That information can be actively gathered by the hypervisor and used to locate both the time point and the physical address of some sensitive data. The controlled-channel methods in theory should still work on SEV-SNP.

9 CONCLUSION

In this paper, we present the first work to demystify AMD SEV's insecure TLB management mechanisms and demonstrate end-to-end TLB Poisoning Attacks that exploit the underlying design flaws. Our study not only presents another vulnerability in the design of SEV, but reveals the difficulty of securely isolating TLBs with untrusted privileged software.

REFERENCES

[1] AMD. 2008. AMD-V Nested Paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[2] AMD. 2019. AMD64 architecture programmer's manual volume 2: System programming.

[3] AMD. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. *White paper* (2020).

[4] AMD. 2020. AMDSEV/SEV-ES Branch. https://github.com/AMDESE/AMDSEV/tree/sev-es.

[5] AMD. 2021. AMD Virtualization (AMD-V). https://www.amd.com/en/technologies/virtualization-solutions.

[6] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R Gross. 2015. $CAIN$: Silently Breaking $ASLR$ in the Cloud. In *9th $USENIX$ Workshop on Offensive Technologies*.

[7] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. 2017. Fault Attacks on Encrypted General Purpose Compute Platforms. In *7th ACM on Conference on Data and Application Security and Privacy*. ACM.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[9] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. 2017. Secure Encrypted Virtualization is Unsecure. *arXiv preprint arXiv:1712.05090* (2017).

[10] Google. 2020. Introducing Google Cloud Confidential Computing with Confidential VMs. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms.

[11] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU.. In *NDSS*, Vol. 17. 26.

[12] Felicitas Hetzelt and Robert Buhren. 2017. Security analysis of encrypted virtual machines. In *ACM SIGPLAN Notices*. ACM.

[13] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[14] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 380–392.

[15] Matt Johnston. 2021. Dropbear SSH. https://github.com/mkj/dropbear.

[16] David Kaplan. 2017. Protecting VM register state with SEV-ES. *White paper* (2017).

[17] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[18] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *2020 IEEE European Symposium on Security and Privacy*. IEEE, 309–321.

[19] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. 2014. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 424–439.

[20] Mengyuan Li, Yinqian Zhang, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium*. 717–732.

[21] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2020. CROSSLINE: Breaking"Security-by-Crash"based Memory Isolation in AMD SEV. *arXiv preprint arXiv:2008.00146* (2020).

[22] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium*. 1257–1272.

[23] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *15th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2020)*.

[24] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 46–64.

[25] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.. In *Proceedings of the Network and Distributed System Security Symposium*, Vol. 17. 8–11.

[26] Microsoft. 2021. Azure and AMD announce landmark in confidential computing evolution. https://azure.microsoft.com/en-us/blog/azure-and-amd-enable-lift-and-shift-confidential-computing/.

[27] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting Secrets from Encrypted Virtual Machines. In *9th ACM Conference on Data and Application Security and Privacy*. ACM.

[28] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *11th European Workshop on Systems Security*. ACM.

[29] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *14th ACM Conference on Computer and Communications Security*. ACM.

[30] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *ACM Asia Conference on Computer and Communications Security*. ACM, 73–85.

[31] David Weston and Matt Miller. 2016. Windows 10 mitigation improvements. *Black Hat USA* (2016).

[32] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy*. IEEE, 1483–1496.