Technische Universität Berlin

Institut für Softwaretechnik und Theoretische Informatik
Security in Telecommunications
Fakultät IV

Bachelor's Thesis

Undermining AMD Secure Encrypted Virtualization through Cache Side-Channel Attacks

Louis Quentin Burda
Matriculation Number: 413574

01.03.2023

1. Reviewer Jean-Pierre Seifert
Security in Telecommunications
TU Berlin

2. Reviewer Florian Tschorsch
Distributed Security Infrastructures
TU Berlin

Supervisors Vincent Ulitzsch

Declaration

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin, 01.03.2023

Louis Quentin Burda

Acknowledgement

I would like to thank Vincent Ulitzsch, my supervisor, for his guidance, encouragement and support throughout this research. He allowed me to work on my own time while maintaining a tight feedback loop via instant messenger, for which I am deeply grateful. I would like to thank Mengyuan Li for taking time to consider my approach and provide valuable comments and advice. I would like to thank the chair and its members for allowing me to work in their offices and for providing the equipment necessary to conduct the research and experiments in this thesis. Finally, I would like to thank my friends and family for their support.

Abstract

AMD Secure Encrypted Virtualization (SEV) provides guarantees for confidentiality and authenticity of data owned by virtual machines running on untrusted hosts through guest memory encryption and integrity checks, enforced by a trusted platform module in the CPU. Despite SEV, an encrypted guest must share resources with the host and thereby influences the system's state. Changes in state may be observed by the host to infer valuable information about the guest's execution.

In this thesis, we examine a cache-based side-channel attack *Prime+Count* on the latest iteration of SEV technology, SEV-Secure Nested Paging (SEV-SNP). The threat model of a malicious hypervisor allows us to reconfigure the system to our needs, in particular, augment traditional timing-based *Prime+Probe* through the use of performance counters to accurately detect cache misses. To demonstate its potential in undermining guest confidentiality and address some of the unique challenges of mounting side-channel attacks against encrypted guests, we developed the attack framework *CachePC*. It enables page- and instruction-wise tracking of SEV-SNP guests while analyzing what physical addresses are being accessed.

Zusammenfassung

AMD Secure Encrypted Virtualization (SEV) ermöglicht die sichere Ausführung von virtuellen Maschinen in der Cloud, indem durch Verschlüsselung und Integritätschecks, verwaltet durch einen sicheren Koprozessor, verhindert wird, dass der Zustand des Gastsystems eingesehen oder manipuliert werden kann. Trotz der Trennung von Gast und Hypervisor durch Verschlüsselung muss ein Gast auf dieselben physikalischen Resourcen wie der Host zugreifen, welches deren Zustand beeinflusst. Veränderungen im Zustand können vom Host eingesehen werden um wertvolle Informationen über die vom Gast verarbeiteten Daten zu gewinnen.

In dieser These untersuchen wir das Potenzial von einem Cache-basierten Seitenkanal-Angriff *Prime+Count* um die Vertraulichkeit von Gastinformationen auf der aktuellsten SEV Version, Secure Nested Paging (SEV-SNP), zu untermauern. Die zusätlichen Rechte, die uns als böswilliger Cloud-Provider durch das Angreifer Modell zustehen, werden benutzt um, im Gegensatz zum herkömlichen *Prime+Probe*, Performance Counter zur Erkennung von Fehlzugriffen im Cache einzusetzen. Um die praktische Anwendbarkeit des Seitenkanal-Angriffs darzustellen, haben wir ein Kernel-Modul names *CachePC* entwickelt, dass die Ausführung von virtuellen Maschinen trotz Verschlüsselung mittels SEV-SNP durch page- und single-stepping kontrollieren, und mittels *Prime+Count* die vom Gast verwendeten physikalischen Addressen auswendig machen kann.

Contents

1 Introduction

Computation running in the cloud is transparent to the provider. A malicious hypervisor can inspect and alter guest memory without the permission or knowledge of the end-user. Although mechanisms such as memory encryption exist to alleviate this issue, proper implementation and guest provisioning are required to prevent the hypervisor from gaining access to the encryption keys. When sensitive information is involved, end-users are wary to take this risk and may even purchase computing hardware themselves, resulting in a loss for both parties.

AMD Secure Encrypted Virtualization (SEV) aims to address this issue by providing end-users integrity and confidentiality guarantees for data owned by virtualized guests on compatible AMD CPUs. This is achieved by completely encrypting guest memory and register state, and enforcing a provisioning scheme that ensures only the AMD Secure Processor (AMD-SP) located in the CPU is ever exposed to the encryption keys [1].

Crucially, AMD's threat model does not account for cache side-channel attacks [6], although they have been applied to competing secure virtualization technology, Intel SGX, with much success [4, 3]. The threat model of a malicious hypervisor with physical and privileged access enables an attacker to reconfigure the system to the requirements of the attack, allowing for more control than in unprivileged side-channel attacks, in particular, the use of performance counters to accurately detect cache misses.

This thesis investigates the potential of the cache-based side-channel attack *Prime+Count* to infer information about the data processed by SEV-SNP protected guests without access to the encryption keys or permission, thereby undermining AMD SEV. To the best of our knowledge, we are the first to investigate the cache-based side-channel attacks against AMD SEV.

Our contributions are:

- Demonstrate that performance counter enhanced side-channel attacks on SEV-enabled guests are possible, low-noise and accurate
- Confirm that single-stepping of SEV-enabled guests is possible using local Advanced Programmable Interrupt Controller (APIC) interrupt timers

- Confirm that page-wise control of SEV-enabled guest execution is possible by manipulating page-table entries
- Discover that manipulation of advertised CPU capabilities enables a downgrade attack, allowing for cache side-channel attacks on AES running in the guest
- Contribute the attack framework *CachePC* for cache-based side-channel attacks against AMD SEV, which implements the aforementioned attack primitives on the latest Linux kernel with SEV-SNP support

2 Background

This chapter provides the necessary background to understand our methodology and the attack framework *CachePC* described in section 3.3. Section 2.1 describes the purpose of the Memory Management Unit (MMU). Section 2.2 introduces modern CPU caches and section 2.3 describes how they can be attacked using a timing-based side-channel called *Prime+Probe*. Section 2.4 describes the purpose of the local Advanced Programmable Interrupt Controller (APIC) in modern CPUs. Section 2.5 describes the confidentiality and integrity gurantees made by AMD SEV and section 2.6 briefly touches on speculation in modern CPUs.

2.1 Memory Management Unit

The Memory Management Unit (MMU) is an important component of modern CPUs used to translate between virtual and physical addresses. It enables processes running on the CPU to access memory from various different physical addresses as a single contiguous block. To do this, the virtual address space is divided into pages, each a few kilobytes in size, which can be individually mapped to different physical addresses [13].

The mapping of virtual to physical addresses is stored in main memory as a page table, consisting of entries (PTE) that perform page-wise translations. Apart from the mapping between a virtual and physical page, each PTE contains auxiliary information such as permission bits and a *present* bit. The *present* bit specifies whether the physical page involved in the translation is resident in main memory. A page fault is raised when a virtual page is accessed with insufficient permissions or the corresponding physical page is not present [13].

A cache called a Translation Lookaside Buffer (TLB) is used to reduce the frequency of slow accesses to the page table stored in main memory [13].

2.2 CPU Cache

Caches are an important component in modern computer systems used to reduce data access time. This is achieved by storing (or *caching*) the results of slower data accesses in a dedicated volatile memory.

A modern CPU has caches organized into a hierarchy (levels 1 to 3). Typically, a large L3 cache is shared by all cores of the CPU, multiple smaller L2 caches are each shared by a subset of all cores, and many small L1 caches are assigned to each core for exclusive access [7].

The data inside a cache is organized into *sets* of cache *lines* of constant size. When data is accessed from main memory, bits in the physical address are used to identify the corresponding cache set. If the requested data is not already present in the cache, a cache line is chosen from the corresponding set and filled with data from the target address. The algorithm which determines which cache lines in a cache set are replaced is called the cache replacement policy [7].

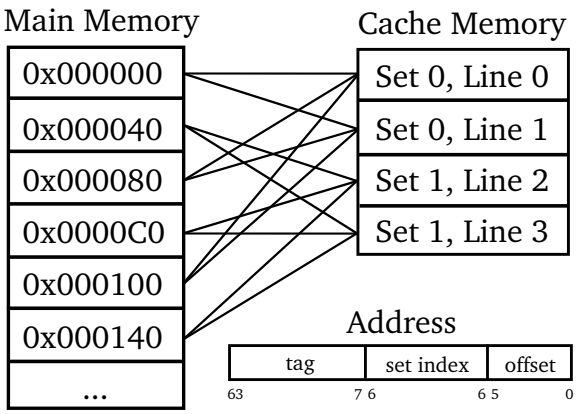

Fig. 2.1: 2-Way Set Associative Cache

In caches with more than one cache line per set, a *tag* consisting of bits from the physical address (see Fig. 2.1) is stored alongside the cached data to associate memory accesses with the correct cache line. In a Physically Indexed, Physically Tagged (PIPT) cache, the tag can only be compared once the physical address is derived from the virtual address through a TLB lookup [7].

Many modern CPUs reduce the latency of cache lookups by performing the TLB and cache line lookup in parallel by indexing the cache using the virtual address instead. These Virtually Indexed, Physically Tagged (VIPT) caches must make an educated guess which line of the cache set (cache *way*) is associated with the memory access before the TLB lookup has completed using a so-called *cache-way predictor*. Once the physical address is known, the CPU can evaluate whether the prediction was correct or the remaining lines in the set need to be considered. To perform the TLB lookup, the tag must identify the physical address with at least page-wise granularity. Without artificially increasing the tag length to account for this, the number of index and offset bits is restricted and limits the

possible cache size [7]. As a result, virtual indexing is typical in L1, but not L2 and L3 caches.

2.3 Prime+Probe

Side-channel attacks are based on the notion of inferring information without observing it through intended means. The eviction of data from a cache changes its state, which has observable side-effects: Access to data that does not reside in a cache takes more time than access to data that does. A *Prime+Probe* attack makes use of this property to analyze accesses made by instructions running on shared hardware.

In the *Prime* phase, the target cache is filled with data controlled by the attacker. Next, the target instructions are run, influencing the cache state through any accesses made (see Fig. 2.2). Finally, in the *Probe* phase, the data previously cached is accessed again line-by-line while measuring the amount of CPU cycles for the access to complete. Cache lines which take more CPU cycles to access were evicted by the target instructions [7].

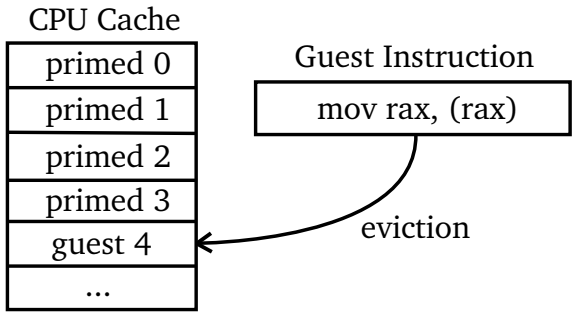

Fig. 2.2: Guest eviction in primed CPU cache

Understanding the target cache's replacement policy is critical for obtaining accurate measurements using *Prime+Probe*. If the target instructions cause a single eviction, ideally, a single cache miss occurs during *Probe* to detect it. If, however, the replacement policy decides to replace cache lines primed by the attacker before those introduced by the victim, probing all lines of a set will result in more cache misses than there were victim-induced evictions [7].

Under an LRU-like replacement policy, a reverse-ordering of cache line accesses during *Probe* ensures that the most recently used cache line is accessed first, which is least likely to have been replaced. This way, all attacker-primed cache lines are accessed before any introduced by the victim, preventing them from being replaced before being measured. However, this chain-reaction of evictions in a set caused during probe can be used to the attacker's advantage when measurements are not accurate enough to distinguish a single line evicted per set, and accuracy beyond a binary outcome (eviction or not) per set is not needed [7].

The following is a high-level example of how *Prime* and *Probe* may be implemented:

Algorithm 1 *Prime* Pseudocode

$i = 1$
while $i \leq N_{\text{cache lines}}$ **do**
    access cache line i
    $i = i + 1$
end while

Algorithm 2 *Probe* Pseudocode

$i = N_{\text{cache lines}}$
while $i \geq 1$ **do**
    $t_{\text{start}} = t_{\text{cpu}}$
    access cache line i
    $t_{\text{end}} = t_{\text{cpu}}$
    save $t_{\text{end}} - t_{\text{start}}$
    $i = i - 1$
end while

The amount of CPU cycles an access takes may be determined by reading the CPU timestamp counter (denoted t_{cpu} above) through the `rdtsc` instruction before and after each access. To ensure the memory access has completed before the second timestamp is read, an `lfence` instruction may be added before it. Modern CPUs also support the `rdtscp` instruction, which guarantees for "*all older instructions to retire before reading the timestamp counter*" [13]. Additionally, a final serializing instruction such as `cpuid` should be added to ensure no subsequent instructions begin execution before `rdtsc` completes [7].

```
# record "start" timestamp
rdtscp
mov %eax, %r8d
cpuid

# access cache line(s)
mov (%rbx), %rbx
# ...

# record "end" timestamp
rdtscp
mov %eax, %r9d
cpuid

# calculate "end" - "start"
sub %r8d, %r9d
```

Listing 2.1: Timing Measurement in GAS-Syntax Assembly

Typically, the L1 data cache is targeted in a *Prime+Probe* side-channel attack, as this cache is not shared among cores, and is separate from the corresponding L1 instruction cache. L2 and L3 caches are often of *unified* type, caching both instruction and data accesses in the same volatile memory, which can cause more noise measurements since instruction cache

misses are more difficult to control. Fig. 4.1 depicts timing results from 10000 iterations of *Prime+Probe* against an L1 cache. The x-axis corresponds to the cache set evicted. In this case, cache set 33 was accessed by the target instruction, resulting in a greater average cycle count $t_{\text{end}} - t_{\text{start}}$ compared to all other sets.

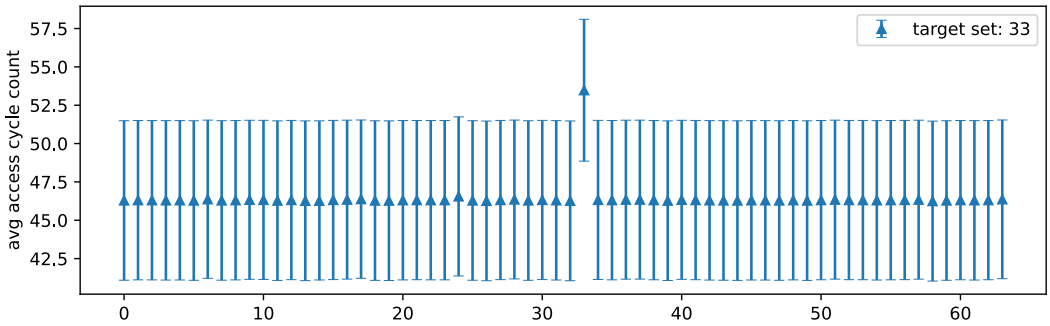

Fig. 2.3: Exemplary *Prime+Probe* timing result

The *Prime+Probe* side-channel attack is especially attractive against encryption algorithms with secret-dependent data accesses. Depending on the size of the table in relation to the size of each cache line and table entry, table lookups can leak a considerable amount of address-bits with each access. If the table index is derived from information such as the plaintext or encryption key, as is the case in (textbook) AES, this may allow an attacker to infer sensitive information even from an encrypted guest.

2.4 APIC

The Advanced Programmable Interrupt Controller (APIC) is a family of interrupt controlers designed for efficient interrupt routing on multiprocessors systems. Each core in a modern multiprocessor system includes a local APIC, used to manage external and generate local interrupts using an integrated high resolution timer.

The local APIC can be programmed to generate a local interrupt by configuring three memory-mapped registers. The desired interrupt type, such as one-shot or periodic, is encoded into an interrupt vector and stored in the *Local Vector Table Entry* register. A 32-bit value proportionate to the desired interrupt period is written to the *Initial Count* register. Once interrupts are unmasked, the 32-bit counter value decreases with a rate defined by the *Divide Configuration* register and raises an interrupt upon reaching zero [13]. As such, the APIC timer frequency, *Divide Configuration* register and *Initial Count* register together control how much time passes before the local interrupt is triggered.

2.5 AMD SEV

AMD Secure Encrypted Virtualization (SEV) provides guarantees for the confidentiality and integrity of data owned by virtual machines running on untrusted hosts. It's purpose is to transparently encrypt and decrypt guest memory accesses, and ensure that an encrypted guest can only run on genuine AMD CPUs and the host meets requirements set by the guest policy. In revisions of SEV since SEV-Encrypted State (SEV-ES), not only the guest's memory is encrypted, but also its CPU register state is loaded and stored to an encrypted memory page called the VM Save Area (VMSA) The newest revision of SEV, SEV-Secure Nested Paging (SEV-SNP), prevents modification of the encrypted VMSA through integrity checks [1, 2, 6].

The physical address of the VMSA and guest Address Space Identifier (ASID) are stored in the VM Control Block (VMCB) along with other information relevant for executing the guest. To execute the guest, the address of the VMCB is passed in the RAX register to the `vmrun` instruction. The CPU continues executing guest instructions natively until an instruction is reached which requires hypervisor emulation (Non-Automatic Exit (NAE)) or an event occurs which needs to be handled by the hypervisor (Automatic Exit (AE)).

In the case of NAE events (see Fig. 2.4), which can be caused by instructions such as `cpuid`, the guest receives a VMM Communication (VC) exception and handles it by saving information necessary for hypervisor emulation (RAX content for `cpuid`) to the Guest-Host Control Block (GHCB) and transitioning control to the hypervisor through the `vmgexit` instruction. The hypervisor reads a code denoting the exit reason from the GHCB and handles the instruction emulation accordingly. Once complete, the result is written to the GHCB and guest execution is resumed by executing `vmrun`. Finally, the guest loads the result from the GHCB, fills the appropriate registers and exits the interrupt handler [2].

Crucially, AE events, caused page faults and interrupts, do not follow this control flow. AE events are handled without guest interaction and thus the instruction pointer remains the same [2].

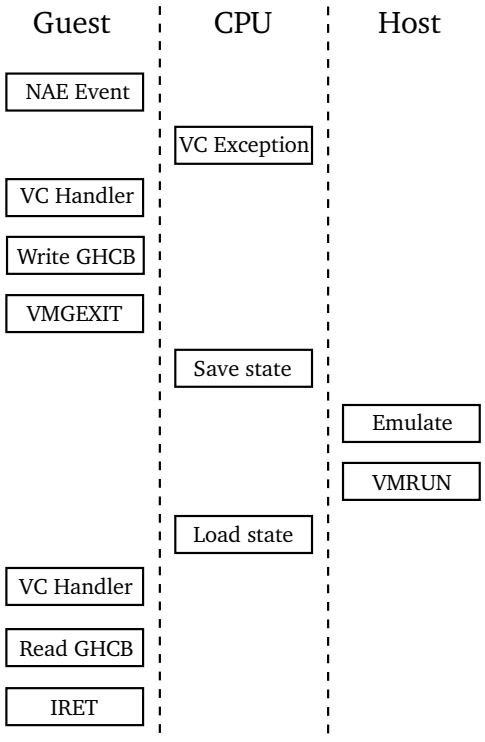

Fig. 2.4: Non-Automatic Exit Flow

Encryption of guest memory is facilitated through AMD Secure Memory Encryption (SME). For each page accessed from RAM by the CPU, an encryption bit (C-bit) in the physical address signals to an AES engine in the on-die memory controllers to encrypt or decrypt the data. The encryption scheme used is AES-Xor Encrypt Xor (AES-XEX), which for the purposes of this thesis is assumed to be secure [1].

The encryption keys are generated during provisioning and associated with the guest ASID. Guests with the same ASID thereby share encryption keys, which enables secure multiprocessing by provisioning one guest per requested virtual core. The encryption keys are managed by the AMD Secure Processor (AMD-SP), an ARM based microcontroller integrated into the CPU. It acts as an integrated trusted platform module and ensures no encryption keys are exposed to the hypervisor [1].

The communcation of the AES encryption engine with the AMD Secure Processor (AMD-SP), as well as the cryptographic isolation between guests among themselves and in respect to the hypervisor, are illustrated in Fig. 2.5.

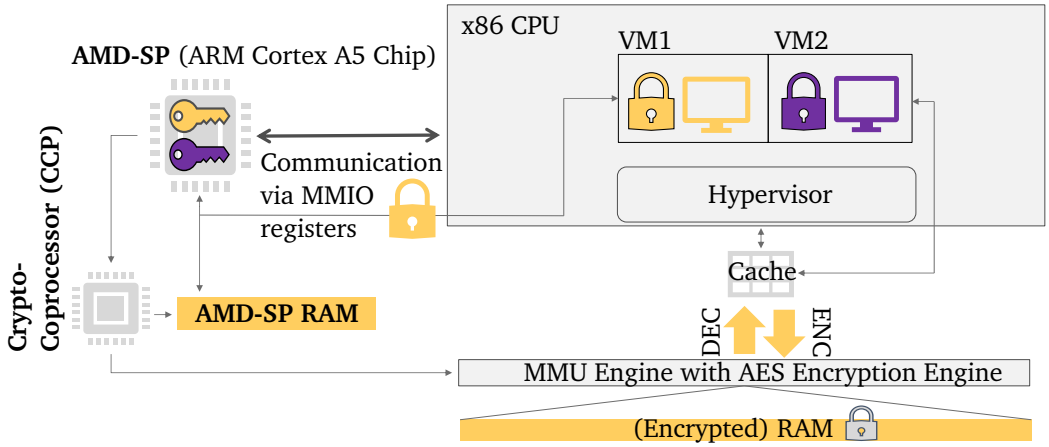

Fig. 2.5: Secure Encrypted Virtualization

To further isolate guests under SEV, various caches tag their entries with the associated ASID. This prevents decrypted memory contents leaking to the hypervisor by accessing cached results and allows avoiding TLB flushes of unrelated entries [10].

2.6 Speculation

Modern execution pipelines greatly increase their instruction throughput by predicting what instructions will be executed and what data will be accessed.

In branch prediction, the result of branching instructions is recorded and used to evaluate which branch is more likely to be taken. An accurate branch predictor can greatly improve CPU throughput, because it allows the CPU to begin processing instructions from the predicted branch before the result the branch depends on is available. Similarly, a prefetcher is used in modern CPU caches to predictively load new cache lines before they are accessed, based on the access pattern of previous instructions [7].

Such optimizations greatly increase the CPUs throughput, but may result in unexpected side-effects when trying to mount cache-based side-channel attacks.

3 Related Work

3.1 CacheSC

In the project report [7], Haller investigates the challenges of mounting *Prime+Probe* attacks on contemporary hardware. Developments, such as multi-threading, out-of-order execution, and prefetching have made it increasingly difficult to apply *Prime+Probe* to modern computer architectures.

A naive *Prime+Probe* implementation might access each cache line in order. Such a linear access pattern will trigger the cache prefetcher and ensure that a single probing access evicts more cache lines than intended, preventing an accurate measurement.

Haller found that a data structure for *Prime+Probe* was needed, which minimizes the potential side-effects of modern optimization and allows the measurement result for each cache line to be stored within it. To this end, each cache line is modelled as a struct of appropriate size, connected to previous and successive cache lines via pointers [7].

By connecting and accessing cache lines as part of a doubly-linked list, speculative execution is defeated, since the address of subsequent load instructions depends on the contents of the cache line previously accessed and must wait for this data to become available. Additionally, a strategic ordering of the cachelines withing the list allows for the stream prefetcher to be defeated.

The doubly-linked list is traversed in reverse order in the probing phase to minimize the evictions caused. This assumes a LRU-like cache replacement policy.

In the process of their research, Haller developed the library CacheSC [17] for mounting *Prime+Probe* attacks on modern hardware. This implementation was later adapted into our attack framework *CachePC* [15].

3.2 SevStep

In their paper [12], Li *et. al* present state of the art attacks against SEV-SNP in a systematic overview. Among the presented attack primitives is a "*page fault controlled channel (is)*

widely used in numerous attacks against AMD SEV", with which it is possible to "*infer the VM's activies and step its execution*".

This side-channel attack, dubbed *SevStep*, makes use of the fact that the mapping of guest physical pages to host physical pages is controlled by the host. A guest physical page can be unmapped by unsetting the *present* bit in the corresponding page-table entry. This forces the guest to halt execution and signal a page-fault to the host upon accessing the unmapped page. By unmapping all pages except those needed to execute the next instruction, it is possible to control how many instructions are executed by the guest at a time with atmost page-size granularity.

Additionally, the amount of instructions retired by the guest during execution of each page are recorded. This metric can be used to profile guest execution and detect when a guest is executing code that was profiled earlier. This is crucial for determining when guest execution has reached an area of interest to the attacker, such that they can begin analyzing cache accesses.

A refence implementation [19] of the attack primitive was made available, which aided the implementation of page-wise stepping in our attack framework *CachePC* [15].

3.3 CIPHERLEAKS

In the paper [11], Li *et. al* demonstrate that building a ciphertext-plaintext dictionary against specific CPU registers is possible under SEV-SNP. This was used to recover plaintext register values during the guest's execution and leak sensitive guest information.

Access to CPU registers after each instruction requires fine-grained control of guest execution. To this end, the authors revisited the idea of using the Advanced Programmable Interrupt Controller (APIC), previously explored in SGX-Step [5], to single-step guest execution.

Directly before the `vmrun` instruction, the CPU core's local APIC is initialized in one-shot mode. Shortly after `vmrun` is executed, the guest is interrupted and control is returned to the host. This is repeated multiple times per single-step while monitoring the ciphertext correspondig to RIP in the VMSA for changes.

This method of single-stepping was implemented in our attack framework *CachePC* [15].

4 CachePC

We developed the attack framework *CachePC* [15] for the purpose of mounting practical side-channel attacks using *Prime+Count*, a cache-based side-channel attack based on *Prime+Probe*, augmented through the use of performance counters. The following describes how *CachePC* may be used to analyze the accesses made by a specific guest instruction:

First, the guest is *page-stepped*, causing the guest to relinquish control to the hypervisor every time a new instruction page is reached. This enables tracking of guest execution on a page-wise granularity, and recording the amount of instructions retired in each page through performance counters. Once a pattern of retired instructions for previously executed pages is reached that corresponds to a region of interest in the guest's code, *single-stepping* is enabled. While single-stepping, the guest is interrupted through local APIC timer interrupts, such that never more than a single instruction can retire before returning control to the hypervisor. The ciphertext of RIP stored in the VM Save Area (VMSA) is used to detect an instruction change. For each instruction, evictions are recorded through *Prime+Count*, which yields an access pattern for each instruction in the page, including the target instruction.

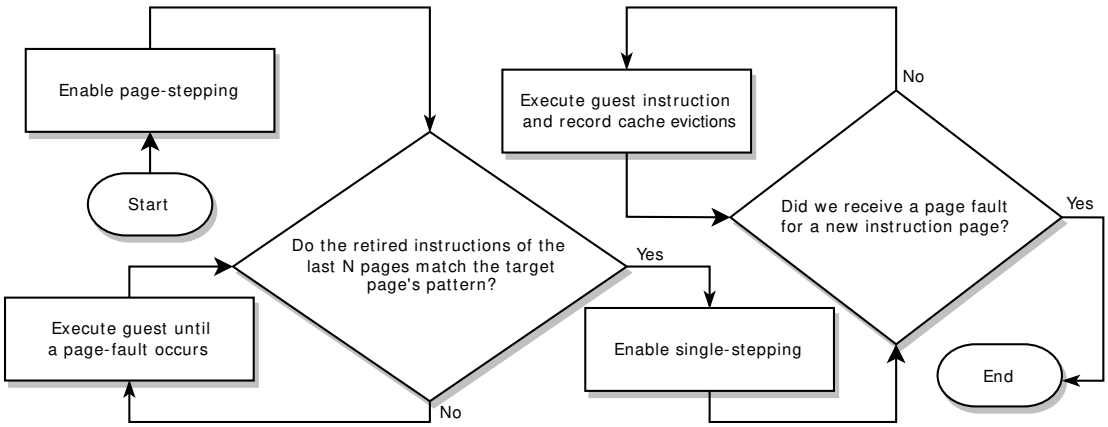

Fig. 4.1: *CachePC* Usage Example

4.1 Threat Model

The potential of cache-based side-channel attacks has already been extensively explored against competing secure virtualization technology, Intel SGX [3, 4]. This research revealed that side-channel attacks are possible with low-noise despite secure virtualization, due to native execution of guest instructions. As such, traditional microarchitectural side-channel attacks can largely be applied in the same way to secure, virtualized guests.

In contrast to unprivileged side-channel attacks, the threat model of a malicious hypervisor allows for reconfiguration of the host system to the requirements of the attack, enabling research into previously unexplored attack primitives. The use of performance counters to accurately detect cache misses presents itself as one such technology seldomly taken advantage of in side-channel attacks due to the required privilege level.

Privileged access to the hypervisor also allows for unique downgrade attacks against select hardware-accelerated cryptographic algorithms. Since the hypervisor is used to evaluate `cpuid` instructions through an NAE event (see section 2.5), the processor capabilities advertised may be falsified to remove the support for certain instruction sets, such as AES-NI. AES implementations in the guest, such as that of the Linux kernel *crypto* module, will subsequently fall back to implementations which are vulnerable against cache-based side-channels.

However, virtualized guests also pose unique new challenges. The execution of the guest is largely intransparent to the hypervisor, making it difficult to target specific guest instructions. The process and privilege level of code running in the guest is unknown, and must be inferred through an additional side-channel. Furthermore, fine grained control over the amount of instructions executed each `vmrun` requires a reproducable and accurate mechanism to interrupt the guest with an AE event.

4.2 Considerations

In this section, we consider the challenges of successfully implementing *Prime+Count*, single-stepping and page-stepping for SEV-SNP guests. In section 4.3 we address the solutions used in *CachePC*, specific to our experimental setup.

4.2.1 Prime+Count

The superiority of performance counter-enhanced *Prime+Count* to traditional *Prime+Probe* is supported by the fact that performance counters generally do not produce false negatives. Additionally, performance counters will detect evictions independent of the cpu clock and locality of the data, factors which influence the accuracy of traditional *Prime+Probe* measurements. Despite this, precautions must be taken to achieve accurate *Prime+Count* measurements. Any accesses made by a task running between or during the *Prime* and *Probe* phases will pollute the cache and invalidate the measurement. Preemption through other tasks and interrupt handlers must be avoided, and instructions between *Prime* and `vmrun` should avoid memory accesses.

Access to virtual addreses may result in a cache miss when the corresponding mapping to a physical address is not present in the TLB. In this case, the page-table located in main memory is consulted, and the retrieved PTE is cached by cpu, causing an eviction. In practice, however, host and guest pages are repeatedly accessed during single-stepping many times before the APIC *Initial Count* register reaches a value large enough for the single-step to complete. Thus, mappings for all virtual pages accessed between *Prime* and *Probe* during a successful single-step are generally already present in the TLB beforehand.

Some memory accesses between *Prime* and *Probe*, such as those made by SEV to load from and save to the VMSA during `vmrun`, can not be avoided. As long as their addresses are consistent, however, these memory accesses (observed as evictions in the cache) may be differentiated from those made by a guest instruction. We define the baseline as the smallest amount of evictions per cache set per single-step observed. By subtracting the baseline from the observed evictions for each single-step, we obtain a significantly less noisy measurement of evictions caused by the instruction running in the guest. Additionally, if multiple measurements of the same instruction can be made, any differing evicted cache sets may be filtered. The eviction caused by the target instruction is consistent and as such must then be amongst the remaining evictions.

Because of the extensive control allowed by our threat model, it is possible for us to disable prefetching entirely through BIOS and MSR configuration. Despite this, it is worth noting how *Prime+Count* can be performed successfully even when prefetching can not be disabled. In this case, the access of cache lines within a set, as well as the order of sets, must be controlled in such a way that the prefetcher is not triggered. According to [9], the stream prefetcher is triggered by a number of consecutive evictions with a consistent direction in respect to their location in the cache. This can be avoided by ordering the cache lines such that the direction and offset of each eviction in respect to the last is not

consistent.
An example of such an access pattern is: $a_N(i) = \left\lfloor \frac{N}{2} \right\rfloor + \left\lfloor \frac{i}{2} \right\rfloor \cdot (-1)^i$.

The *Probe* phase can only determine the amount of guest evictions per cache set successfully if yet unmeasured cache lines primed by the attacker are not evicted while measuring other cache lines of the same set. This requires a detailed understanding of the cache replacement policy and cache way predictor used by the CPU. Due to time constraints and missing official information on AMD's cache-way predictor, the assumption was made that cache lines are chosen through a LRU-like policy, which our results show is accurate enough in the common case.

4.2.2 Page-Stepping

Discerning each page transition made by a guest requires unmapping all guest pages containing code except the ones currrently being executed. This is done by unsetting the *present* bit in the corresponding page-table entries. Once the guest accesses a new page, it is forced to let the hypervisor handle the resulting page-fault. The *fetch* bit in the page-fault error-code can then be used to determine whether the faulting page was intended to be executed, and thus wether a new *instruction* page was reached.

Speculation prevents accurate page-stepping by causing subsequent pages to fault before they are reached. As such, speculation must be disabled to track guests with page-wise accuracy.

Instructions lying across the page boundary will cause more than one guest page to fault at once. In this case, the order of page-faults can be used to determine which is being executed next. To ensure further page transitions between the two pages mapped are detected, a single-step must be performed when more than one page-fault occurs for the same instruction. If a less-accurate page-wise tracking is acceptable to track guest execution faster, this additional single-step may be omitted.

4.2.3 Single-Stepping

To perform *Prime+Count* with minimal noise, ideally the data accesses of a single guest instruction are analyzed at a time. This requires accurate single-stepping through the use of the local APIC timer [11], which relies heavily on the consistency of the amount of time spent before entering the guest context and the amount of additional time spent in the guest by incrementing the *Initial Count* register.

To address the first of these requirements, the amount and type of instructions executed between the initialization of the local APIC and `vmrun` should be considered. Instructions that will have varying servicing time depending on caching and locality of the data accessed should be avoided. Additionally, single-stepping may be peformed multiple times with the same counter value to help mitigate the effects of inconsistent time spent before reaching the target instruction.

The second of these requirements may be addressed by setting the local APIC *Divide Configuration* register to the smallest possible value. The divisor determines the rate (amount of APIC timer clock cycles) at which the counter is decremented. A large divisor decrements the counter slower, which means the interrupt length increases more per increment, and stepping is less accurate.

Since the executing core's clock determines how fast a cycle, and by extension an instruction, completes, it too must be kept as consistent as possible. A slower clock compared to the APIC timer frequency is favourable, as it increases the opportunities at which an instruction may be interrupted.

4.3 Experimental Setup

The experiments were conducted on a Supermicro H12SSL-i V1.01 motherboard and an AMD EPYC 72F3 CPU. The motherbord bios version is 2.4 and was released on the 14. April 2022. Except for the modifications mentioned in this section, all bios settings were left at the optimized defaults.

A host kernel and qemu with the necessary modifications to support SEV-SNP were built using a modified version [16] of the AMDESE/AMDSEV [14] repo.

The influence of other tasks running on the same core is minimized by designating a specific core for running the single-threaded guest. This core is added to `nohz_full=` in the kernel commandline arguments to reduce kernel activity and scheduling of user processes. Additionally, the kernel is compiled as non-preemptive and interrupts are disabled for the duration of prime and probe, by clearing the CPU's Interrupt-Enable (IE) flag.

Prime, *Probe* and the initialization of the local APIC oneshot timer were moved as close as possible to `vmrun`. For reasons mentioned in section 4.2, the APIC oneshot timer is initialized *after* prime, but in such a way that any memory accesses required to load the APIC's settings are completed *before* prime. Since some non-guest memory accesses between *Prime* and *Probe* can not be avoided, these are made as consistent as possible by

running the kernel with Address Space Layout Randomization (ASLR) disabled, such that they will be filtered through the baseline measurement.

To keep the clock of the CPU core running the guest as consistent as possible, the `Core Performance Boost` and `Global C-state Control` CPU settings were disabled in the BIOS. Additionally, the CPU frequency governor was set to `performance` and the minimum and maximum frequecy to 1.5 GHz. We chose the lowest of the available frequency steps, since we observed frequency drops for other configurations despite the set limits.

The cache stream prefetcher and CPU speculation were disabled through the BIOS and MSR configuration to reduce side effects from speculation.

To verify our attack primitives in a low-noise environment we developed minimalist kvm guests written in 8086 real-mode compatible assembly. This allowed for precise control over the amount of guest-induced data accesses per single-step, as well as testing of specific edge-cases, such as instructions lying across page boundaries, in a controlled manner.

4.4 Experimental Results

4.4.1 Prime+Count

Our performance-counter enhanced version of traditional *Prime+Probe* allowed for precise detection of evictions by the guest per instruction step.

Prime and *Probe* were implemented in the form of GNU assembler macros, such that they can be inserted around `vmrun` in the *kvm* kernel module without the need for an additional `call`, which would result in stack accesses and additional evictions.

Prime is performed multiple times per `vmrun`, since the cache replacement policy is only LRU-*like* and occasionally not all cache lines are evicted by a single `prime_pass` (see Listing. 4.1). Priming the cache four times resulted in the most consistent measurements.

```
1 .macro barrier
2     mfence # finish load and stores
3     mov $0x80000005, %eax
4     cpuid # prevent reordering
5 .endm
6
7 .macro prime_pass name cl_in cl_tmp cl_out
8     barrier
9
```

```
    mov \cl_in, \cl_tmp
.rept L1_SETS * L1_ASSOC - 1
    mov CPC_CL_NEXT_OFFSET(\cl_tmp), \cl_tmp
.endr
    mov \cl_tmp, \cl_out
    mov CPC_CL_NEXT_OFFSET(\cl_tmp), \cl_tmp

    barrier
.endm

.macro prime name cl_in cl_tmp cl_out
    prime_pass pass1_\name \cl_in \cl_tmp \cl_out
    prime_pass pass2_\name \cl_in \cl_tmp \cl_out
    prime_pass pass3_\name \cl_in \cl_tmp \cl_out
    prime_pass pass4_\name \cl_in \cl_tmp \cl_out
.endm
```

Listing 4.1: *CachePC Prime* in GAS-Syntax Assembly

In the *Probe* phase, the L1 cache miss performance counter is queried before and after traversing all cache lines of a set to determine the amount of cache lines evicted by the victim for that set (see Listing. 4.2).

```
.macro readpmc event out
    barrier

    mov $0, %rax
    mov $0, %rdx
    mov $0xc0010201, %rbx
    mov \event, %rcx
    shl $1, %rcx
    add %rbx, %rcx
    mov $0, %rbx
    rdmsr
    shl $32, %rdx
    or %rax, %rdx
    mov %rdx, \out

    barrier
.endm

.macro probe name pmc cl_in cl_tmp1 cl_tmp2 pmc_tmp1 pmc_tmp2
    barrier

    mov \cl_in, \cl_tmp1

probe_loop_\name:
```

```
25      readpmc \pmc \pmc_tmp1
26
27  .rept L1_ASSOC - 1
28      mov CPC_CL_PREV_OFFSET(\cl_tmp1), \cl_tmp1
29  .endr
30      mov CPC_CL_PREV_OFFSET(\cl_tmp1), \cl_tmp2
31
32      readpmc \pmc \pmc_tmp2
33
34      sub \pmc_tmp1, \pmc_tmp2
35      mov \pmc_tmp2, CPC_CL_COUNT_OFFSET(\cl_tmp1)
36
37      mov \cl_tmp2, \cl_tmp1
38      cmp \cl_in, \cl_tmp1
39      jne probe_loop_\name
40
41      barrier
42  .endm
```

Listing 4.2: *CachePC Probe* in GAS-Syntax Assembly

Since the L1 prefetcher was disabled, a special ordering of cache lines was not necessary.

The implemented *Prime+Count* was accurate enough to detect which set the line evicted by an instruction was in among a small amount of false positives.

Despite the precautions taken, we observed *Prime+Count* measurements being invalidated with a probability of roughly 0.1%. We were not able to discern the source, but among the potential causes are unmaskable interrupts, spurious interrupts or `wbinvd` instructions dispatched on other cores.

4.4.2 Page-Stepping

Modifying the kernel's page-fault handling in the way demonstrated by *A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP* [12], as described in 3.2, and disabling speculation, allowed for consistent page-wise stepping of minimalist kvm and qemu guests, independent of SEV protections. Our implementation was modified in respect to the original *SevStep* [19] to allow for correct handling of edge-cases, such as instructions lying on a page boundary.

Furthermore, we are able to infer which guest pages contain code by observing whether the *fetch* bit is set in their page-fault error code. Initializing two *retired instructions* performance counters to record the amount of guest instructions retired for both CPL $= 0$ and CPL $\neq 0$, we found it is even possible to discern user from kernel pages.

4.4.3 Single-Stepping

We were able to successfully and consistently single-step kvm guests, independent of the SEV protections present. This result was verified by observing changes in the guest's RIP register and the amount of evictions recorded each single-step.

The guest RIP register can be decrypted under SEV-ES and SEV-SNP guests by enabling debugging through the guest policy, which allows for decryption of the VMSA entry for RIP, as pointed out in *CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel* [11]. To access the VMSA contents, the kernel was modified to prevent the VMSA from being removed from the direct map upon migrating the corresponding page ownership to the SEV firmware.

Combining page-fault analysis with single-stepping allowed correctly inferring whether a single-step involves a data access, and whether the evictions deduced by *Probe* are relevant for the given step.

4.5 Example

In the following we demonstrate the *Prime+Count* side-channel attack against a secure encrypted guest under SEV-SNP, running a minimal example program. Since the feasibility of page-wise tracking was already demonstrated by Li *et. al* [12] and is not crucial for *Prime+Count* to be applied succesfully, it is not the focus of this example.

The guest assembly (Listing. 4.3) consists of a data access to address `0x12c0`, surrounded by `nop` instructions, and a `jmp` back to the first instruction. The read access to `0x12c0` results in an eviction in set 11 of the L1 cache when performed after *Prime*, and is used to demonstrate the accuracy of *Prime+Probe*. The `nop` instructions are difficult to single-step, because they perform no computation and thus can be executed extremely quickly by the cpu. They will be used to demonstrate the accuracy of single-stepping.

```
1  0x00:   90                      nop
2  0x01:   bb c0 12                mov $0x12c0,%bx
3  0x04:   8a 1f                   mov (%bx),%bl
4  0x06:   90                      nop
5  0x07:   eb f7                   jmp $0
```

Listing 4.3: Guest Assembly in GAS-Syntax

First, the guest assembly is single-stepped to calculate an adequate baseline of cache evictions, which will be subtracted from future measurements. The following table shows

the evictions included in the baseline measurement. Each entry corresponds to the cache set denoted by the sum of column and row offset (first row & column).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	02	02	02	03	01	01	00	01	01	01	01	01	01	02	01	01
16	03	04	03	03	02	03	01	03	02	03	01	02	01	01	00	01
32	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00

Tab. 4.1: Baseline evictions

Single-stepping the guest assembly with *Prime+Count* enabled results in the following eviction profiles:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
32	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Tab. 4.2: Evictions for `nop`, new RIP is `0x01`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	03	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
32	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Tab. 4.3: Evictions for `mov $0x12c0, %bx`, new RIP is `0x04`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	06	00	00	00	00	00	00	00	00	00	00	01	00	00	00	00
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01
32	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Tab. 4.4: Evictions for `mov (%bx), %bl`, new RIP is `0x06`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	04	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
32	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Tab. 4.5: Evictions for `nop`, new RIP is `0x07`

The first and last `nop` instructions were correctly single-stepped and resulted in a similar eviction profile, demonstrating the consistency of single-stepping and *Prime+Count*.

The eviction profile of the access instruction reveals a subset of cache sets the guest access could have taken place in. In this case, the candidates are 0, 11 and 31, demonstrating that the true evicted set 11 is among them. By single-stepping the same guest instructions multiple times, the number of candidates may be reduced further.

5 Conclusion

In conclusion, we were successful in undermining AMD Secure Encrypted Virtualization (SEV) through the *Prime+Count* cache side-channel attack, since the confidentiality of guest physical addresses accessed by the guests was undermined. Therefore, bits of secret-dependent guest accesses can be leaked and, when combined with an AES downgrade attack, potentially used to infer a guest's AES private key.

Although we did not succeed in successfully mounting *Prime+Count* against a specific process or kernel module running in a fully-fledged Linux guest due to time constraints, we are confident that such an attack is possible.

By providing the attack framework *CachePC* [15], we hope to lay the ground work for further research into microarchitectural side-channel attacks against AMD SEV.

5.1 Future Work

Identifying and eliminating the source of the spurious cache evictions we observed during our experiments would enable more accurate tests to infer the cache replacement policy and develop a suitable priming strategy. A more sophisticated analysis of the cache line replacement policy through the use of tools such as [18] and research into the cache way predictor [8] could then be used to determine more suitable *Prime* and *Probe* strategies to further improve the accuracy of the attack.

As the paper *A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP* [12] has demonstrated, page-fault analysis can be used to create a fingerprint of guest execution. In their research, a tuple of page fault number and retired instructions was used to fingerprint guest execution. A more detailed fingerprinting method may involve counting not only retired instructions, but using performance counters related to instructions classes such as branch- and return instructions to more accurately identify code executed by the guest.

Glossary

AE Automatic Exit. 8, 14

AES Advanced Encryption Standard. 2, 7, 9, 14, 24

AES-NI AES-New Instructions. 14

AES-XEX AES-Xor Encrypt Xor. 9

AMD Advanced Micro Devices Inc.. iv, 1–3, 8, 9, 16, 24

AMD-SP AMD Secure Processor. 1, 9

APIC Advanced Programmable Interrupt Controller. 1, 3, 7, 12, 13, 15–17

ARM ARM Instruction Set Architecture. 9

ASID Address Space Identifier. 8, 9

ASLR Address Space Layout Randomization. 18

GAS GNU Assembly Syntax. 6, 19–21

GHCB Guest-Host Control Block. 8

guest A system virtualized on a host. iv, 1, 2, 7–9, 12–18, 20–22, 24, 25

host A system interacting with hardware directly. iv, 8, 12, 14, 17

hypervisor A host virtualizing one or more guests. iv, 1, 8, 9, 13, 14, 16

IE Interrupt-Enable. 17

Intel Intel Corporation. 1, 14

LRU least recently used. 16

MMU Memory Management Unit. 3

NAE Non-Automatic Exit. 8, 14

page-step Interrupting execution every new instruction page reached. 13, 14, 16

performance counter A CPU register used to count system events. iv, 1, 13–15

PIPT Physically Indexed, Physically Tagged. 4

PTE Page Table Entry. 3

SEV Secure Encrypted Virtualization. iv, 1–3, 8, 9, 15, 20, 21, 24

SEV-ES SEV-Encrypted State. 8, 21

SEV-SNP SEV-Secure Nested Paging. iv, 1, 2, 8, 11, 12, 14, 17, 21

SGX Intel Software Guard Extensions. 1, 12, 14

single-step Interrupting execution every new instruction reached. 1, 12–18, 21, 23

SME Secure Memory Encryption. 9

TLB Translation Lookaside Buffer. 3, 4, 9, 15

trusted platform module Secure cryptoprocessor with integrated keys. iv, 9

VC VMM Communication. 8

VIPT Virtually Indexed, Physically Tagged. 4

virtual machine A Smsystem with emulated access to hardware. iv, 8

VMCB VM Control Block. 8

VMSA VM Save Area. 8, 12, 13, 15, 21

Bibliography

[1] David Kaplan, Jeremy Powell, and Tom Woller. “AMD memory encryption”. In: *White paper* (2016) (cit. on pp. 1, 8, 9).

[2] AMD. “Protecting VM register state with SEV-ES”. In: *White paper* (2017) (cit. on p. 8).

[3] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, et al. “Software Grand Exposure: SGX Cache Attacks Are Practical.” In: *WOOT*. 2017, pp. 11–11 (cit. on pp. 1, 14).

[4] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. “Cache attacks on Intel SGX”. In: *Proceedings of the 10th European Workshop on Systems Security*. 2017, pp. 1–6 (cit. on pp. 1, 14).

[5] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “SGX-Step: A practical attack framework for precise enclave execution control”. In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 2017, pp. 1–6 (cit. on p. 12).

[6] AMD. “Strengthening VM isolation with integrity protection and more”. In: *White Paper, January* (2020), p. 8 (cit. on pp. 1, 8).

[7] Miro Haller. *Revisiting Microarchitectural Side-Channels*. Project Report. 2020 (cit. on pp. 4–6, 10, 11).

[8] Moritz Lipp, Vedad Hadžić, Michael Schwarz, et al. “Take a way: Exploring the security implications of AMD’s cache way predictors”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 813–825 (cit. on p. 25).

[9] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. “Reverse engineering the stream prefetcher for profit”. In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2020, pp. 682–687 (cit. on p. 15).

[10] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. “Crossline: Breaking "security-by-crash" based memory isolation in AMD SEV”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2937–2950 (cit. on p. 9).

[11] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. “CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel.” In: *USENIX Security Symposium*. 2021, pp. 717–732 (cit. on pp. 12, 16, 21).

[12] Mengyuan Li, Luca Wilke, Jan Wichelmann, et al. “A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 337–351 (cit. on pp. 11, 20, 21, 25).

Webpages

[13] AMD. *AMD64 Architecture Programmer's Manual*. 2023. URL: https://www.amd.com/system/files/TechDocs/40332.pdf (visited on Feb. 23, 2023) (cit. on pp. 3, 6, 7).

[14] AMD. *AMDSEV*. 2023. URL: https://github.com/AMDESE/AMDSEV (visited on Feb. 10, 2023) (cit. on p. 17).

[15] Louis Burda. *CachePC*. 2023. URL: https://github.com/Sinitax/cachepc (visited on Feb. 10, 2023) (cit. on pp. 11–13, 24).

[16] Louis Burda. *CachePC AMDSEV*. 2023. URL: https://github.com/Sinitax/cachepc-AMDSEV (visited on Feb. 10, 2023) (cit. on p. 17).

[17] Miro Haller. *CacheSC*. 2023. URL: https://github.com/Miro-H/CacheSC (visited on Feb. 10, 2023) (cit. on p. 11).

[18] Pepe Vila. *CacheQuery*. 2023. URL: https://github.com/cgvwzq/cachequery (visited on Feb. 10, 2023) (cit. on p. 25).

[19] Luca Wilke. *SevStep*. 2023. URL: https://github.com/UzL-ITS/sev-step (visited on Feb. 10, 2023) (cit. on pp. 12, 20).