Hannes Weissteiner, BSc

CRACKPIPE
Covertly Reconstructing Arbitrary Code tracKs using Per-Instruction Performance-counter Evaluation

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Advisors

Stefan Gast

Daniel Gruss

Institute of Applied Information Processing and Communications

Graz, April 2024

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Acknowledgements

Firstly, I would like to thank my advisors Stefan Gast and Daniel Gruss for all the discussions and brainstorming sessions.

Special thanks to Luca Wilke, for helping with the SEV-Step setup and getting it to work reliably on our systems.

I also thank Sebastian Felix for his help with debugging and proofreading, as well as motivating me to finish this thesis.

Additionally, I am thankful to Lena Heimberger and Andreas Kogler for their support and for motivating me to finish this thesis.

I thank everyone at LosFuzzys for providing an environment that allowed me to further my interest and knowledge in information security. Thank you all for the insightful discussions, long nights in the Lab, and fun adventures we had.

I would also like to extend a special thanks to my parents, who have always supported me throughout my studies.

Lastly, I am grateful to my girlfriend Michelle for her unwavering support and understanding, even when I returned home late after a long night at the FuzzyLab.

Abstract

AMD SEV is a processor extension that has been introduced to provide cloud customers with integrity and confidentiality guarantees in a shared hosting environment. Encryption, combined with access restrictions, ensures that a cloud provider cannot access or modify customer data. For this purpose, SEV and its extensions ES and SNP enable a variety of new features to limit the capabilities of a potentially malicious hypervisor. In this thesis, we demonstrate that some processor performance statistics available to privileged attackers can be used to break the system's confidentiality guarantees. We present the `CRACKPIPE` attack, in which a malicious hypervisor abuses performance counter differences to leak secret data out of an SEV-SNP protected virtual machine. `CRACKPIPE` interrupts the guest virtual machine after each instruction to precisely determine the outcomes of conditional branches, and recovers secrets from the resulting trace. Therefore, despite enabling all protection features of AMD SEV-SNP, the CPU remains vulnerable to `CRACKPIPE`. In multiple case studies, we demonstrate how the underlying primitive can be used in real-world attacks to recover keys from cryptographic algorithms, such as RSA, or drastically reduce the time complexity of a brute-force attack. Finally, we discuss how mitigations for `CRACKPIPE` impose trade-offs, such as performance overhead or limiting the hypervisor's ability to detect malicious guests.

Keywords: AMD, Cloud Providers, Performance Counter, SEV, SNP, Side Channel, Single Trace, TEE, Virtual Machines

Kurzfassung

AMD SEV ist eine Prozessorerweiterung, die eingeführt wurde, um Cloud-Kunden Integritäts- und Vertraulichkeitsgarantien in einer gemeinsamen Hosting-Umgebung zu bieten. Verschlüsselung und Zugriffsbeschränkungen stellen sicher, dass ein Cloud-Anbieter nicht auf Kundendaten zugreifen oder sie verändern kann. Um dies sicherzustellen, bieten SEV und seine Erweiterungen ES und SNP eine Vielzahl neuer Funktionen, um die Möglichkeiten eines potentiell bösartigen Hypervisors einzuschränken. In dieser Arbeit wird gezeigt, dass einige Prozessorleistungsstatistiken, die privilegierten Angreifern zur Verfügung stehen, zuverlässig gemessen werden können, um die Vertraulichkeitsgarantien des Systems zu brechen. Wir stellen den `CRACKPIPE`-Angriff vor, bei dem ein böswilliger Hypervisor Unterschiede in Performance-Countern missbraucht, um geheime Daten aus einer SEV-SNP-geschützten virtuellen Maschine zuverlässig zu extrahieren. `CRACKPIPE` unterbricht die virtuelle Gastmaschine nach jeder Instruktion, um die Ergebnisse von Conditional Branches genau zu bestimmen, und extrahiert sicherheitsrelevante Daten aus den resultierenden Messwerten. Daher bleibt die CPU trotz Aktivierung aller Schutzfunktionen von AMD SEV-SNP anfällig für `CRACKPIPE`. Mehrere Fallstudien zeigen, wie das zugrundeliegende Primitiv in realen Angriffen verwendet werden kann, um Keys von kryptographischen Algorithmen wie RSA wiederherzustellen oder die Zeitkomplexität eines Brute-Force-Angriffs drastisch zu reduzieren. Abschließend wird diskutiert, dass die Verteidigung gegen `CRACKPIPE` mit Kompromissen verbunden ist, wie z.B. der Einschränkung der Fähigkeit des Hypervisors, bösartige Gäste zu erkennen, oder einem Performance-Overhead.

Schlagwörter: AMD, Cloud Anbieter, Performance Counter, SEV, SNP, Side Channel, Seitenkanal, Single Trace, TEE, Vertrauenswürdige Ausführungsumgebung, Virtuelle Maschinen

Contents

Chapter 1.

Introduction

Cloud providers play a huge role in modern computing. Moving away from local infrastructure to cloud services has numerous advantages: It relieves local administrators from having to deal with hardware issues and having to care about hardware upgrades. Furthermore, it allows quick scaling of resources to meet demand while minimizing waste of computing power during downtimes. Lastly, up-front costs for servers and networking setups are eliminated. Usually, machines running in a cloud instance are virtualized, which enables them to be moved dynamically between physical machines, depending on demand and load. Virtual machines are controlled by a *Hypervisor*, which runs at a higher permission level than the kernel of the virtual machine. The hypervisor manages access to the underlying hardware between multiple Virtual Machines (VMs), virtualizing hardware features and managing access to privileged resources. Traditionally, the hypervisor can access all data in a virtual machine, including register values, memory, and storage. As a result, traditional cloud computing requires trust in the provider, as all secrets are accessible to them.

This makes traditional virtual machines unsuitable for handling sensitive, critical data. To solve this issue, hardware vendors came up with solutions that would limit hypervisor access. Technologies like Intel Secure Guard Extensions (SGX), ARM `TrustZone`, and AMD Secure Encrypted Virtualization (SEV) provide a Trusted Execution Environment (TEE), which is supposed to protect secrets from malicious physical and software access.

In this work, we focus on AMD SEV, using the latest extension called Secure Nested Paging (SNP). In contrast to others, AMD encrypts complete virtual machines in a way that is supposed to prevent the hypervisor from obtaining any information on their data. The bootstrapping process is as follows: The user can upload a VM image, complete the attestation process to make sure the hardware is genuine and has not been tampered with, and then supply either the sensitive data or the decryption key for it [6]. In this process, the guest owner can provide a *guest policy*, in which they can configure some settings for the virtual machine. AMD specifies that they do not protect hypervisors from fingerprinting software in the guest via performance counters or page access tracking. AMD argues that attackers gaining information about the executed code is not critical because sensitive information is stored as data, not code [6].

In this thesis, we show that this assumption does not hold. Using single-stepping, we obtain detailed information about the execution inside the VM. We leak sensitive data from the VM by observing performance counters, which AMD has *explicity* declared as non-critical [6]. We show that, in some cases, an attacker is able to recover private keys by observing performance counters during cryptographic computations. Even though information about the state inside the guest is limited, we show that attacks can be targeted precisely, with multiple redundant methods to ensure our assumptions are correct.

1.1. Motivation

Many previous attacks on AMD SEV rely on the ability to observe and modify unencrypted register states [31, 76] or the ability to modify or remap guest physical pages [31, 23, 51, 40, 77]. Through changes introduced by the SEV extensions Encrypted State (ES) and Secure Nested Paging (SNP), these types of attacks are mitigated by design. To mitigate attacks that leak data by observing the encrypted register state [42], AMD introduced the *VMSA Register Protection* feature. Other attacks use CPU bugs to exploit encrypted guests [82] or rely on noisy power measurements to exfiltrate data [73].

In contrast, `CRACKPIPE` uses performance counters, a legitimate CPU feature, to leak data from an encrypted guest virtual machine in a single trace. It achieves an instruction-level resolution for this data by single-stepping the guest. While single-stepping an SEV guest without its consent is not an intended feature of AMD SEV, it is achievable using `APIC` functionality. Additionally, we use page faults to improve reliability further. Since these three primitives are legitimate features used in a new way, mitigations for `CRACKPIPE` will require tradeoffs between multiple factors like security of host and guest, manageability of system load, and performance.

Contributions

1. We show that even though AMD has declared performance counters and page tracking as not problematic from a security standpoint, we can obtain information about data inside the VM using those primitives.
2. We show how different techniques can be combined to efficiently mount an end-to-end attack.
3. We recover a 4096-bit private key from a program decrypting a ciphertext using Mbed TLS in a single trace within 8 minutes using this attack.
4. We show how `CRACKPIPE` can be used for brute-forcing TOTP tokens in an average of 30 attempts.
5. We recover the secret from a TOTP in a single trace.
6. We discuss potential mitigations and their drawbacks.

1.2. Structure of this Document

Chapter 2 explains how SEV and its extensions work and gives an overview of existing attacks on SEV and other trusted execution environments. In Chapter 3, we explain the different primitives used by the attack. Chapter 4 details how we implement the different primitives and the actual attack. In Chapter 5, we present example victim programs and discuss which factors play into the reliability of the attack. Chapter 6 discusses possible mitigations for this attack. Chapter 7 concludes.

Chapter 2.
Background

2.1. Virtualization

Virtualization is a technology that allows multiple VMs to run on the same host machine, each seemingly independent of the other. This process involves emulating and virtualizing underlying hardware, like memory, CPU-internal timers, PCI-, or other IO devices. The hypervisor manages and allocates resources for the whole machine. It, therefore, operates at a higher permission level than the kernel of each VM. The hypervisor is responsible for distributing the available system resources between all VMs and deciding when to schedule each guest. While it is possible to abstract a system in a way so that it behaves almost exactly like real hardware, making the guest oblivious to the fact it is being virtualized (*full virtualization*), a more efficient approach is to use *paravirtualization*. With paravirtualization, the guest is aware of the virtualization and cooperates with the hypervisor using *hypercalls* to improve performance and reliability [52]. Hypercalls are a mechanism similar to system calls; however, instead of calling the guest's kernel, they call the hypervisor.

Virtualization usually works by limiting the virtual machine's ability to interact with hardware components and configuration. This is achieved by trapping privileged instructions and delegating them to the hypervisor. After verifying the guest's permissions, the hypervisor can then either run the instructions or emulate them. While some techniques, like binary patching, enable this functionality in software only [62], hardware-assisted virtualization is the norm nowadays [67, 64]. Processors that support hardware-assisted virtualization provide specific instructions, registers, and functionalities to enable efficient and selective trapping of privileged instructions and configuration of the processor's behavior. Different instruction sets implement these features in different ways. In the case of x86-64 CPUs, Intel and AMD provide different extensions for hardware virtualization.

While Intel's Virtual Machine Extensions (VMX) and AMD's Secure Virtual Machine (SVM) have similar principles of operation, their implementation differs in multiple places. Both have different instructions for interacting with VMs, various extensions and features, and different *hypercall* instructions. This work focuses on SVM, and more specifically the SEV mode with the SNP extension.

2.2. Software Guard Extensions (SGX)

Intel's SGX is a set of extensions to the x86_64 architecture on supported Intel processors that allows users to execute code on an untrusted remote machine while protecting secrets [18]. SGX achieves this by executing code in a secure enclave. The computation within the enclave is safeguarded against interference from the operating system and hypervisor. The integrity of the enclave is verified using an attestation process.

Processors that support SGX reserve a special memory region known as *Processor Reserved Memory*. This memory region is protected against all non-enclave accesses by the hardware. The system loads the initial data into the enclave when starting the enclave. This process is performed by untrusted software, which can see and interact with the initial data. However, during the attestation process, users can verify the integrity of the initial enclave data using a hash.

The enclave can be enetered from userspace by executing the newly introduced `EENTER` instruction. When an interrupt occurs, SGX performs an *Asynchronous Enclave Exit*. During such an asynchronous exit, the CPU saves the state of the enclave and clears register values before exiting the enclave. Afterward, the operating system can handle the interrupt. To transfer control back to the enclave, the process running the enclave uses the `ERESUME` instruction.

In contrast to SEV, which encrypts an entire virtual machine, SGX enclaves are designed to contain only the most security-critical parts of a program. This approach has the advantage of a smaller codebase inside the enclave, which makes it easier to maintain and audit for security issues. However, attackers are also aware of the precise location of the victim's secrets.

Since its introduction, SGX has been the target of numerous attacks. Since many of the principles behind these attacks also apply to SEV, we discuss some of them in the following section. We only consider attacks that do not rely on faulty software implementations, such as buffer overflows or logic errors, in the program running inside the enclave.

Attacks on SGX

New attack vectors emerge in an environment where the operating system is considered malicious. One such type of attack is called *controlled-channel attack* [80]. Controlled-channel attacks leak secret data from secret-dependent memory access patterns. Attackers can gather access information by unmapping the pages that will be accessed in the enclave, which requires operating system privileges. When a memory access occurs, the operating system recieves a page fault event that contains information about which page was accessed. Depending on the specific program, this information can be used to leak secret keys. In a traditional model, the operating system can always read the entire system memory. Therefore, this side channel was not relevant before. However, it becomes a valid attack on a TEE.

As a follow-up to this attack, Van Bulck et al. [71] discovered a method for monitoring memory accesses without generating page faults. They introduced two primitives: One primitive monitors the `accessed` and `dirty` bits in the page tables of the target page to determine if there was a memory read or write on that page. The other primitive makes use of the fact that accessing a memory location loads it into the cache. As demonstrated by *Flush+Reload* [81], it is possible to determine if a memory location was accessed by flushing a cache line and later measuring the access time between the flush and the measurement. The measured access time is lower if the memory location was accessed between the flush and the timed reload. However, it is impossible to clear the encrypted data pages' caches in the enclave from the outside. Instead, the authors use the page table access latencies to leak memory access information.

Van Bulck et al. [70] also introduced *SGX-Step*, a framework that allows users to single-step SGX enclaves. It also allows modifications to page table from userspace, eliminating the need for custom syscalls to attack an enclave via controlled channels. Single-stepping the enclave is achieved by configuring a timer to interrupt the enclave after one instruction. They use the accessed bits of the instruction pages to distinguish between single-steps and zero-steps. SGX-Step serves as a framework in a number of other attacks on SGX [68, 60, 32, 50, 10]

The *Nemesis* attack [69] involves stepping through an enclave and measuring interrupt latencies. The time required to handle interrupts varies depending on the execution time of different classes of instructions. This fact is used to classify instruction types, enabling the distinction between different code paths, even with the same number of instructions.

The *Prime+Probe* [56] side channel fills an entire cache set with attacker-controlled data, evicting all other cache entries. When another process accesses memory, the corresponding cache line is replaced with the victim's data. The evicted portion of the attacker's data will have an increased access latency, thereby leaking the cache set accessed by the victim. Unlike other cache side-channels like *Flush+Reload* [81], *Flush+Flush* [28] and, *Evict+Reload* [29], Prime+Probe does not require a shared buffer. It does not need to flush or access the data at all to leak access information.

In their *CacheZoom* attack, Moghimi et al. [49] used this property of Prime+Probe to track enclave memory accesses. Some AES implementations use T-tables to improve performance, by combining the MixColumn and SubBytes steps in a single table lookup. They were able to recover the AES encryption keys from a SGX enclave by measuring memory access patterns. Additionally, Prime+Probe is used in several cache-based attacks on SGX as a means to measure microarchitectural state [27, 11, 19]

Schwarz et al. [61] also performed a cache attack using Prime+Probe. However, unlike other attacks on SGX, their attack code was placed inside an SGX enclave. Enclaves are designed to prevent anyone, including the operating system, from extracting the code or data from the enclave. Therefore, the CPU prevents the host machine from detecting the malicious code. This attack can target not only SGX enclaves but also other software running on the system.

Other attacks on SGX include branch predictor attacks [38, 24, 32]. Branch predictor attacks work similarly to cache attacks. However, instead of flushing or evicting cache lines and measuring the access time to determine whether a memory location was accessed, they attack data structures in the branch predictor unit of the CPU. Although branch prediction performance information is not tracked inside enclaves, the branch predictor state is still updated. If the attacker completely fills a data structure in the branch predictor and executes the enclave, portions of the contents may be overwritten depending on the specific instructions executed in the enclave. Attackers can then determine whether their branch outcomes are correctly predicted by, for example, measuring execution time or obtaining performance information from the CPU.

Sgxpectre [14] and *SpectreRSB* [37] demonstrate that speculative execution attacks [35] also apply to SGX enclaves. Since SGX runs on a standard processor core, albeit with additional security measures, performance optimizations such as branch prediction are used. However, since the enclave memory is inaccessible to the attacker, the preparing the branch predictor states and extracting the leaked data requires some additional steps. Sgxpectre first poisons the *Branch Target Buffer* by repeatedly performing indirect jumps from the start address to the desired address. Then, Sgxpectre drains the *Return Stack Buffer* by returning several times, before entering the enclave. This forces the processor to use the poisoned branch target buffer for branch prediction. The enclave is then forced to speculatively access an area of shared memory to extract the data. The area that is being accessed can then be measured using Flush+Reload, allowing the data to be exfiltrated from the enclave.

SpectreRSB pollutes the return stack buffer by executing the `CALL` instruction at a desired address, which also works speculatively. Then, SpectreRSB edits the return address on the stack, which does not update the return stack buffer. This causes the next `RET` instruction to speculatively jump to the instruction after the initial `CALL`. By placing a leak gadget at this mispredicted return location, the attacker can use Flush+Reload to leak data. This return branch misprediction works across enclave boundaries.

Although SGX protects the enclave memory from architectural reads from the outside, internal CPU buffers may still contain secret data. Microarchitectural Data Sampling (MDS) attacks are a class of attacks that specifically target the extraction of data from these internal buffers.

The *Rogue In-Flight Data Load* [72] attack leaks data from the *Line Fill Buffer (LFB)* by accessing not-yet-mapped pages, causing a page fault. The CPU speculatively uses data from the LFB to continue execution before reverting when the page fault occurs. This transient data can be extracted using Flush+Reload. *ZombieLoad* [60] extends this primitive, and additionally leaks LFB data by inducing TSX faults, and by exploiting microarchitectural page faults that trigger when the `access` bit of a page table entry is 0.

While the other MDS attacks use the LFB, which is only shared between threads on the same core, *CrossTalk* [58] uses the *Staging Buffer* to leak data across CPU cores. They are able to use this primitive to leak the result of the `RDRAND` instruction across cores, recovering all randomness from the victim.

Plundervolt launches a fault attack at SGX enclaves by undervolting the processor [53]. By lowering the operating voltage of the processor using dedicated MSRs, they are able to fault power-intensive instructions like multiplication and AES-NI instructions. They leverage the faulted multiplications by mounting a *Bellcore* attack on RSA [9]. With this attack, they can recover the complete encryption key. Additionally, they apply differential fault analysis to the faulted AES instructions, which allows them to recover the AES key of an encryption in a couple of minutes.

ÆPIC Leak is a flaw in certain Intel processors that causes read operations to return stale data from internal CPU buffers when reading from undefined registers from the Advanced Programmable Interrupt Controller (APIC) [10]. They find that, when reading from the APIC, undefined data offsets are not properly cleared, resulting in reads returning seemingly random data. The leaks correspond with data that recently traveled through the cache structure. The authors introduce multiple techniques to manipulate victim data to travel through caches, thereby increasing the likelihood of leaking secrets. A notable difference between Æpic leak and other attacks on SGX is that the data leak is entirely architectural, meaning that no microarchitectural extraction techniques such as cache timings are required.

2.3. Secure Virtual Machine (SVM)

With SVM, AMD provides functionality that enables the hypervisor to securely and efficiently run virtual machines with hardware assistance [7]. The behavior of SVM is configurable using Model-Specific Registers (MSRs) (for global settings) and a Virtual Machine Control Block (VMCB) per guest (for guest-dependent settings).

MSRs are hardware registers that are specific to processor models. They are used to enable and configure processor features. MSR options for SVM include, e.g., which extensions of SVM are enabled or where the host state will be saved.

The VMCB is a data structure that SVM uses for the configuration of VMs. It consists of a continuous physical memory area and needs to be passed to the `VMRUN` instruction whenever the hypervisor wants to schedule a guest. The VMCB provides virtual-machine-specific options that the hypervisor can configure based on each guest's individual requirements:

Intercept type	**Description**
Instruction intercepts	Configures whether to intercept specific instructions executed by the guest. Examples include `CPUID`, `IRET`, reads and writes to control registers.
IOIO intercepts	Configures whether to intercept the virtual machine when specific IO ports of the CPU are accessed.
MSR intercepts	Configures whether to intercept the virtual machine when specific MSRs are accessed.
Exception intercepts	Configures which exceptions in the guest or the host should be intercepted.
Feature settings	Configures which CPU features are enabled for the guest.

Table 2.1.: Available intercepts and settings for SVM

2.3.1. Basic Operation

To be able to run a virtual machine, the hypervisor, also called Virtual Machine Monitor (VMM) in the documentation, first has to set up a VMCB with the chosen options and settings for the specific guest. It then has to save its own state, move the physical address of the VMCB to `RAX`, and execute the `VMRUN` instruction to give control to the guest.

Whenever an intercept triggers, a `#VMEXIT` occurs. This transfers control back to the hypervisor, which has to save the guest state to resume that guest later. It saves some register values to a hypervisor-defined data structure and executes the `VMSAVE` instruction to save the rest of the guest state to a VM Save Area (VMSA) page.

Afterward, the hypervisor can use fields in the VMCB to determine why the `#VMEXIT` was triggered. Common causes for `#VMEXIT`s are hardware interrupts (e.g., key presses), which the hypervisor might forward to the VM by triggering an interrupt in the guest. The hypervisor achieves this using specific fields in the VMCB. Other common `#VMEXIT` reasons include interactions with hardware (real or virtualized) or configuration changes by the guest (e.g., MSR reads/writes). In those cases, the hypervisor can set the guest registers to the correct values in the VMSA and resume the virtual machine.

The hypervisor can resume the virtual machine by executing the `VMLOAD` instruction, restoring the remaining registers, and calling the `VMRUN` instruction to enter the guest.

2.3.2. Nested Paging

Modern processors use *paging* to separate the memory of different user processes. They create a continuous *virtual address space* for each process by mapping *physical pages* to the correct location in the process's address space. When looking for a specific virtual address, the processor walks through the page table levels to find the corresponding physical address. This allows the same virtual address to point to two physical pages in different processes.

Another address translation layer is introduced to achieve a similar separation for VMs. AMD calls this *nested paging*. It follows the same principle: Every Guest Physical Address (gPA) is mapped to a system physical address using a Nested Page Table (nPT). A graphic representation of this process can be seen in Figure 2.1. Therefore, every virtual machine has its own physical address space and cannot access the memory of other VMs or the host machine.

Whenever a process in a guest accesses an address, the processor tries to find it by using the Guest Page Table (gPT). Due to the fact that the addresses for the gPT are gPAs, it has to look up every level in the nPT to find the actual location in physical memory.

In SVM, the gPT has the same layout as standard page tables. Therefore, the hypervisor can use the same bits (e.g., `writable`, `No eXecute (NX)`, `accessed`) to control and monitor the guest like a kernel could a normal process.

However, SVM itself does not protect the guest from a malicious hypervisor in any form. Although VMs are isolated from each other, the hypervisor can access the entire guest address space. The hypervisor can read and modify registers, inject code, and

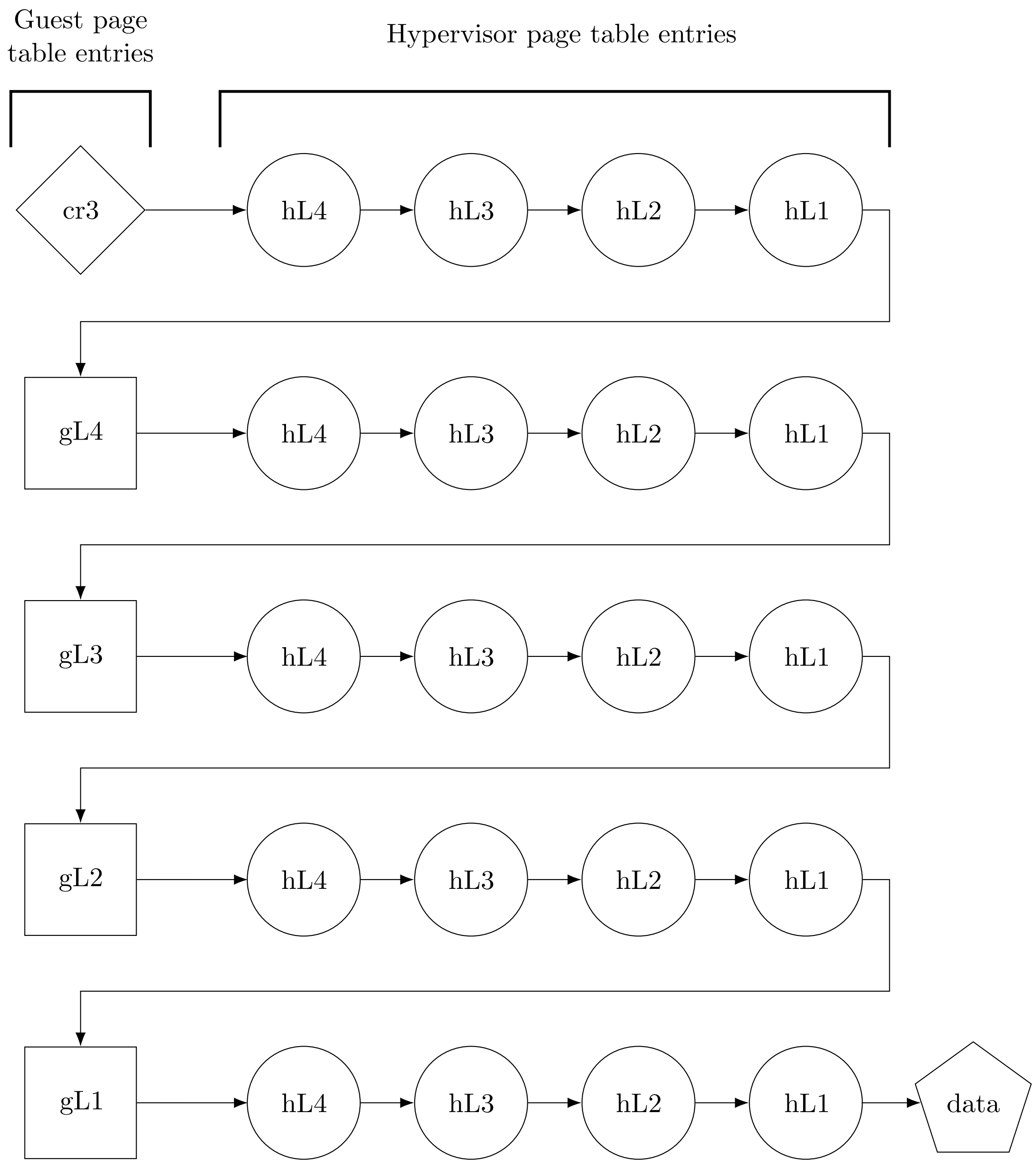


Figure 2.1.: Address translation with nested paging (c.f. [8]).

generally do everything the guest kernel can do in the virtual machine. As a result, a trusted hypervisor is required, which might not always be guaranteed in a cloud hosting scenario.

2.4. Secure Memory Encryption (SME)

Modern server-grade AMD CPUs support the Secure Memory Encryption (SME) feature, which allows the kernel to mark pages as encrypted by setting a special bit within the page table [33]. The pages are then encrypted with AES-128 using an AES key randomly generated on each system boot. SME uses a coprocessor on the chip, the AMD Secure Processor, as a trusted device running signed, trusted AMD firmware to manage keys and run security-critical functions. Encryptions and decryptions are performed in the Memory Management Unit (MMU) using *transparent encryption*. This means the kernel does not need to take any explicit action to encrypt or decrypt the data.

Although this encryption does not protect the memory against access by a malicious operating system, it mitigates data recovery through physical probing of DRAM modules. Additionally, since memory encryption is enabled using the page tables, a simple memory-scanning tool cannot recover unencrypted data. Therefore, while SME cannot fully protect against a malicious machine owner running custom kernels, it can protect against simple memory dumping attacks from a rogue system administrator.

2.5. Secure Virtual Machine (SEV)

AMDs SEV is a step towards protecting the virtual machine from the host. SEV is an extension to the SVM architecture that uses the same memory encryption engine as SME. One crucial feature of SEV is that, unlike SME, it uses multiple encryption keys.

If SEV is enabled, each virtual machine has a private and unique key managed by the Secure Processor. Each key is mapped to a virtual machine using the Address Space ID (ASID). This means that even though the hypervisor can access guest page data, it cannot decrypt it without the cooperation of the Secure Processor (SP). Additionally, the hypervisor cannot trivially perform targeted modifications of the guest data, as any change in the ciphertext of a guest will lead to (ideally) unpredictable changes in the plaintext.

SEV's memory encryption encryption protects confidential information stored in the guest's physical memory from being easily accessed by the hypervisor. Since the data is encrypted on disk and only decrypted when the virtual machine runs, the guest is protected against targeted modifications of code or data pages.

SEV provides an attestation process to allow the guest owner (the customer) to verify the correct virtualization settings and operation of the virtual machine. During the startup process of the virtual machine, the SP checks the status of the processor and the guest virtual machine and creates an attestation report for the guest owner. This report contains data like microcode and firmware versions, hardware identifiers and settings. It is signed by the SP using a processor-specific key, which is verified by an AMD certificate chain. The guest owner can check if the attestation report matches the customer's desired security policy. The SP then provides them with a secure channel to the virtual machine, which they can use to transfer secrets like disk encryption keys. However, if the owner decides not to trust the virtual machine, for example, due to insecure virtualization settings or outdated firmware versions, the virtual machine will not proceed to launch.

2.6. Encrypted State (ES)

The Encrypted State (ES) extension for AMD's SEV introduces encryption for register states. On every `#VMEXIT`, register values are encrypted and saved to a dedicated area in the VMSA. This dramatically reduces the information leakage through registers.

However, the hypervisor cannot inspect registers to determine why a `#VMEXIT` has occurred. This is why SEV-ES introduces a new interrupt type for the guest and a new way of handling some types of interrupts and `#VMEXIT`s:

Whenever the hypervisor requires information from the host to handle a `#VMEXIT` (e.g., nested page faults, MSR accesses or IO operations), a VMM Communication Exception (VC) is triggered in the guest. The guest can write data to the Guest-Host Communication Block (GHCB) to provide the hypervisor with the necessary information. The GHCB is an unencrypted memory area reserved specifically to pass information between the guest and host. In order to trigger the actual `#VMEXIT`, SEV-ES introduces a new instruction called `VMGEXIT`, which allows the VC handler to pass execution to the hypervisor. A flowchart of this process can be seen in Figure 2.2

This approach drastically reduces the attack surface compared to SEV without extensions. However, SEV-ES does not protect against page remapping attacks as described in Section 2.8.2.

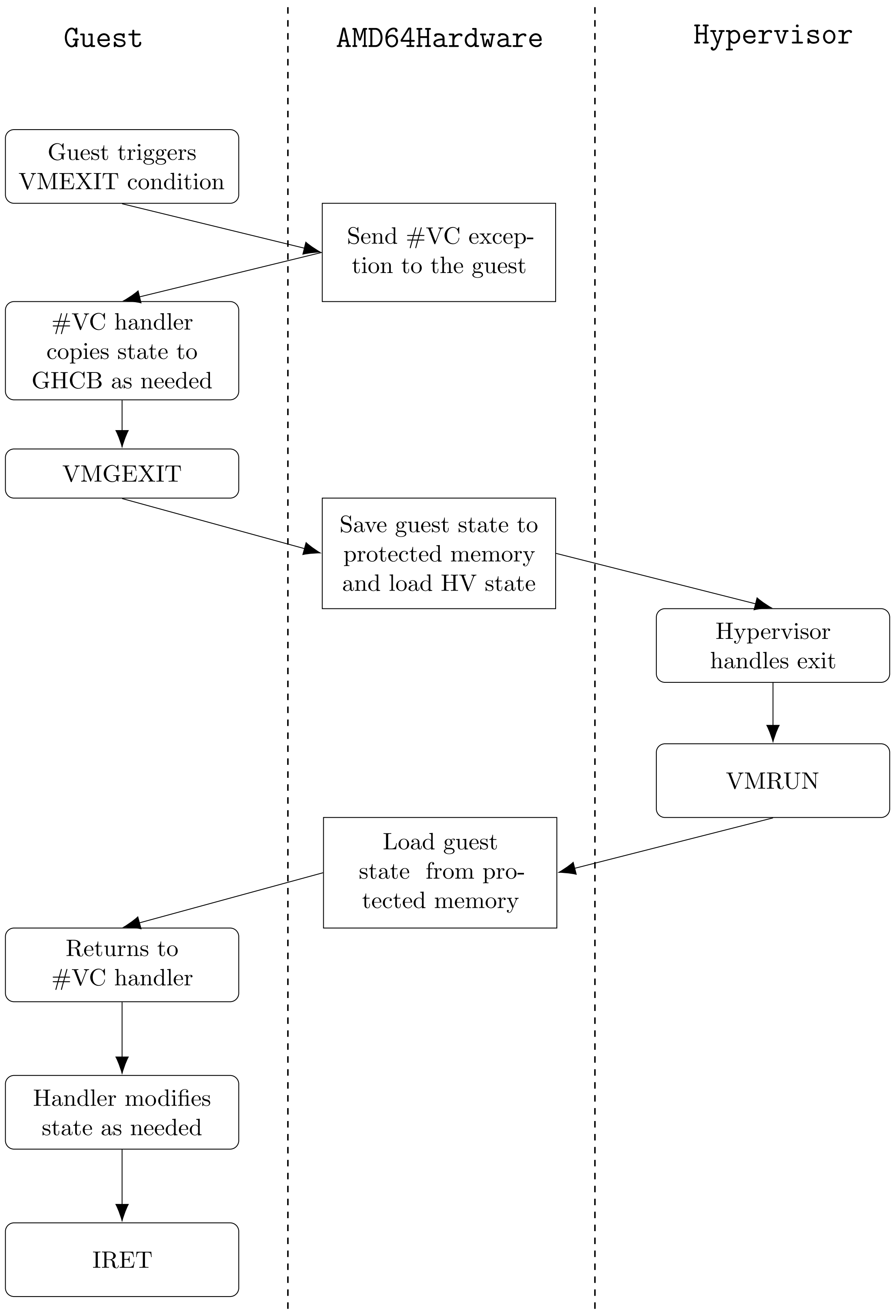

Guest
AMD64Hardware
Hypervisor
Guest triggers VMEXIT condition
Send #VC exception to the guest
#VC handler copies state to GHCB as needed
VMGEXIT
Save guest state to protected memory and load HV state
Hypervisor handles exit
VMRUN
Load guest state from protected memory
Returns to #VC handler
Handler modifies state as needed
IRET

Figure 2.2.: SEV-ES `#VMEXIT` flowchart [7, p. 598].

2.7. Secure Nested Paging (SNP)

AMD introduced the SNP extension as a way to to mitigate issues caused by the fact that the hypervisor can modify mappings in the nPT and write to encrypted guest physical pages.

To achieve this, AMD introduced a new system-wide data structure: the Reverse Map Table (RMP). This table contains information on how specific physical pages can be used. This includes information on whom a page belongs to, where it is supposed to be mapped in guest physical memory, various configuration options, and a `Validated` bit.

When enabled, this structure is initialized during boot in cooperation with the Secure Processor, and imposes additional restrictions during page accesses.

The newly introduced page access checks that may be performed are the following [7, p. 608-609]:

RMP-Covered: Checks that the target page is covered by the RMP. A page is covered by the RMP if its corresponding RMP entry is below `RMP_END`. Any page not covered by the RMP is considered a hypervisor-owned page.

Hypervisor-Owned: Checks that if the target page is covered by the RMP then the `Assigned`-bit of the target page is 0. If the page-table entry that specifies the sPA indicates that the target page size is 2MB, then all RMP entries for the 4KB constituent pages of the target page must have the `Assigned`-bit set to 0. Accesses to 1GB pages only install 2MB TLB entries when SEV-SNP is enabled, therefore, this check treats 1GB accesses as 2MB accesses for purposes of this check.

Guest-Owned: Checks that the ASID field of the RMP entry of the target page matches the ASID of the current VM.

Reverse-Map: Checks that the `Guest_Physical_Address` of the RMP entry of the target page matches the guest physical address of the translation.

Validated: Checks that the `Validated` field of the RMP entry of the target page is 1.

Mutable: Checks that the `Immutable` field of the RMP entry of the target page is 0.

Page-Size: Checks that the following conditions are met: If the nested page table indicates a 2MB or 1GB page size, the `Page_Size` field of the RMP entry of the target page is 1.

If the nested page table indicates a 4KB page size, the `Page_Size` field of the RMP entry of the target page is 0.

Table 2.2.: RMP memory access checks [7, p. 609].

Host/Guest	**SNP Active**	**Type of Access**	**C-Bit**	**Check**	**Fault**
Host	-	Data write, Page Table Access	-	Hypervisor-Owned	`#PF`
Guest	✗	Data write, Page Table Access	-	Hypervisor-Owned	`#VMEXIT(NPF)`
Guest	✓	Instruction Fetch, Page Table Access	-	RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size	`#VMEXIT(NPF)`
				Validated	`#VC`
				VMPL	`#VMEXIT(NPF)`
Guest	✓	Data write	0	Hypervisor-Owned	`#VMEXIT(NPF)`
Guest	✓	Data write, Data read	1	RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size	`#VMEXIT(NPF)`
				Validated	`#VC`
				VMPL	`#VMEXIT(NPF)`

VMPL: Checks that the VMPL permission mask allows access.

According to Table 2.2, any write operation from the hypervisor to a non-hypervisor-owned page leads to a page fault, preventing the host from modifying data in guest physical memory. For guests with SNP enabled, the `Reverse-Map` check prevents the hypervisor from exchanging pages using the nPTs and from mapping the same page to multiple locations. These protections mitigate most memory-targeted exploits from previous SEV versions, as shown in Section 2.8.2.

The `Validated` bit mitigates the possibility of the hypervisor taking over the ownership of a page, changing the page contents, and then returning the page to the guest.

Since the RMP pages are not owned by the hypervisor but by the AMD SP, the hypervisor can only modify RMP entries using the new `RMPUPDATE` instruction. This instruction automatically causes the modified entry to get invalidated. When accessing an invalidated page, the guest receives a VC and can decide how to handle the invalidated page. When the guest accesses a page for the first time, it can validate it using the `PVALIDATE` instruction.

In order to meet the desired integrity of SEV-SNP, the guest VM should never validate memory corresponding to the same GPA more than once [6].

AMD recommends that the guests validate all pages at boot time and refuse to validate pages at any other point in time. Upon an invalid page access, the hypervisor is faulty or malicious, and the guest should react accordingly. While this approach protects against software-side modifications of guest memory, attack vectors modifying data on the RAM chips are not detected. There might be ways to change RMP entries without invalidating them, e.g., fault attacks or special custom hardware. According to AMD, SNP does not protect against attacks on the physical memory, and such attacks are considered out of scope [6].

2.8. Existing Attacks on SEV

There are many attacks on the different versions of SEV. In this section, we highlight some of them and look at the mitigations in place.

2.8.1. Register-Based Attacks

One of the flaws of SEV without extensions is that it does not encrypt or protect register states. Before hardware with support for the technology was even available, Hetzelt and Buhren [31] showed that it is possible to gain arbitrary code execution in the guest by interacting with the guest register state. They do this with a technique similar to return-oriented programming (ROP) [59], the difference being that the gadgets end in a `hlt` instruction instead of `ret`. They can force the guest to jump to the target gadget by editing the stored instruction pointer in the VMSA. By using the `hlt` instruction to force a `#VMEXIT`, the hypervisor can craft a chain of multiple gadgets to execute arbitrary code. This makes it possible to exfiltrate arbitrary data by creating a chain that copies memory contents to an unencrypted memory area. Breaking KASLR to find the correct addresses of the gadgets can be done by tracking the instruction pointer register during faults or exceptions with known exception handler locations.

Werner et al. [76] show that access to register states allows for efficient code identification using single-stepping. By observing the register state changes, they can accurately infer which instructions were executed. This is a possible approach to find relevant code pages for further attacks.

An important change when upgrading to SEV-ES is that the guest state is now encrypted. In addition to preventing the hypervisor from reading register values, this also limits the ability of the hypervisor to redirect control flow. The AMD SP decrypts the register state when loading the guest state. Since the decrypted value is not easily predictable, injecting a specific register value is difficult.

However, even with ES, SEV can still leak information about its execution or even secrets through registers.

Li et al. [42] demonstrate an attack on the encrypted register values of the guest. They abuse the fact that the VMSA is encrypted as independent 16-byte blocks, tweaked based on their physical address. While this means that two different encrypted blocks cannot be compared, the same 16-byte plaintext in the same VMSA block will always result in the same ciphertext.

Using this property, they build a similar attack to Werner et al. [76]. They infer information about the guest's state by observing register changes. Applying a combination of page tracking and single-stepping, they track the execution of the guest. They also distinguish between different guest processes by comparing the ciphertexts of control registers, e.g., `CR3`. Additionally, it is possible to build a dictionary of plaintext-ciphertext pairs by passively collecting combinations during `#VMEXIT`s where the guest state is deliberately shared with the hypervisor or by actively injecting register values, e.g., using IO operations. By strategically choosing the dictionary values, it is possible to attack constant time cryptosystems, which they show by successfully breaking RSA and ECDSA implementations in OpenSSL.

To mitigate this issue, AMD released a feature called *VMSA Register Protection* [4]. When enabled, this mitigation will obfuscate register values using a pseudo-random value. The patch breaks the direct mapping between ciphertext and plaintext. Furthermore, it changes the ciphertext of affected registers on every `#VMEXIT`, preventing attackers from obtaining information when tracking those registers via the VMSA. The feature is only available on AMD CPUs supporting SEV-SNP.

As of writing this thesis, since the release of this feature, no new register-based attacks on SEV have been published.

	Mitigated in SEV Version		
Attack	SEV	SEV-ES	SEV-SNP
Arbitrary control flow redirection [31]	✗	✓	✓
Data leak through VMSA registers [76]	✗	✓	✓
Fingerprinting through encrypted registers [76]	✗	✗	✓*
Data leaks via predictable register encryption [42]	✗	✗	✓*

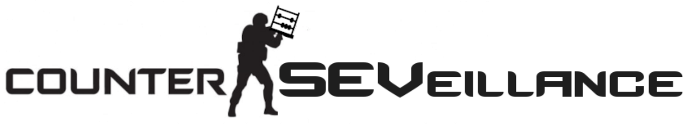
*With VMSA Register protection feature

Table 2.3.: Overview of register-based attacks on SEV.

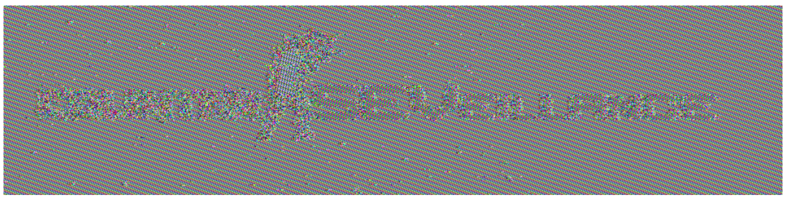
2.8.2. Memory-based Attacks

AMD uses AES to transparently encrypt memory pages in the MMU. To minimize the performance penalty, they cannot use AES schemes that reuse data across ciphertext blocks. As a result, this leaves only Electronic Code Books (ECB) as a feasible encryption scheme. ECB has the well-known flaw that the same plaintexts will always be encrypted to the same ciphertext, making it possible to distinguish patterns in the ciphertext (see Figure 2.3). To mitigate this, AMD tweaks the encryption based on the host physical address of the data.

In addition to their attack on guest registers, Hetzelt and Buhren [31] show that, even with these tweaks, the encrypted state of a specific address stays the same. As a result,



(a) Original Image.



(b) ECB encrypted image.

Figure 2.3.: Effect of ECB on a plaintext image.

the hypervisor can replay the guest state. Because the hypervisor manages the nPT, it can remap guest physical pages to any host physical page, thusm controlling the tweak of the encryption. An attacker could collect encrypted data on a specific host physical page by mapping it to an attacker-controlled guest physical page. Later, the hypervisor can map the same host physical page to another guest physical page, overriding the original data with the previously collected, attacker-controlled page content [23]. One of the ways for the hypervisor to control the contents of a page is by using IO operations. Since the hypervisor controls all IO ports, it can use them to send arbitrary data to the guest. The guest must store the received data somewhere in its memory, which creates a page filled with attacker-controlled data.

On the other hand, Morbitzer et al. [51] show that a similar principle can be used to leak data from the guest's memory if the guest machine runs any service that sends data from memory (e.g., a website). By remapping pages, the hypervisor can force the service to send arbitrary data from the guest's physical address space. This is possible by switching the page containing the response data with the target page. With this attack, the hypervisor can leak the entire main memory contents of the guest machine.

While IO operations usually have side effects that the VM can detect, Li, Zhang, and Lin [40] found a way to exfiltrate some data when the virtual machine is already shut down. During the boot process of a virtual machine, the hypervisor needs to cooperate with the SP to set up the initial memory contents of the guest, generating an encryption key that is tied to a specific ASID. According to AMD, executing another virtual machine with a reused ASID would lead to a crash since its memory contents were not initialized and encrypted with this reused key. Decrypting them with it results in invalid data, which means that gaining code execution in that address space would be prevented.

However, the authors found a way to leverage page table walks to exfiltrate data to the hypervisor dubbed *CrossLine*. Since the hypervisor has complete control over the nPTs, it can follow the page table walks of the guest. CrossLine exploits this behavior by manipulating the guests instruction pointer in the VMSA. Through this, they can reuse an ASID with a malicious virtual machine to leak the entire guest paging structure. With this knowledge, they modify the attacker virtual machine's nPT in a way that allows them to execute encrypted instructions in the victim's address space. By executing an instruction that moves data from memory to a register, they can leak the entire memory contents.

Wilke et al. [77] were able to exploit the XOR-Encrypt-XOR encryption mode used in SEV to reuse blocks of encrypted memory in other parts of the guest address space. SEV-ES uses the following functions to encrypt and decrypt data:

$$\mathrm{Enc}_K(m, p) := \mathrm{AES}_K(m \oplus T(p)) \oplus T(p)$$

$$\mathrm{Dec}_K(c, p) := \mathrm{AES}_K^{-1}(c \oplus T(p)) \oplus T(p)$$

p is the host physical address of the data, and $T(p)$ is a tweak function that is used for encryption. Considering a full flow of encryption and decryption:

$$m' = \mathrm{Dec}_K(\mathrm{Enc}_K(m, p), q) = \mathrm{AES}_K^{-}1(\mathrm{AES}_K(m \oplus T(p)) \oplus T(p) \oplus T(q)) \oplus T(q)$$

If p and q were equal, m and m' would be the same. If the attacker had moved the block to another location in memory (and therefore changed q), the result would be changed.

The attack targets the insecure tweak function T. If an attacker knows $T(p) \oplus T(q)$, they can take this term into account when moving the data:

$$m' = \mathrm{Dec}_K(\mathrm{Enc}_K(m, p), q) = \mathrm{AES}_K^{-}1(\mathrm{AES}_K(m \oplus T(p)) \oplus T(p) \oplus T(q) \oplus T(p) \oplus T(q)) \oplus T(q)$$

$$m' = \mathrm{Dec}_K(\mathrm{Enc}_K(m, p), q) = \mathrm{AES}_K^{-}1(\mathrm{AES}_K(m \oplus T(p))) \oplus T(q) = m \oplus T(p) \oplus T(q)$$

Since the values of T can be determined, the attacker can find parts of known memory contents m at addresses p, such that the resulting $m \oplus T(p) \oplus T(q)$ is a useful gadget to attack the target, for example redirecting control flow. This way, they can gain control over the victim without any IO operations.

Li et al. [43] found that forcing a victim process to execute chosen instructions is possible using Translation Lookaside Buffer (TLB) poisoning. This attack requires a hypervisor-controlled unprivileged process to run inside the virtual machine. They abuse the fact that TLB management is handled by the hypervisor. They interrupt the victim when it is about to execute an instruction at virtual address V_0. Then, they schedule another vCPU running the attacker process and clear the TLB. They instruct the process to run the chosen instruction at the address V_0, poisoning the TLB entry for that virtual address with the corresponding physical address. Afterward, the hypervisor schedules the victim process again without clearing the TLB. The victim process will execute the instruction injected into the TLB. The same primitive method is also usable for injecting data from the victim into attacker processes. The authors show an attack that injects the correct password hash from a legitimate SSH login process into a victim, bypassing the password check. According to the paper, AMD stated that this attack is mitigated in SEV-SNP. However, they do not mention how this mitigation is implemented.

When preparing the guest's boot, the hypervisor collaborates with the AMD-SP to load the initial binary into the encrypted guest memory. The hypervisor can upload the binary block by block using the `LAUNCH_UPDATE_DATA` command with the SP. The guest owner can verify the integrity of the initial data during the attestation process using measurements provided by the SP. Wilke et al. [79] found that these measurements did not change when they modified the order of the blocks. The uploaded blocks can be as small as 16 bytes. By reordering the blocks in specific ways, they were able to construct a ROP chain to execute arbitrary instructions. This allowed them to leak the secrets sent by the guest owner after the successful attestation. SEV-SNP mitigates this issue by increasing the minimum block size to 4KB and incorporating the block size in the measurement value.

Li et al. [39] take the idea of *CipherLeaks* [42], deriving information from encrypted memory blocks, and applies it to the entire memory space of the virtual machine instead of only the VMSA. Since *VMSA Register Protection* only tweaks the VMSA before `#VMEXIT`s, they can mount similar attacks as presented in CipherLeaks even with the feature enabled.

The most recent attack on SEV at the time of writing is *CacheWarp* [82]. CacheWarp exploits a software-based fault attack in the cache of AMD CPUs. Specifically, they invalidate the processor's internal caches by using the `invd` instruction. In contrast to the `wbinvd` instruction, `invd` does not write back the cache contents before invalidating. This incoherence leads to dropped write operations, resulting in reverted memory contents. Using single-stepping (see Chapter 4), they can precisely control which write operations to drop. One potential attack target is authentication: For example, when exploiting the `sudo` binary, they drop the write to a field in a struct, which should contain the user ID. Since the struct is zero-initialized, and the root user on Linux systems always has the user ID 0, `sudo` thinks it was launched as root and does not require further authentication. The attack works not only on explicit write operations but also on implicit writes in the microcode, such as the return address written to the stack during a `call` instruction.

Since this attack does not touch the guest memory directly and works by exploiting a flaw in the cache implementation, it is the only memory-based attack known to exploit SEV-SNP systems at the time.

2.8.3. Other attacks

Other attacks on SEV do not target guest registers or memory. One such attack is *PwrLeak* [73]. It uses the processor's built-in power consumption measurement functionality purely from software to gain insight into the guest. By measuring the power consumption of instructions, they can infer which instructions are executed by the guest and, in some cases, even obtain information about the operands. They use these primitives to leak JPEG files and parts of an RSA private key during processing. Since the power consumption differences of such a measurement are very small, they employ two different amplification strategies:

Table 2.4.: Overview of memory based attacks on SEV

Attack	Mitigated in SEV Version		
	SEV	SEV-ES	SEV-SNP
Code injection via page switching [23]	✗	✗	✓
Data extraction via page switching [51]	✗	✗	✓
CrossLine [40]	✗	✗*	✓
Code injection via predictable encryption tweaks [77]	✗	✗	✓
TLB Poisoning [43]	✗	✗	✓
Data extraction via predictable memory encryption [39]	✗	✗	✗
Data corruption via dropped cache writebacks [82]	✗	✗	✗

*Limited impact due to register encryption

Emulation-based amplification repeatedly emulates the target instructions to amplify the power consumption (and, therefore, the difference in the consumption of other instructions). This strategy only works on SEV without extensions since it requires access to guest registers. However, it is the more reliable of the two strategies

The other strategy is interrupt-based amplification, which uses strategic interrupts to force the target instruction into a time window where power measurement facilities are updated. The authors of the attack claim that this less reliable strategy should also work on ES and SNP. They hypothesize that the extra protections (and associated computations and checks) would introduce more noise into the measurement, making the side channel even less reliable than it is currently.

On the other side of power-based attacks, Buhren et al. [12] leveraged power glitching to force the AMD-SP to accept their public key and successfully verify and boot their self-signed payload. This way, they could gain code execution on the SP. With the ability to run custom code, they discuss multiple possibilities to exploit a guest:

By extracting chip-specific secrets, it is possible to forge signed data in an attestation process. It is also possible to change settings after the attestation has finished, like enabling debug functionality. The debug functionality can usually only be enabled if the guest allows it and allows the hypervisor to decrypt guest pages. Lastly, they could dump the firmware of the SP. By reverse-engineering the algorithms used for key generation, they can obtain complete knowledge of a specific CPU's secrets. This compromises the security of all guests running on that specific processor.

2.9. SEV-Step

Wilke et al. [78] introduced a single-stepping framework called *SEV-Step*, which provides similar functionality for SEV as *SGX-Step* does for Intels SGX [70]. The framework consists of a modified Linux kernel with additional features added to the `kvm_amd` module and libraries, allowing easy interaction with the kernel module from userspace. In addition to single-stepping, the framework provides page tracking functionality that can be used interactively from userspace.

2.9.1. Communication between kernel- and userspace

Implementing the final exploit in userspace and minimizing changes in kernel space has several advantages.

Compiling and starting a new userspace program is generally faster and more straightforward than compiling a new kernel module. Significant changes may require a reboot to load the new code, as a simple reload of the kernel module may not suffice. A program in user space is compiled and linked with the SEV-Step userspace library to produce a self-contained binary that can be executed immediately.

In contrast, a crash due to a coding error in the kernel is likely to cause a system crash, or at least require a reboot. Keeping all experiments in the userspace means that a crash of the exploit may lead to a stuck virtual machine, which is still recoverable without restarting the machine. This is particularly advantageous when developing exploit prototypes rapidly, as it reduces the need for reboots.

By definition, the interactive nature of *SEV-Step* requires more communication between kernel- and userspace. The userspace library uses the `ioctl` syscall to communicate with the kernel module. This interface configures the kernel's single-stepping and page-tracking behavior and initializes the shared memory buffer used for the rest of the communication. The shared memory buffer transfers data from kernel- to user space. The communication is synchronized using a spinlock on the shared memory buffer.

To guarantee that the kernel halts execution until the userspace has acknowledged and processed an event, SEV-Step uses an ACK mechanism. Whenever an event occurs, the kernel will acquire the spinlock on the shared memory region and write data corresponding to the event to the appropriate locations. When it has finished, it will clear the `event_acked` flag in the shared structure and set the `have_event` flag. Before continuing, the kernel will wait for `event_acked` to be set. The user space will do so after it has finished handling the event. The userspace library will also reset the `has_event` flag and wait for the kernel to set it again, notifying the user space of a new event.

2.9.2. Supported functionality

The SEV-Step API provides similar functionality to SGX-Step. Since the hypervisor controls and can freely modify the nPT, it can leverage the access bits (`present`, `writable`, `NX`) to notify the attacker whenever the target accesses a specific memory location in the chosen manner. Similar to Li et al. [41], attackers can use this functionality to obtain information about running programs and use it much like a breakpoint in a traditional debugger, but with coarse, page-by-page resolution. This technique allows the victim program to run at normal speed up to the target code section, where it is interrupted and the attacker can interactively single-step the code.

The main feature of SEV-Step is the single-stepping of SEV VMs. It can do this interactively and gives the attacker detailed information about each step via the userspace API. The information returned by default includes the number of steps taken and the time it took the virtual machine to exit.

Chapter 3.
Attack Primitives

In this chapter, we go into the fundamental building blocks of `CRACKPIPE`. We show the individual parts required for the exploit to work, providing a high-level overview of the implementation.

3.1. Page-Fault Tracking

Page faults are exceptions triggered when memory is accessed in a way that is not allowed for a specific memory region. Access permissions are configured in the page tables. Page faults are relatively common and are used to implement mechanisms like on-demand page mapping or copy-on-write. In a regular system, they are handled by the kernel. Attackers operating with kernel privileges have permission to access any data on the system. Thus, in the traditional model, memory accesses cannot leak any data that the attacker cannot access already.

However, when operating in TEEs, page faults become relevant side channels. A central principle of trusted execution is protecting secrets even when a malicious attacker controls the rest of the system. However, both SGX and SEV are subsystems of a running system, where they do not run at the highest permission level. In the case of SGX, the kernel is in charge of managing system resources, while on SEV, the hypervisor manages the hardware.

In SEV, the hypervisor can configure the access permissions on a page-by-page basis using the nested page tables. This is required since the hypervisor must be able to allocate pages to the guest physical address space dynamically. However, a malicious hypervisor can use these access permissions to track the execution of specific programs or the entire operating system.

Unsetting the `present`-bit on every page, every memory access on any page will cause a `#VMEXIT` with a nested page fault, informing the hypervisor on which page was accessed. The hypervisor can determine access patterns with page size granularity by resetting the `present`-bit for a page until the subsequent page fault occurs. While this technique is very slow, it allows an attacker to fingerprint the running software in a virtual machine and find the required guest physical page numbers for subsequent attacks.

The attacker can start with more selective attacks when the required pages are known. Depending on the attack target, the attacker may only be interested in write operations, meaning they can remove the `writable`-permission on the page. In the case of a secret-

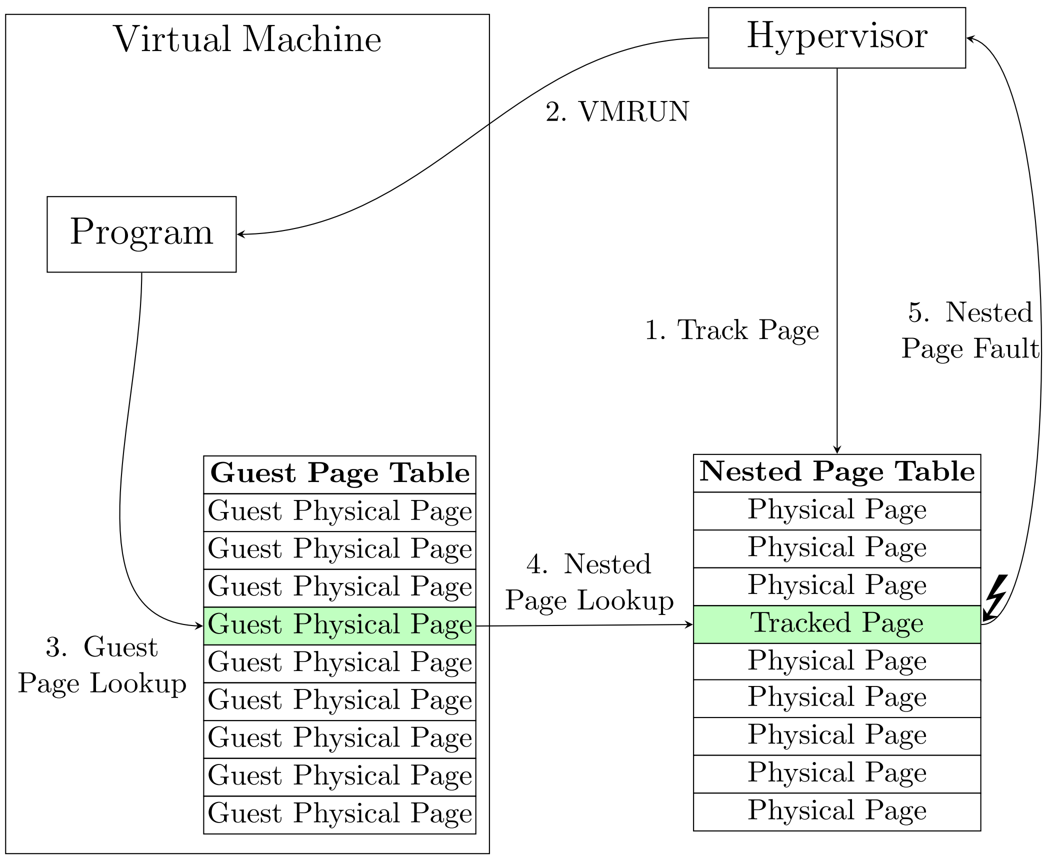


Figure 3.1.: Page-fault tracking overview. The tracked nested page causes a nested page fault on access, which triggers a `#VMEXIT`.

dependent data access, they might only turn off the `present`-bit for specific pages to recover the secret by observing the access patterns.

Another strategy is turning off the NX-bit, which means the attacker will receive an interrupt when the guest tries to execute code from the target page [74]. These interrupts can be used to find the entry point to a relevant part of the code without slowing down the system until the target code is about to be executed. The attacker can then switch to other attack primitives, like single-stepping, to have more fine-grained control over the execution.

3.2. Single-Stepping

Single-stepping is the process of pausing the execution of a program after each instruction. It is essential in software development, to trace the execution of any piece of code and pinpoint the exact location where something goes wrong. Modern processors have

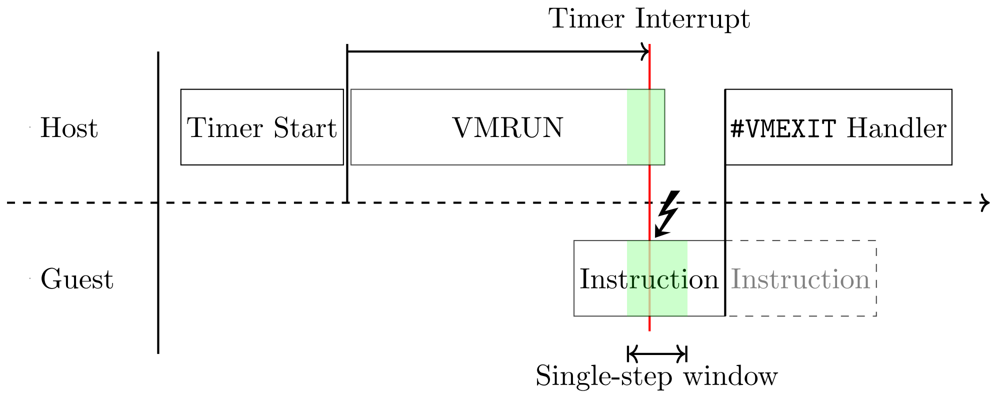


Figure 3.2.: Visualization of a single-step using the APIC timer. Timer intervals are chosen such that the interrupt hits the single-step window at the end of `VMRUN`.

facilities that provide the kernel with the required functionality for efficient debugging. In the case of Linux, the kernel provides the `ptrace` interface to userspace, allowing user processes to debug each other with features like single-stepping, breakpoints, register modification, or memory interaction. However, the kernel imposes some constraints on which processes can be debugged by which user. Access control to debugging is essential since it would lead to grave security vulnerabilities if any user could debug any process.

Since hypervisors are also software that needs to be programmed and debugged, the same facilities can be used there. It is possible to cause the guest to trap after a single instruction. The platform provides breakpoint registers, which cause a trap once the execution reaches the chosen memory location. It is also possible to send a request to the Platform Security Processor (PSP) to ask it to decrypt pieces of memory for debugging purposes.

Obviously, allowing the hypervisor to read memory and registers would completely undermine the security of SEV. To mitigate this, guest owners can turn off debugging for their virtual machine by zeroing the `DEBUG` bit in the guest policy. This disables the use of any builtin debug functionality and disallows the use of the `SEV_CMD_DBG_DECRYPT` and `SEV_CMD_DBG_ENCRYPT` commands, preventing the hypervisor from decrypting memory contents.

However, it is possible to implement a form of single-stepping (albeit less reliable) by trying to interrupt the guest shortly after allowing it to start, only allowing a single instruction to finish.

3.3. Timing Analysis

The duration of operations often leaks information about the CPU's internal state, the operation's input data, or the operation itself. As Wilke et al. [78] showed, a Nemesis-style timing attack [69] is possible under SEV-SNP. The interrupt latency can be used to distinguish different types of instructions. AMD states that preventing application fingerprinting is out of scope for SEV. However, interrupt latencies can be used to verify a function's current position while single-stepping. Measuring latency incurs very little performance overhead when single-stepping. Therefore, the measurements can confirm that the executed code matches the expectations.

3.4. Performance Counters

Performance counters are a way for the CPU to track various performance-related statistics. They can be used to determine application bottlenecks, identify resource-hogging applications or functions, and even detect some types of attacks.

Some examples of the tracked information are the number of retired instructions per core, current CPU clock speed, number of retired instructions for some specific types (like Advanced Vector Extensions (AVX)), cache events (misses and flushes), and more. Performance counters are only accessible by the kernel. However, the kernel can pass performance information to the userspace for users with the correct privileges.

Contrary to Intel's SGX, performance counters are not automatically disabled in SEV. Since SEV runs a full virtual machine as a TEE instead of only a small enclave, it makes sense to allow the hypervisor at least some insight into which kinds of operations the virtual machine is performing. Since the technology is developed with cloud computing in mind, the hypervisor has to be able to detect if a secure virtual machine is hogging shared resources or performing some attack on other guests or the host itself.

According to AMD, preventing application fingerprinting via performance counters is not in scope for SEV since code is usually not confidential, but data is. In the upcoming chapters, we demonstrate that this security assessment for performance counters is inaccurate and performance counters can actually leak information about data inside SEV as well.

Chapter 4.

Implementation

In this chapter, we detail how we implemented our attack primitives. We also show how our attack combines these primitives to leak data from a victim virtual machine.

4.1. Single-Stepping

The guest owner can turn off debugging functionality in a properly configured SEV environment. Therefore, we develop a custom implementation of single-stepping.

The APIC provides programmable interrupt support to the operating system. This means that the operating system can enable and configure a number of different interrupt types which will trigger whenever the corresponding event occurs. Such events include input/output operations, timers, performance counters, and thermal management. The relevant interrupt type for our application is the *APIC timer*. The APIC timer is a software-configurable countdown timer. The operating system can set it up by using multiple control registers.

The kernel chooses a divider in the *Divide Control Register* to control the frequency of the timer. A smaller divider means better resolution but leads to less maximum range since the timer value decreases faster. Since we only want to start the timer before entering the guest, we configure the timer in *one-shot mode*, which causes it not to restart automatically. We move our desired value into the *Current Count Register* to start the timer. The APIC then starts decrementing the counter at the configured interval and trigger an interrupt when it reaches 0. We configure the settings by writing them to the memory-mapped APIC configuration area.

By configuring the timer to interrupt the core at a specific point after the `VMRUN` instruction, we can to single-step the guest. The specific timing depends on factors like the current clock speed, CPU utilization, kernel- and hardware settings, and differences in the silicon. Since the hypervisor controls the system, we can limit the impact of those factors by fixing our clock, reserving an entire core for the target virtual machine, and choosing the ideal kernel- and hardware parameters.

The attacker can profile the hardware in advance to account for potential differences between processors. The timer value required for consistent single-stepping during our experiments remained constant.

We configure and start the timer as late as possible before executing `VMRUN` to minimize latency variances caused by other instructions. We also prepare as much as possible beforehand to minimize the timing overhead. We pass the prepared values into the function as arguments. A nice side effect of this approach is the ability to use standard Linux kernel functions and macros for most of our implementation, for example, to determine the address of required APIC registers.

The default SEV-ES and SNP entry code, shown in Listing A.1, starts with a standard function prologue, pushing callee-saved registers to the stack. All `push` instructions in the function prologue (lines 6 to 16) can potentially have some variable timing to them, for example, due to full internal CPU buffers or other congestion. Therefore, we want to start the timer as late as possible.

If we only want to implement single-stepping without additional features, the ideal place would be line 20, which is just before we enable interrupts and execute `VMRUN`. We precompute the address for the current count register and pass it as an argument to the function, along with our desired timer value. We then start the timer by writing our desired timer value into the *Current Count Register*. The kernel can run the virtual machine without single-stepping by simply passing 0 as the desired timer value since this turns off the timer without an interrupt.

In the case of base SEV, most register values need to be manually saved and loaded by the host during the context switch. The entry function for SEV is significantly longer than the SEV-ES version. This makes implementing a reliable form of single-stepping more challenging since the host has to overwrite its registers with the guest's saved values before the context switch. This would overwrite the precomputed values passed to the function. Therefore, we instead start the timer before restoring all guest register values. These memory accesses cause more variance in the delays between starting the timer and entering the guest, which reduces the reliability of our implementation.

To mitigate this uncertainty, we start with a lower-than-necessary timer value and retry single-stepping the guest multiple times before increasing the value. We repeat this process until we detect a single step. However, this "automatic" search for a valid timer value, starting with a too-small value often causes a "runaway", i.e. the timer interrupt is triggered before the `VMRUN` instruction starts. Since the code executes `STI`, enabling interrupts, before `VMRUN`, it handles the timer interrupt before entering the guest. This causes the virtual machine to execute freely until the next `#VMEXIT` occurs.

To prevent this, we had the idea to overwrite reserved SBZ (should-be-zero) bits in the currently active VMCB, in the interrupt handler. If the interrupt triggers before `VMRUN`, the instruction fails due to an invalid VMCB. The code always resets the bits to zero before trying to enter the guest, restoring a valid state. However, to prevent unnecessary modifications to the kernel code, we opted to provide more precise starting timer values instead of addressing the error.

Starting with SEV-ES, the virtual machine entry code is more concise, since the `VMRUN` instruction manages the guest register state. Therefore, achieving a basic form of single-stepping is easier with ES enabled.

When experimenting with the timer values, we discovered that the required timer value to single-step does not change based on the next instruction. For example, a heavy `CPUID` instruction required the same timer value as a simple `NOP`. We believe that there may be two potential causes for this behavior. Firstly, when a timer interrupt happens, the CPU may still try to process all instructions in the reorder buffer before performing a `#VMEXIT`. Secondly, there might be a window during the execution of `VMRUN` specifically, after which the next instruction cannot be interrupted, and will finish even though a timer interrupt has happened. Performance optimizations may cause this behavior if the first guest instruction is allowed to execute while the `VMRUN` instruction is unfinished.

The second hypothesis is the most likely due to multiple reasons. First, we see distinct steps for combinations of `test` and conditional jumps when single-stepping. Usually, these combinations get fused to one macro-op, so we expect only one step for both of them [25]. Second, the reliability of single-stepping `NOP` instructions seems too high for an interrupt window with just the length of the `NOP`. Third, when single-stepping memory accesses, we can only detect latency differences between cached and non-cached memory locations if we place `mfence`s between the memory accesses. This behavior means that the CPU already fetches the data for future instructions that are not executed before the `#VMEXIT`.

Our first proof-of-concept was developed on SEV, without the ES extension. We monitored the `RIP` field in the VMSA to distinguish between single- and zero-steps. Even with an encrypted VMSA, the ciphertext only changes if the underlying register value is updated. Therefore, this allows us to detect changes in the instruction pointer even on SEV-ES. However, with the *VMSA Register Protection* feature, this assumption does hold. Alternatively, it may possible to employ the same strategy as `SGX-Step` and monitor the `accessed`-bits of the nPT to detect whether instructions were fetched.

In parallel to our work, Wilke et al. [78] released their single-stepping framework *SEV-Step*. They use the same basic principle of APIC interrupts to single-step guests. However, they use the *Retired Instructions* performance counter to distinguish between zero- and single-steps. This method works even with *VMSA Register Protection* enabled and reports the exact number of instructions executed in a single `VMRUN`. Additionally, they implemented features like page tracking and a userspace library for interactive single-stepping and debugging. For SEV-SNP, we rely on their framework for single-stepping.

SEV-Step follows the same principle as our implementation. They achieve a reliable single-stepping behavior by placing the timer's start close to the `VMRUN` instruction. However, their latest framework version also includes latency tracking used for their *Nemesis* [69] experiments on AMD [78]. This timing analysis code uses the `rdpru` instruction to obtain a high-resolution timestamp, and introduces additional delay in form of some register arithmetic operations to compute the correct timestamp and a `PUSH` to the stack. The single-stepping timing is not significantly affected by these operations. In contrast, the noise in the timing measurements from an APIC write may be more significant.

4.2. Performance Counters

Performance counters are a feature provided by the CPU so that the operating system can monitor the use of certain CPU functions. Kernel-level code can select multiple

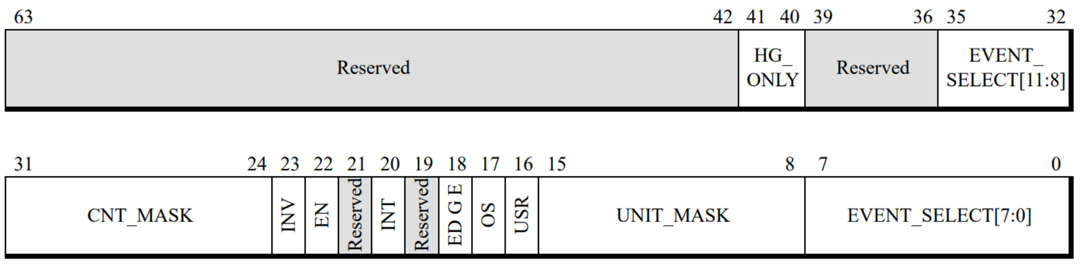


Figure 4.1.: `PerfEvtSel` register layout [7].

performance-related events via one of many `PerfEvtSel` MSR. This will cause the CPU to count the occurrence of the selected event in the corresponding `PerfCnt` MSR. The specific events that can be tracked depend on the processor family and can be found in the manual for the specific processor family [5].

The layout and available fields of the `PerfEvtSel` register can be seen in Figure 4.1. To use a performance counter to track a virtual machine, we set the `HG_ONLY` field to `0x1`, which makes the performance counter only track events while inside the guest and ignore events in the hypervisor. For our attack, we do not need most of the provided features. Configuring the counter to only track user- or kernel-space events could be helpful for some attacks. However, since the hypervisor controls any hardware interrupts the guest receives, it can avoid any jumps into kernel space not caused by the executed code, rendering the filter redundant. At last, we enable the counter by setting the `EN`-bit to 1.

Table 4.1.: `PerfEvtSel` register field description [7].

Bits	**Mnemonic**	**Description**	**Access Type**
63:42	Reserved		RAZ (Reads as Zero)
41:40	HG_ONLY	Host/Guest Only	R/W
39:36	Reserved		RAZ
35:32	EVENT_SELECT[11:8]	Event select bits 11:8	R/W
31:24	CNT_MASK	Counter Mask	R/W
23	INV	Invert Comparison	R/W
22	EN	Counter Enable	R/W
21	Reserved		RAZ
20	INT	Interrupt Enable	R/W
19	Reserved		RAZ
18	EDGE	Edge Detect	R/W
17	OS	Operating-System Mode	R/W
16	USR	User Mode	R/W
15:08	UNIT_MASK	Unit Mask	R/W
07:00	EVENT_SELECT[7:0]	Event select bits 7:0	R/W

SEV-Step by default uses event `0x0C0`, *Retired Instructions*, to track how many instructions have been executed during one `VMRUN`. This works by computing the difference between the instruction count before and after the `VMRUN`. For our attack, we added tracking for event `0xC2` (*Retired Branch Instructions*), and `0xC4` (*Retired Taken Branch Instructions*). On every single step, the difference between the count before and after the `VMRUN` is sent to the userspace for evaluation via SEV-Step's shared memory buffer. While the *Retired Conditional Branch Instructions* event exists and would ignore unconditional branch instructions like `JMP`, `CALL`, and `RET`, these known taken branches can be used for orientation in the code.

In the future, a similar attack using other performance events could use leaks that do not depend on conditional branches. Candidates for such events are *Div Cycles Busy Count* combined with *Div Op Count*, which would leak the order of magnitude of the result of the division. In addition, performance counters that track cache events could be used to side-channel data across cores.

4.3. The `CRACKPIPE` Attack

`CRACKPIPE` reveals secret information if the victim uses the secret data for branching decisions. The attack consists of two phases. In the first phase, our attacker program uses our extended version of SEV-Step to generate a trace of the victim program. The trace includes information about all branch outcomes in the single-stepped section. In the second phase, combining this data with knowledge of the executed code can potentially recover data used for branching decisions. This second phase is completely offline and requires only a single trace.

4.3.1. Gathering Traces

Since we use the framework provided by Wilke et al. [78], the general format of the trace gathering tool is inspired by the examples in their public repository.

According to Li et al. [41], it is possible to identify pages in memory by observing page access patterns. We argue that we can confidently determine the correct pages when combining their strategy with monitoring performance counters, single-stepping, and latency analysis. Therefore, we assume we know the guest physical addresses of our attack target.

We need to know all guest physical pages of the code we target for our attack. We can single-step a whole function or only parts of it, with a page-size granularity. To optimize the performance of our attack, we want to minimize the single-stepped code. To know when to start and stop single-stepping the program, we need to know a start- and stop page. When single-stepping an entire function, the start- and stop pages are identical since the target function will return to the code where it was called from when it finishes.

To start the attack, we begin by tracking execute accesses (*execute-tracking*) on the start page. Unless the victim executes code from this page, this is undetectable and has no performance impact. When the guest execution hits the start page, we track our target pages instead. We start single-stepping as soon as the guest jumps to a target page.

To optimize the performance of our attack, we want to limit single-stepping the guest to a minimum. When we enter the target function, we start *execute-tracking* all guest pages except our target pages. We continue single-stepping while recording performance counters and step sizes. Whenever we receive a page fault on another page than our target, we stop single-stepping, untrack all pages, and start execute-tracking solely the target pages. This usually happens when the tracked code calls a function. Therefore, we record this event as a function call in our data. This method allows the rest of the program to run without any performance impact while minimizing the amount of unnecessary data recorded.

When the called function returns, we receive a page fault on our target page. When this happens, we start single-stepping again while recording a function return event in our data. We also track all other pages again to prepare for the following function call. We know our target code has finished when we receive a page fault on the stop page. We stop single-stepping and turn off all page-tracking completely, allowing the guest to continue running normally. We save the generated trace, which consists of single-step, multistep, function call, and function return events, for later analysis.

We show a graphic representation of this process in Figure 4.2.

4.3.2. Recovering secrets

Once the data collection is complete, we evaluate our results offline. Our attack program creates a JSON file containing an array of events. Each event is of one of four types: `enter_victimpage_pagefault`, `functioncall_pagefault`, `singlestep` or `multistep`. Depending on the event type, they contain different fields: `singlestep` and `multistep` events have a `counted_instruction` field, which counts the number of instructions executed within a single `VMRUN` (always 1 in case of `singlestep`). Step events also contain the performance counter changes in the corresponding step as the fields `retired_branches_perfct` and `branch_taken_perfct`. Pagefault events are primarily present for orientation since knowing when the program has called a function is helpful.

We need the program binary to evaluate the trace since we need the exact instruction pattern. Different compiler versions or optimization levels might cause changes, making the program flow reconstruction impossible. Evaluation of the trace is not automatic since it requires tailoring the code to the specific application. Creating an automated tool for this purpose might be possible using modern disassembly tools' scripting capabilities.

By comparing the values of `retired_branches_perfct` and `branch_taken_perfct`, we can infer a lot of information about the execution. The execution of unconditional branches (e.g., `CALL`, `RET`, and `JMP`) will cause both performance counters to increment by one. We can use this fact to have some orientation in the code since the branch instructions provide a fixed pattern in the data. We can also combine this with the page fault data to have reliable information about our current location in the execution.

By leveraging this knowledge to navigate the instructions, we can leak the outcome of any conditional branch instruction in a single trace. Since we know which instruction we

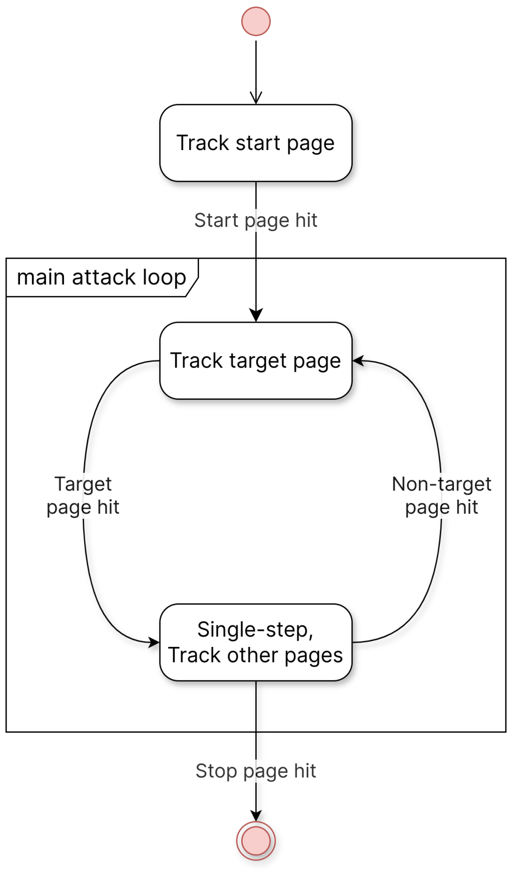


Figure 4.2.: Diagram of the attack phases. We first track only the target page to wait for the call to our target. While single-stepping, we track all other pages to detect calls to other functions. We disable single-stepping during function calls to minimize the attack runtime.

are executing, we can find the single-step event corresponding to a conditional branch instruction and check the `branch_taken_perfct` field. If it is 1, we know the condition it depended on was `true`. If it is 0, we know the condition must have been false. We can ensure that `retired_branches_perfct` is 1 to double-check that we are looking at the correct instruction.

We created a custom script for *Ghidra* [1] to simplify this process of following the trace through the program code. The script prompts for a trace file, the address of the first single-stepped instruction, and a list of addresses to trace. By following the control flow using Ghidra's scripting API and using the single-step data to decide which branch to take for conditional branches, we can easily reconstruct the control flow of the traced program. The script outputs the branching conditions at the specified addresses to the console output, facilitating for easy further analysis.

However, single-stepping with SEV-Step is not completely reliable. In our tests, there was a chance that multiple instructions would execute in a single `VMRUN`, resulting in a *multi-step*.

We can still try to recover since we know how many instructions were executed in a multi-step. Our chances of recovering from a multi-step process depend on the type of executed instructions.

If a multi-step contains only one unknown branch instruction, we can always recover the outcome of the unknown branch. Since we know how many branch instructions were executed during the multi-step and how many must have been taken, we can compute the outcome by subtracting the known number of taken branches from the measurement.

When a multi-step contains two or more unknown conditional branch instruction with branches of different length, we can try to recover the outcomes by comparing the possible paths taken with our data. We can determine exactly where we landed by using known branch locations and page faults to orient ourselves in the code. By using the interrupt latency, we can have even more of an indication of our current position. We can reduce the possible paths because we know the number of branches taken and the number of instructions executed during the multi-step. If these constraints combine to result in a single path, we can recover the branches' outcomes. However, the likelihood of a successful recovery from such a multi-step shrinks with the number of unknown branches in the multi-stepped section. An increasing number of unknowns expands the number of possible paths unconditionally and drastically increases the chance of multiple combinations with the same length. We cannot recover the exact path if multiple combinations of taken or not-taken branches can reach the same end point.

If a multi-step contains two or more unknown conditional branch instructions with branches of identical length, we are guaranteed to have multiple valid paths to get to the end goal. This means we cannot recover the exact branch outcomes without relying on information from other sources. However, we know the exact location of the unknown outcomes and how many conditions were true. We can use this information to reduce the keyspace of a later bruteforce attack. Additionally, if two branches use different types of instructions, we could still try to recover the exact path of the code. Some options to infer more information about the execution are instruction latency, memory accesses, or other performance counters (e.g., *Div Op Count* or *Retired MMX/FP Instructions*) depending on the attack target.

The impact of multi-steps depends mainly on the exact implementation of the program. Our implementation is designed to page-fault whenever our victim code calls a function. This causes any multi-step to end and effectively acts as a synchronization point. Code that frequently calls other functions is, therefore, straightforward to attack with this method since the impact of a multi-step is minimal. Performance-optimized, concise code without function calls, such as a string-comparison loop, is more challenging to attack since it consists of few instructions. However, even with a multi-step, we can determine how many iterations of the loop were successful before an early exit was triggered.

Since functions like secure byte- or string comparisons are explicitly written to avoid branching, we cannot attack such constant-time code.

Chapter 5.

Evaluation

In this chapter, we present our experiments, and discuss the results.

5.1. Toy Example

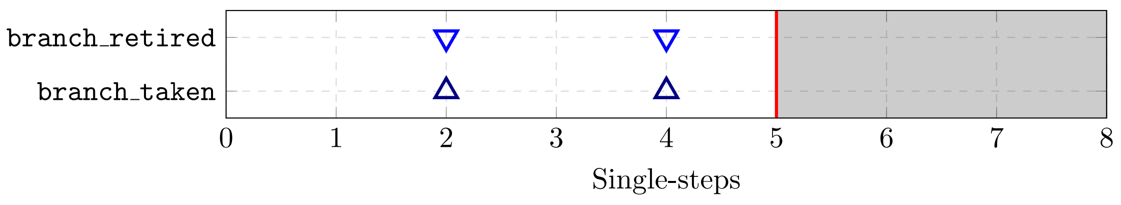
```
.align 4096
.globl function_target_wrapper
function_target_wrapper: ; Wrapper function to terminate singlestepping
    call r9 ; first 5 params already initialized
    ret

.align 4096 ; align code to have it on its own page
.globl simple_branch_target
simple_branch_target:
    cmp rdi, 0 ; rdi is provided by the caller
    je jumptarget
    nop ; NOPs added to have extra instructions in one path
    nop
    nop
jumptarget:
    nop
    ret
```

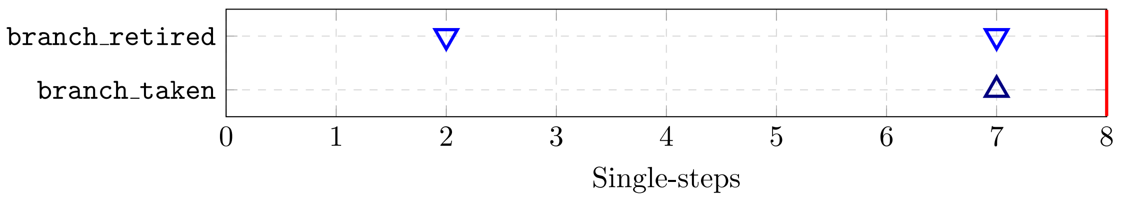
Listing 5.1: Sample victim code with a single branch instruction. The initial value of `rdi` decides between two branches with different lengths.

As a first proof of concept, we attack a code snippet with a single conditional instruction. We wrote the assembly code by hand and used the end2end example code in the SEV-Step repository for this experiment. As our code requires a wrapper function to know when it can terminate, we introduced a function to the SEV-Step victim code that takes a function address in `R9` and calls it, forwarding the other parameter registers as-is. We also extended the HTTP communication so the attacker can pass parameters to victim functions in order to avoid recompiling the victim code with every experiment. We used this function to call our actual target functions. Our first target function is shown in Listing 5.1.

Figure 5.1 shows a graphical representation of the recorded trace data. A symbol on the respective line means that the corresponding `branch_retired` or `branch_taken`



(a) Function called with `RDI=0`.



(b) Function called with `RDI=1`.

Figure 5.1.: Effect of data in `RDI` on the performance-counter trace. Branch events that occurred during each single-step are marked on the corresponding line. The red vertical line represents the end of the collected trace. Figure 5.1a shows a taken branch at step 2, and ends earlier, while Figure 5.1b shows that the branch was not taken, resulting in a longer trace.

event occurred at the marked step. The most noticeable difference between Figure 5.1a and Figure 5.1b is that the Figure 5.1a is shorter. This is caused by the different number of instructions in the two possible paths. In Figure 5.1a, the data shows a **`branch_taken`** event at the second single-step, which means we know that `RDI` was zero since this is what influences the result of line 11 at Listing 5.1. Consequently, the victim jumps to line 15, skipping a few `NOP`s. We can verify this by counting the instructions. In Figure 5.1b, we can see a **`branch_retired`** at the second single-step, which lets us know that there was a branching instruction. By checking the corresponding **`branch_taken`** value, we can find out that the branch was not taken, which tells us that the value in `RDI` was not zero.

As visible in both visualizations, the last single-step event is a taken branch instruction for both code paths. This is the `RET` instruction in line 17 of Listing 5.1. The `RET` instruction and other unconditional branch instructions line `JMP` or `CALL` show up in both the *Retired Taken Branch Instructions* and *Retired Branch Instructions* performance counters.

We could filter these unconditional jumps by observing the *Retired Conditional Branch Instructions* (`0x0D1`) performance counter instead of *Retired Branch Instructions*. However, in our experiments, the noise in the data created by unconditional branches does not negatively impact our ability to recover code paths. On the contrary, the known branch instructions help us navigate the output files more quickly to find the parts that contain the leaked data.

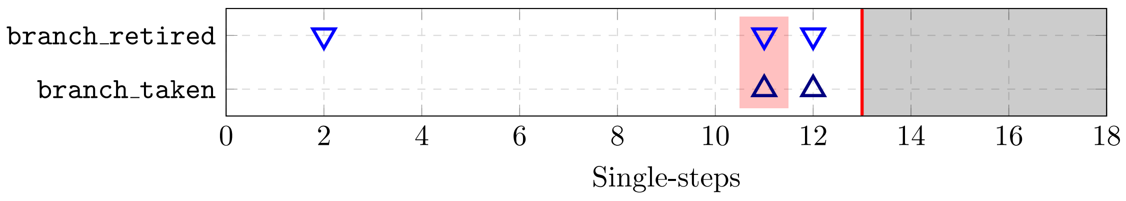
5.2. 64-bit Square-and-Multiply in Assembly

```
 1  ; RDI: m
 2  ; RSI: exponent
 3  ; RDX: modulus
 4
 5  .align 4096
 6  .globl actual_squareandmultiply
 7  actual_squareandmultiply:
 8      nop
 9      nop
10      mov rcx, rdx        ; keep modulus in rcx for easier div management
11      mov rdx, 0          ; zero rdx for multiplication
12      mov rax, rdi        ; initialize result variable to rdi
13      mov r8, rdi         ; backup rdi for multiply
14
15  sam_repeat:
16      cmp rsi, 0          ; exponent is done
17      je sam_end
18      xor rdx, rdx
19      mul rax             ; rdx:rax * rax, square
20      div rcx             ; remainder in rdx
21      mov rax, rdx        ; move remainder back to rax
22      mov rdi, rsi        ; check for last bit
23      and rdi, 1
24      shr rsi, 1
25      cmp rdi, 1
26      jne sam_skip        ; bit is 1 -> multiply
27      xor rdx, rdx        ; zero rdx
28      mul r8              ; multiply with base
29      div rcx             ; remainder in rdx
30      mov rax, rdx
31
32  sam_skip:
33      jmp sam_repeat
34
35  sam_end:
36      ret
```

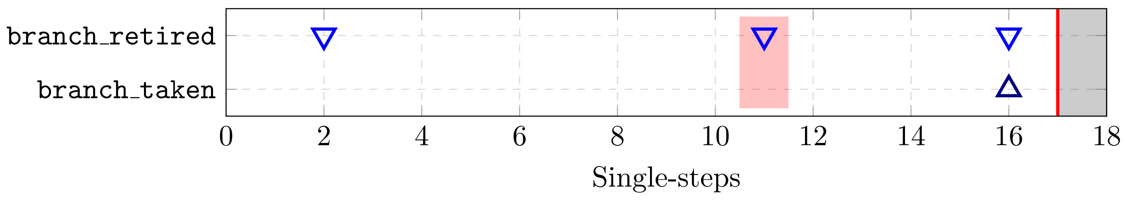
Listing 5.2: Sample implementation of square-and-multiply in assembly, without bignum support, which limits the size of the modulus to 64 bit. Line 26 leaks the key.

The square-and-multiply algorithm is a method for performing modular exponentiation of large numbers. Therefore, it can be used to perform RSA operations. Other attacks also target the conditional branch of Square-and-Multiply to leak the key (e.g., [45]), but they are noisy and require multiple measurements.

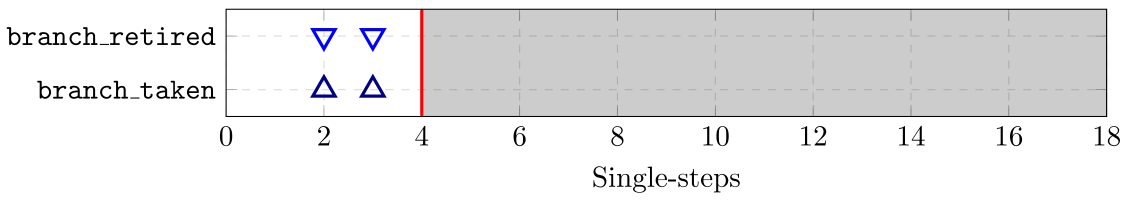
Our example code for square-and-multiply can be found in Listing 5.2. Our implementation only supports numbers up to 64 bits so as not to require complex functions for bignum operations. We call the function using the same wrapper as in Listing 5.1.



(a) Loop iteration where the current bit is zero. The secret-dependent branch is indicated in red.



(b) Loop iteration where the current bit is zero. The secret-dependent branch is indicated in red.



(c) Last loop iteration

Figure 5.2.: Different types of traces in a square-and-multiply function. The `je sam_end` (line 17 in Listing 5.2) is aligned on step 2.

In this code, we use the two known branches (line 17 and line 33) as a reference to find the instruction that leaks data. We know that we will hit the first branching instruction 8 cycles after getting our page fault in the victim function. By analyzing the assembly code, we know that the exponent-dependent jump instruction happens 9 instructions after the CPU does not take the jump to the end of the function. If the exponent's least significant bit was 1, the execution reaches the jump instruction of line 33 exactly 4 cycles later; if it were 0, it would jump again after the `jne`.

Figure 5.2 shows the three different possible traces that we get when single-stepping Listing 5.2. In Figure 5.2a, the processed bit is zero, so the squaring part of the square-and-multiply algorithm is skipped. This is visible via the branch information at step 11, and the length of the loop. Figure 5.2b shows the trace when the current bit is one. This time, the victim does not take the branch, so it executes the additional squaring instructions, which results in a longer loop iteration. The last instruction in both cases is an unconditional `JMP` instruction, which always shows up in the trace as a taken branch. Figure 5.2c shows the last iteration of the loop, where the exit condition is true. The last instruction in that case, the `RET` instruction, also shows up as a taken branch. We recover one bit of the exponent by monitoring the branch instruction at step 11. Additionally, we know how long the loop iteration is. This means we can easily find the branching instruction again in the next iteration. We can recover the entire key by tracing all loop iterations.

We ran this experiment with 32-bit exponents and 64-bit exponents. We need about 500 single-steps to recover a 32-bit exponent using this method. For 64-bit exponents, this number rises to approximately 1000, with the exact number depending on the exponent. We let the victim generate random exponents, which were then sent to the attacker to verify the recovered key. Because this method was relatively fast for this experiment, we did not implement recovery from multi-steps, opting to discard the data and start a new run.

We were able to successfully single-step 64-bit exponentiations 242 out of 300 attempts. We also traced 32-bit exponentiations in 866 out of 1000 attempts. When running without multi-steps, we can recover the exponent 100% of the time. When multi-steps occur, we can only recover key bits before to the multi-step and, if we know the length of the key, the following bits. We know how many bits during the multi-step were 1. However, since the distribution of zeroes and ones in a randomly generated private key is uniform, this does not provide much information. However, we know the location and number of unknown bits and can brute-force them if the amount of unknown bits is reasonably low.

5.3. 2048-bit Textbook RSA Decryption

To verify our attack with larger key sizes, we used the *tiny-bignum-c* library [36] and a modified version of their RSA test case. We adapted the library to support up to 2048-bit numbers. As test data, we generated a private key and ciphertext, which the program decrypts to display the encrypted message. The implementation of the victim program is inspired by the latest experimental VM server released by Wilke et al. [78] shortly before the development of this exploit. Since it has been shown before that it is possible to find guest-physical addresses by behavioral analysis of the victim [41], we do not consider this part of our attack.

When the program starts, it prints the guest-physical addresses of the wrapper and target functions to the terminal and wait for input. The attacker program starts tracking those pages, after which the victim performs its computation.

Most of the heavy computational work of the decryption is handled by library-provided functions. The secret-dependent branching instruction that we use to leak the private key is located at address `rsa_decrypt+8B` in Figure 5.3.

The main loop of the minimal implementation discussed in Section 5.2 consisted of 16 instructions. When compiled, an iteration of the loop in `pow_mod_faster` consists of 45 instructions, or 33 if the current bit of the exponent is zero. This means that each loop has 2 - 2.5 times more instructions than the previous experiment. Additionally, the exponent at 2048 bits is 32 times larger than the 64-bit exponent of the previous experiment. We used these numbers to estimate the expected number of instructions:

$$new\_instructions = old\_instructions * 32 * 2.25 = 72000$$

```c
// Adapted from https://github.com/kokke/tiny-bignum-c/blob/
    ac136565378c624365e0f5f556d386b3966bff32/tests/rsa.c
void pow_mod_faster(struct bn* a, struct bn* b, struct bn* n, struct bn*
    res)
{
    bignum_from_int(res, 1); /* r = 1 */
    struct bn tmpa;
    struct bn tmpb;
    struct bn tmp;
    bignum_assign(&tmpa, a);
    bignum_assign(&tmpb, b);

    while (1)
    {
    if (tmpb.array[0] & 1)     /* if (b % 2) */
    {
        bignum_mul(res, &tmpa, &tmp);  /*   r = r * a % m */
        bignum_mod(&tmp, n, res);
    }
    bignum_rshift(&tmpb, &tmp, 1); /* b /= 2 */
    bignum_assign(&tmpb, &tmp);

    if (bignum_is_zero(&tmpb))
        break;

    bignum_mul(&tmpa, &tmpa, &tmp);
    bignum_mod(&tmp, n, &tmpa);
    }
}
```

Listing 5.3: Implementation of textbook-RSA using *tiny-bignum-c* [36].

In our assembly implementation, we needed around 1000 single steps to trace the whole program, with a success rate of approximately 70%. Assuming this value scales linearly, the probability of single-stepping every instruction in this main loop is $0.70^{72} = 0.00000000000703$.

Fortunately, the *Instructions Retired* performance counter shows how many instructions were executed in a multi-step. As discussed in Chapter 4, we can use this information to recover from accidental multi-steps.

However, in the case of our victim program, we have an even easier and more reliable way to avoid data loss due to failed single-stepping. Looking at Figure 5.3, we can see that multiple functions are called throughout each loop iteration. We track each function call to turn off single-stepping when we do not need it. The page faults that trigger when this happens act as a synchronization point and stop multi-steps.

We want to recover the condition at `rsa_decrypt+89`. Considering the worst possible case: The program returns from `bignum_mod` at `rsa_decrypt+152` and immediately multi-steps until the subsequent page fault stops it. This page fault occurs when `bignum_mul` is called at `rsa_decrypt+A8`. When we check which branch instructions are executed in this section, notice the following:

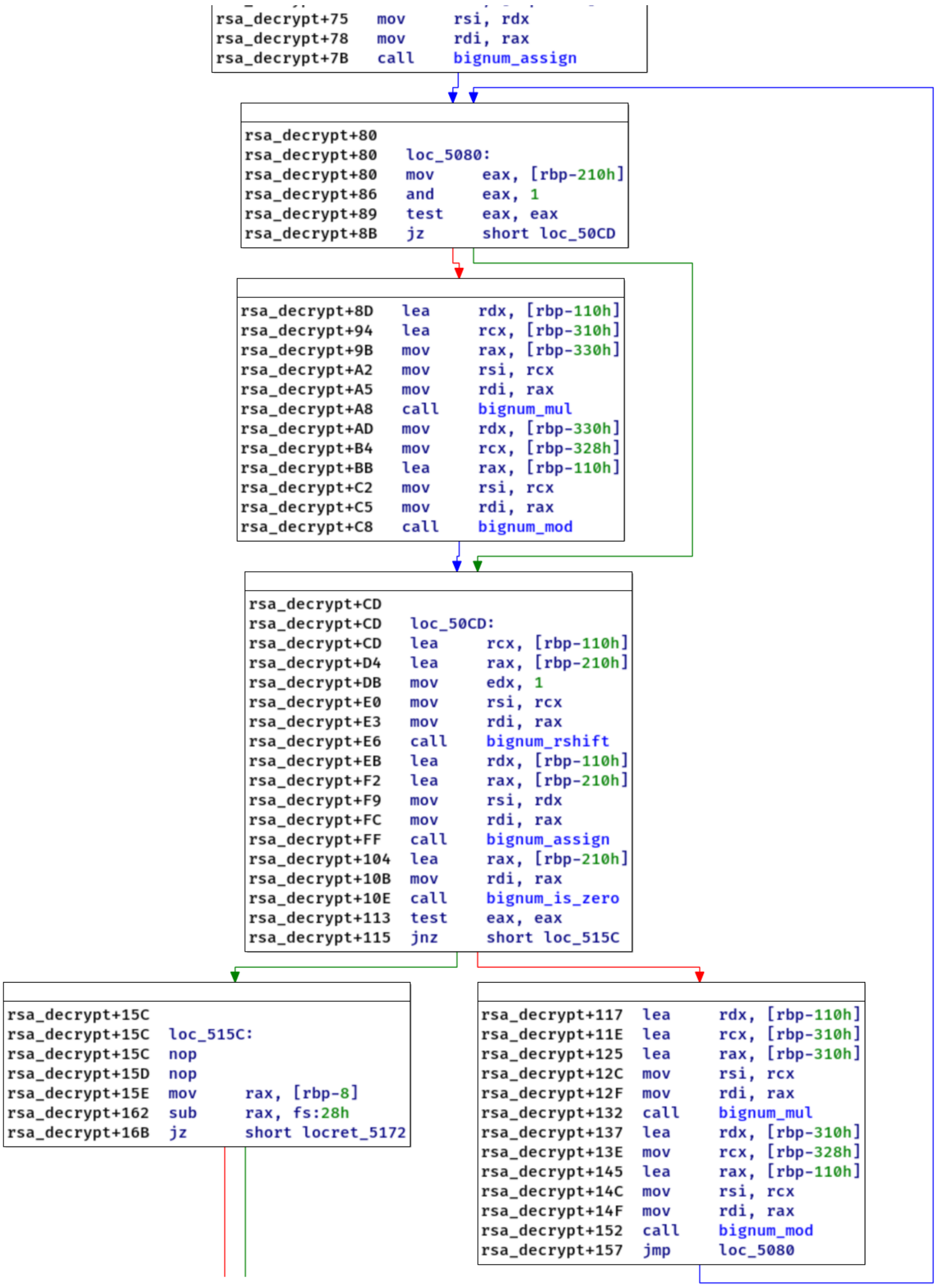
1. `jmp loc_5080` $\rightarrow$branch_retired = 1, branch_taken = 1
2. `jz short loc_50CD` $\rightarrow$branch_retired = 2, branch_taken = 1 + x
3. `call bignum_{mul,rshift}` $\rightarrow$branch_retired = 3, branch_taken = 2 + x

As we can see, in the entire section, only the result of the `jz` instruction is dynamic, while every other instruction always increments both branch_retired and branch_taken. We can therefore determine:

$$x = 1 \iff \text{branch\_retired} = \text{branch\_taken}$$

We use equality as an indicator instead of assuming that the value must be 3 if $x = 1$ since we occasionally measured the values $(2, 1)$ for our value pair after multistepping. These strange value pairs appeared in clusters, and we theorize that the `call` instruction that causes the page fault might not increment the performance counters before faulting in some cases. However, using equality as an indicator mitigates this problem since instructions will either increment all performance counters or none.

Because we have very fine-grained control over what is single-stepped, although limited by the placement of the code in memory, we can afford to leave single-stepping disabled for computationally heavy functions. Theoretically, this could be optimized even further by single-stepping only the block between the two external functions containing the conditional branch. However, we opted for a simpler method, allowing for better debugging workflow since all steps in the target function are recorded. Using this method, we can extract a full 2048-bit private key, without any bit errors, from a program running the encryption inside a SEV-SNP virtual machine in around 4 minutes.



```
rsa_decrypt+75   mov     rsi, rdx
rsa_decrypt+78   mov     rdi, rax
rsa_decrypt+7B   call    bignum_assign
```

```
rsa_decrypt+80
rsa_decrypt+80   loc_5080:
rsa_decrypt+80   mov     eax, [rbp-210h]
rsa_decrypt+86   and     eax, 1
rsa_decrypt+89   test    eax, eax
rsa_decrypt+8B   jz      short loc_50CD
```

```
rsa_decrypt+8D   lea     rdx, [rbp-110h]
rsa_decrypt+94   lea     rcx, [rbp-310h]
rsa_decrypt+9B   mov     rax, [rbp-330h]
rsa_decrypt+A2   mov     rsi, rcx
rsa_decrypt+A5   mov     rdi, rax
rsa_decrypt+A8   call    bignum_mul
rsa_decrypt+AD   mov     rdx, [rbp-330h]
rsa_decrypt+B4   mov     rcx, [rbp-328h]
rsa_decrypt+BB   lea     rax, [rbp-110h]
rsa_decrypt+C2   mov     rsi, rcx
rsa_decrypt+C5   mov     rdi, rax
rsa_decrypt+C8   call    bignum_mod
```

```
rsa_decrypt+CD
rsa_decrypt+CD   loc_50CD:
rsa_decrypt+CD   lea     rcx, [rbp-110h]
rsa_decrypt+D4   lea     rax, [rbp-210h]
rsa_decrypt+DB   mov     edx, 1
rsa_decrypt+E0   mov     rsi, rcx
rsa_decrypt+E3   mov     rdi, rax
rsa_decrypt+E6   call    bignum_rshift
rsa_decrypt+EB   lea     rdx, [rbp-110h]
rsa_decrypt+F2   lea     rax, [rbp-210h]
rsa_decrypt+F9   mov     rsi, rdx
rsa_decrypt+FC   mov     rdi, rax
rsa_decrypt+FF   call    bignum_assign
rsa_decrypt+104  lea     rax, [rbp-210h]
rsa_decrypt+10B  mov     rdi, rax
rsa_decrypt+10E  call    bignum_is_zero
rsa_decrypt+113  test    eax, eax
rsa_decrypt+115  jnz     short loc_515C
```

```
rsa_decrypt+15C
rsa_decrypt+15C  loc_515C:
rsa_decrypt+15C  nop
rsa_decrypt+15D  nop
rsa_decrypt+15E  mov     rax, [rbp-8]
rsa_decrypt+162  sub     rax, fs:28h
rsa_decrypt+16B  jz      short locret_5172
```

```
rsa_decrypt+117  lea     rdx, [rbp-110h]
rsa_decrypt+11E  lea     rcx, [rbp-310h]
rsa_decrypt+125  lea     rax, [rbp-310h]
rsa_decrypt+12C  mov     rsi, rcx
rsa_decrypt+12F  mov     rdi, rax
rsa_decrypt+132  call    bignum_mul
rsa_decrypt+137  lea     rdx, [rbp-310h]
rsa_decrypt+13E  mov     rcx, [rbp-328h]
rsa_decrypt+145  lea     rax, [rbp-110h]
rsa_decrypt+14C  mov     rsi, rcx
rsa_decrypt+14F  mov     rdi, rax
rsa_decrypt+152  call    bignum_mod
rsa_decrypt+157  jmp     loc_5080
```

Figure 5.3.: Disassembly of main decryption loop compiled from Listing 5.3.

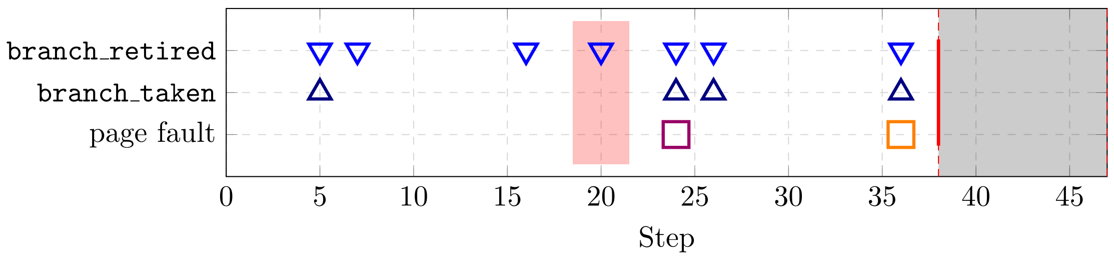
5.4. Mbed TLS RSA Key Recovery

```
 1 for(;;) {
 2   // ...
 3   // ei contains the current bit of the exponent
 4   if (ei == 0 && state == 1) {
 5     MBEDTLS_MPI_CHK(mpi_select(&WW, W,
 6             w_table_used_size, x_index));
 7     mpi_montmul(&W[x_index], &WW, N, mm, &T);
 8     continue;
 9   }
10   state = 2;
11   nbits++;
12   exponent_bits_in_window |= (ei << (window_bitsize - nbits));
13   if (nbits == window_bitsize) {
14     for (i = 0; i < window_bitsize; i++) {
15       MBEDTLS_MPI_CHK(mpi_select(&WW, W,
16               w_table_used_size, x_index));
17       mpi_montmul(&W[x_index], &WW, N, mm, &T);
18     }
19     MBEDTLS_MPI_CHK(mpi_select(&WW, W,
20             w_table_used_size, exponent_bits_in_window));
21     mpi_montmul(&W[x_index], &WW, N, mm, &T);
22     state--;
23     nbits = 0;
24     exponent_bits_in_window = 0;
25   }
26   // ...
27 }
```

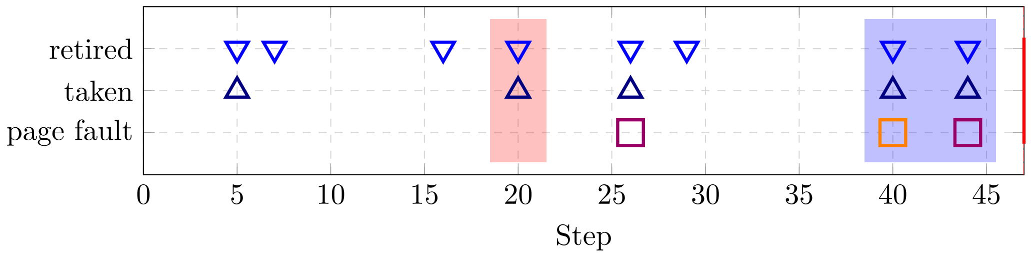
Listing 5.4: Mbed TLS' `mbedtls_mpi_exp_mod` function. We can attack the branching condition in Line 4.

We created a victim program using the *Mbed TLS* library (version 3.5.2) [44] to verify the real-world applicability of `CRACKPIPE`. We configured the library with a window size of 1, forcing it to use the square-and-multiply algorithm for exponentiation.

The relevant part of the `mbedtls_mpi_exp_mod` function, which is used for modular exponentiation, is shown in Listing 5.4. The basic principle of the attack is the same as the attack in Section 5.3. The function loads the current bit of the exponent into the `ei` variable. We attack the secret-dependent branching condition in line Line 4. Since we set our window size to 1, compiler optimizations remove the branching conditions at Line 13 and Line 14. Even with these optimizations, the `mbedtls_mpi_exp_mod` function is still significantly more complex than our simple example code. There are multiple different code paths containing various types of branching instructions. While function calls in Section 5.3 are placed such that the resulting page faults synchronize any potential multi-steps with only one unknown, secret-dependent branching condition between page faults, this is not the case in Mbed TLS. As a result, when our trace contains multi-steps,



(a) `ei` = 0



(b) `ei` = 1

Figure 5.4.: Event patterns for single iterations of the exponentiation loop. If `ei` is 0, the attacked branch instruction in Step 20 is not taken. If `ei` is 1, the attacked branch is taken, which later causes a pattern where `mpi_montmul` and `mpi_select` are called without any branch instructions in between. This sequence is used for reliable multi-step recovery.

we can not reliably isolate the effects of the secret-dependent branching instructions on the *Retired Taken Branch Instructions* performance counter.

However, we can detect that `ei` is 1 via a special case in that code branch. The `MBEDTLS_MPI_CHK` macro automatically checks function return values, and performs error handling. To do this, `MBEDTLS_MPI_CHK` adds additional branches after the function call. However, in Line 19, the `mpi_select` function gets called directly after `mbedtls_mpi_exp_mod`, without any branching instructions between them. This is the only instance in the machine code where `mbedtls_mpi_exp_mod` and `mpi_select` are called without branching instructions between them, which means that detecting this pattern is a reliable way of distinguishing the two branches that depend on `ei`.

With this technique, we can recover a full 4096-bit RSA private key without any bit errors. We traced the signature process with 10 different private keys. We were able to recover the key from each trace. Tracing a Mbed TLS decryption process requires around 200.000 single-steps, which takes just over 7 minutes.

5.5. TOTP Brute Force Attack

In this case study, we show that common code patterns can be targeted to leak information about the data inside a SEV-SNP virtual machine. As an example, we looked for a

```
1 for (size_t i=0; i<data->digits; i++)
2 {
3     if (key[i] != time_str[i])
4         return OTP_ERROR;
5 }
```

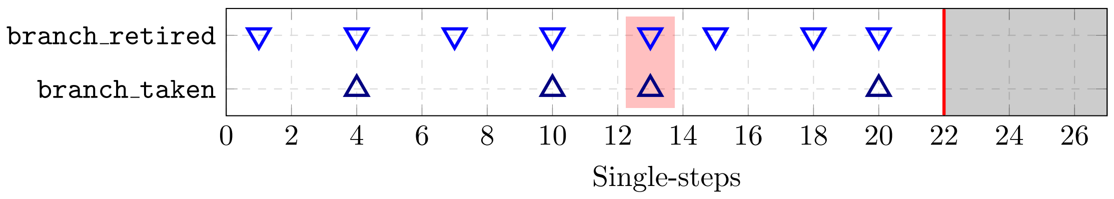
Listing 5.5: COTP token comparison code. `key` is the user-provided input, which is compared to `time_str`, containing the correct TOTP token. Line 3 exits early if the comparison fails.

simple TOTP implementation written in C. We found the *COTP* library [66] and created a small wrapper program around it. The wrapper program provides the library with a hardcoded key, and uses the library-provided functions as shown in the documentation. The attack does not depend on any particular flaw in the implementation of the program using the library. Our victim program reads TOTP codes from the standard input and verifies them using the `totp_verify` function. This function generates a valid TOTP token from the secret and compares it with the input.

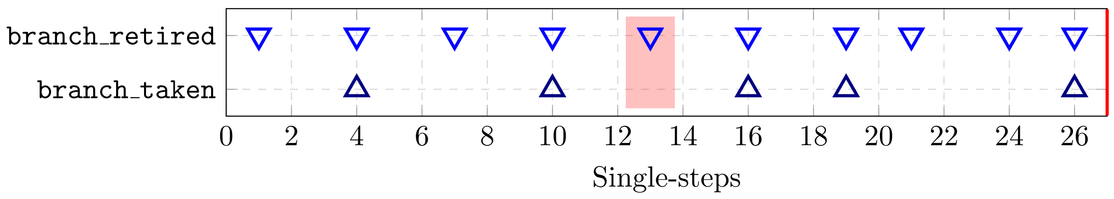
However, this comparison is performed using a string-comparison loop with an early exit, as shown in Listing 5.5. We can detect this early-exit condition using CRACKPIPE. We can detect how many of our input characters were correct for every attempt. This allows us to brute-force the token character-by-character, which reduces the average number of brute-force attempts from $10^6/2 = 500.000$ to $10 \cdot 6/2 = 30$.

To verify our attack, we implemented a script that connects to the victim via SSH. The attacker script starts our trace generation tool and then types a token guess into the victim program. The script immediately analyzes the generated trace, determining how many loop iterations of the equality check were executed. Based on this information, the script adjusts the next key guess.

With this strategy, we can perform 2 key guesses per second, which results in 60 guesses in 30 seconds, the default expiration time of TOTPs. We are able to recover the correct TOTP token within this timeframe 50 out of 50 times, with an average of 31.1 guesses and an average runtime of 18.14 s per token.



(a) TOTP check with 2 correct digits.



(b) TOTP check with 3 correct digits.

Figure 5.5.: The TOTP verification trace forms a distinct pattern of 2 branch instructions per character, consisting of the length check and the early-exit. The highlighted branch instruction leaks if the guessed TOTP character was correct.

5.6. TOTP Secret Recovery

This case study shows how something as simple as an insecure key decoding algorithm can compromise security. We use the same victim program as in Section 5.5. Instead of the TOTP code, we target the secret in this attack.

The library decodes the Base32 encoded secret string to bytes to generate the TOTP token using the `otp_byte_secret` function. Since Base32 encodes 5 bits of information per character, the library decodes 8 characters at a time, resulting in 5 output bytes. The code that performs this conversion is shown in **??**.

The Base32 decoding algorithm uses the `OTP_DEFAULT_BASE32_CHARS` array as a lookup table for Base32 encoding. The algorithm iterates through each character in a block and determines the 5-bit value of each character by finding its position in the lookup table. It does this by iterating through the list and comparing all lookup table entries to the current character. Utilizing `CRACKPIPE`, we can identify when the loop's exit condition is true and reconstruct the character. Following the code flow back to the lookup table loop allows us to reconstruct all Base32-encoded characters of the TOTP secret.

Because the critical code section does not cause any page faults, multi-steps cannot be caught. Therefore, we cannot recover the full secret key if a multi-step spans multiple characters. In our experiments, we recover the secret key in 75 out of 86 cases.

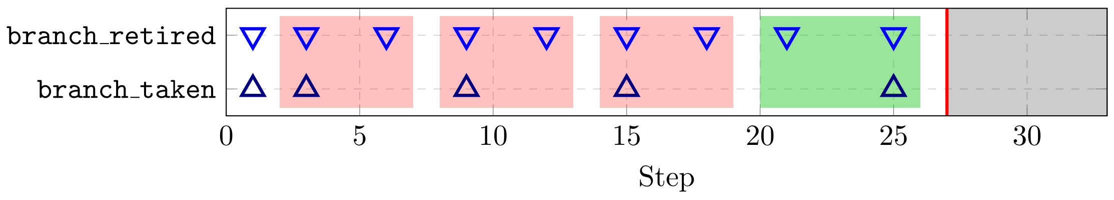
The issue stems from a flawed implementation of Base32 decoding. However, it should be noted that when searching GitHub, multiple TOTP implementations have the same flaw [63, 30]. Other implementations do not leak the entire secret, but instead let us recover information about whether each character in the secret is a letter or a digit [22,

```
 1 static const char OTP_DEFAULT_BASE32_CHARS[32] = {
 2 'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P',
 3 'Q','R','S','T','U','V','W','X','Y','Z','2','3','4','5','6','7'
 4 };
 5
 6 COTPRESULT otp_byte_secret(OTPData* data, char* out_str)
 7 {
 8   //...
 9   for (size_t i = 0; i < num_blocks; i++) {
10     unsigned int block_values[8] = { 0 };
11     for (int j = 0; j < 8; j++) {
12       char c = data->base32_secret[i * 8 + j];
13       int found = 0;
14       for (int k = 0; k < 32; k++) {
15         if (c == OTP_DEFAULT_BASE32_CHARS[k]) {
16           block_values[j] = k;
17           found = 1;
18           break;
19         }
20       }
21       //...
22     }
23     //...
24   }
25
26   return OTP_OK;
27 }
```

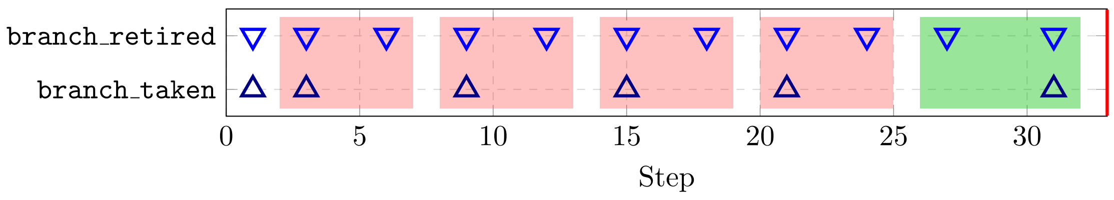
Listing 5.6: COTP `otp_byte_secret` Base32 decoding algorithm. The early-exit condition at line 15 leaks the TOTP secret.

57, 20, 54]. Additionally, when searching for Base64 decoding libraries in C, we can find multiple cases where the full encoded data is recoverable [75] or where the algorithm leaks information about character classes [55, 34]. Although those implementations were not originally intended for trusted execution, it is possible that similar implementations may exist as parts of other projects.

We also checked the default implementations for current versions of node.js, Go, Python, and Rust. All of the implementations make use of a dedicated Base64 decoding table, where invalid entries are mapped to a known value for error checking. This approach makes it impossible to recover the encoded data using `CRACKPIPE`. However, the translation tables might be vulnerable to cache attacks, similar to AES T-Tables.



(a) Base32 character is `D`



(b) Base32 character is `E`

Figure 5.6.: Event trace for the base32 decoder of the COTP library. Each loop iteration appears as two branches. The first branch is taken until the input character matches `OTP_DEFAULT_BASE32_CHARS[k]`. Therefore, the current base32 character can directly be derived from the number of taken branches.

Chapter 6. Discussion

In this chapter, we discuss the scope of the `CRACKPIPE` attack. We estimate the impact across common software stacks. Finally, we present possible defenses.

6.1. Attack Scope

As we showed in Chapter 5, `CRACKPIPE` can recover the execution path through a program. The severity of this attack depends on the target application. Any secret-dependent branch in the code leaks information about the secret. It is possible to harden cryptographic functions and implement them as constant-time code, preventing attacks like the one shown in Section 5.3. However, this often requires manual assembly implementations and careful choices of functions, leading to increased development- and execution time. Modern cryptographic libraries usually implement constant-time functionality already.

The `CRACKPIPE` attack can leak any data used to decide branching conditions. SEV is supposed to allow for trusted cloud computing, meaning that all private data, not only private keys, is safe even from the hypervisor. However, with `CRACKPIPE`, any code touching private data would have to be constant-time code for the data to be secure. As shown by Lipp et al. [45], it is possible to use power analysis to extract similar data about code paths from an SGX enclave. Therefore, SGX enclaves must use constant-time code to avoid leaking secrets. This is feasible when considering a small TEE like an SGX enclave, but in SEV, the TEE is the entire virtual machine. SGX forces users to carefully consider which parts to include in the enclave and which data is safe for the host to see. If a secure connection to a server is needed, the user has to implement an encrypted channel starting from the enclave. The whole concept of SGX makes this evident to users - anything outside the enclave is visible to the host.

With SEV, the entire virtual machine is supposed to be encrypted and secure from the outside world. However, AMD states specifically that SEV-SNP does not protect against microarchitectural side channels. They state that, as with standard software security practices, code which is sensitive to such side-channel attacks (e.g., cryptographic libraries) should be written in a way that prevents such attacks [6].

While hardened cryptographic libraries can protect the processed data and keys while encrypting and decrypting, all parts of the virtual machine that are not explicitly written in constant-time code are susceptible to the `CRACKPIPE` attack.

Every piece of code that handles sensitive data can accidentally leak information about it. This may include the program itself, any other tools installed on the server, an entire web stack, and the whole Linux kernel. Most of these tools were not developed with microarchitectural side-channel protection in mind, since, in a classical scenario, the hypervisor can access all the guest data by design. SEV introduces a new scenario where code has to be protected even from privileged attackers. However, most software does not need consider such attacks and is not designed to protect against them.

AMD states that no fingerprinting attack protection is implemented in SEV-SNP. They say that while fingerprinting can sometimes provide information about the code being run inside a VM, typically the most sensitive information is the data itself (e.g., data in the database), not the code being run (e.g., which version of the database software is being used [6]).

As we have shown in this thesis, this statement is not correct. By monitoring performance counters and access patterns, we can leak information about the data itself. Common functions like `strcmp` leak the length of the patching part of the strings. When one of the inputs is user-controllable, we can recover the other input string of `strcmp` character by character, similar to the attack shown in Section 5.5. Especially when handling long strings, the early-exit condition can greatly impact performance. In a performance-optimized system, such leaking conditions will most likely be present in many code paths. For example, compilers may use nested branches instead of jump-tables when compiling switch-case statements, depending on the possible outcomes. If the hypervisor has a way to trigger these code paths, for example, via a network request, it could extract arbitrary data using this technique. This is a direct contradiction to AMD's statement.

6.2. Impact

AMD states that code sensitive to side-channel attacks should be written in a way that prevents such attacks [6]. However, as we show in this work, protecting cryptographic operations is insufficient if other parts of the system leak unencrypted data. This means that guest owners must audit their entire software stack to avoid leaks or write their own from scratch.

We present a few estimations on how much effort would go into such a task. Let us assume that a developer can audit 200 lines of code per workday or 25 per hour. We also estimate that around 80 % of the code can be ignored since it is irrelevant, unused, or does not touch sensitive data. Remember that these are very coarse estimations to understand the order of magnitude of such a task. We obtained the number of lines of code by running the `cloc` utility [1] on the latest main branch of the respective repository.

[1] `https://github.com/AlDanial/cloc`

Example 1: Linux

Linux alone is estimated to have 25 to 30 million lines of code. To account for the large number of architectures and drivers included in the code, we use 10 million as a baseline. A single developer reviewing 200 lines of code while skipping over 80 % of code (as it is still irrelevant) would need 10,000 person-days or just over 27 person-years to ensure the entire kernel is secure.

Example 2: Website with Database

For this case, we consider a fairly standard web stack. We use nginx as a reverse proxy, node.js as runtime, and a MariaDB database.

Software	Approximate lines of code	Days to review	years to review
nginx	200,000	200	0.55
node.js	8,000,000	8,000	21.92
MariaDB	2,000,000	2,000	5.48
total	10,200,000	10,200	27.95

Table 6.1.: Estimation for required effort to audit a website stack with database.

Example 3: File Server

In this case, we look at a file server running Samba and nextcloud to provide file storage for a company, and runs automated backups using borg. We do not include the time required to audit the web stack of any web interface in this estimation.

Software	Approximate lines of code	Days to review	Years to review
Samba	3,000,000	3,000	8.22
nextcloud	1,000,000	1,000	2.74
borg	60,000	60	0.16
total	4,060,000	4,060	11.12

Table 6.2.: Estimation for required effort to audit file server software.

As we show in Table 6.1 and Table 6.2, auditing the entire technology stack necessary to run a modern web application or file server would each take around 20,000 person-hours or almost 50 person-years. While such an effort is not impossible for a large company in the long run, it would still require a significant amount of time until the entire application stack is audited and patched. However, software-side protections would likely slow down the whole technology stack and would be criticized by users prioritizing performance over security.

6.3. Defenses

`CRACKPIPE` uses legitimate CPU performance counters and infers information from their values. Combined with single-stepping and page tracking, they allow us to get instruction-level branch information. The entire attack does not use bugs or side effects of CPU instructions. Other architectures solve this problem in different ways.

On Intel CPUs, performance counters are disabled when an enclave is running in *production* mode. Although this does not prevent attackers from single-stepping the enclave and obtaining some information through side channels like page faults [80] or interrupt latencies [69], it forces them to use less reliable measurements for their attacks.

ARM's `TrustZone` requires a *secure monitor* that resides on a hierarchically higher level than the Normal World and Secure World in its architecture. The secure monitor performs the world switch when triggered through a mechanism similar to syscalls. While the "host operating system" has more control over the hardware than guest operating systems, it can only control guests via calls to the secure monitor. Modifications to the secure monitors are prevented through integrity checks. However, an attack is still possible if a guest or host finds a vulnerability in the secure monitor and obtains code execution at its permission level.

While `CRACKPIPE` only works on non-constant-time code, as discussed before, converting all code that might touch sensitive data to constant-time is not feasible. Therefore, we discuss how `CRACKPIPE` can be mitigated.

6.3.1. Disabling Performance Counters

The most apparent mitigation for this attack is turning off all performance counters while an SEV virtual machine is running. This solution would mitigate `CRACKPIPE` and drastically reduce the reliability of SEV-Step. Detecting multi-steps would be significantly more challenging and relying on side channels. Recovery from a multi-step would most likely be infeasible because attackers would not know the number of executed instructions. Turning off performance counters entirely (possibly via a guest policy) is the best protection for a guest against `CRACKPIPE` and other single-stepping attacks.

However, in a shared cloud hosting environment, hosting providers may need to know about usage statistics for load-balancing and billing purposes and to protect against attacks from a guest. Performance counters are a common way to mitigate side-channel attacks [13, 21, 47, 2]. While there are no attacks from an SEV guest to a hypervisor published at the time of writing, hiding all performance counter information from the hypervisor may hinder future mitigation development if such an attack is found.

Additionally, a cloud provider may use performance counters to detect activity which is against the terms of service. For example, some providers do not allow crypto mining in some cases [26, 48, 3]. Even in an encrypted machine, such activities are detectable using performance counters [46, 65].

Therefore, turning off performance counters is a trade-off between the security of the guest from a malicious hypervisor, security from a malicious guest, and the provider's ability to monitor activity on their systems.

6.3.2. Detection through Instruction Delays

Single-stepping introduces significant delays to the execution of a program. The guest can measure this delay using the Timestamp Counter (TSC). To do this securely, the guest must enable the *Secure TSC* feature via the `SEV_FEATURES` MSR. When this feature is disabled, the hypervisor can specify an offset, which will be added to the TSC whenever the guest tries to read it. A malicious hypervisor could use this offset to hide the single-stepping delays from the guest. With the Secure TSC feature enabled, the offset is ignored, and the guest can obtain unmodified timing data.

The guest can use this reliable timing data for single-step detections. Chen et al. [17] introduce *Déjà Vu*, a framework that detects attacks on an enclave by measuring the execution time of basic blocks. If the measured time does not match with previously generated training data, the enclave is notified of the potential attack and can react accordingly. Their framework includes a static analysis tool to automatically find and measure basic blocks.

However, splitting all executable code into basic blocks is not feasible when running an entire virtual machine. Instead, developers could place *checkpoints* throughout critical sections of the code, which record the time taken between them. If multiple checkpoints report abnormally high delays between each other, the program can assume it is being single-stepped and handle it accordingly. Similar strategies have been used to detect security-critical properties like hyperthread co-location [16], interrupts [15] throughout the execution of SGX enclave.

A possibility is to instrument the compiler to place such checkpoints randomly throughout the code. While it is likely that the attacker could trick the checkpoints, given knowledge of their positions in the code, this protection would likely trigger in the data-gathering phase before the actual exploit is run. However, depending on the density of the checkpoints, there would be some performance overhead. Additionally, this strategy does not mitigate `CRACKPIPE` when secrets can be recovered by distinguishing total performance counter values between page faults, as described in Section 5.3 and Section 5.4.

Chapter 7. Conclusion

In this thesis, we identified an attack surface of AMD SEV that has not been exploited in published works before. We have shown how malicious cloud providers can reliably monitor performance counters to break the confidentiality of SEV-SNP protected VMs. To the best of our knowledge, this represents the only attack that leverages information gained from observing performance counters to break the confidentiality guarantees in the AMD SEV-SNP security architecture. We have proven that this attack vector is realistic and practical with full key recovery PoC attacks against RSA. Even though AMD states that performance counters only leak information about the running code, we have provided evidence that this is not always the case. We have shown that `CRACKPIPE` is not limited to cryptographic primitives, but enables other types of data leaks as well. We have exploited branching conditions such as early exits in fundamental programming constructs such as string comparisons, and shown their relevance for leaking secret data out of AMD SEV-SNP protected VMs.

We proposed possible mitigations for this issue. However, they all come with their tradeoffs, ranging from performance impacts, even for non-SEV users, to impacting the hosting provider's ability to protect themselves from malicious guests. The most realistic solution to this issue is the restriction of critical performance counters when an SEV-enabled virtual machine is active. However, determining which performance counters to disable is not a trivial task. While leaks by performance counters like *Retired Branch Instruction Taken* are pretty obvious, other counters might leak data more indirectly. Therefore, much more work must be done to harden SEV virtual machines against performance counter side channels.

In conclusion, our work provides additional evidence that outsourcing sensitive computations to untrusted cloud providers, with the presence of an untrusted hypervisor, creates new and unexpected attack surfaces. Previously irrelevant functionality can suddenly lead to new side-channel primitives. This thesis serves as a reminder that security assumptions must be re-evaluated when fundamental computing principles change.

Future Work Possible future work includes the detailed evaluation of other performance counters and possible data leaks they create. Furthermore, the Linux kernel is not designed to defend against malicious hypervisors. Defense against side-channels when operating in a virtual machine is not a priority. Therefore, more attack surfaces that can be used to leak data may exist in the kernel. When controlling the exact timing of interrupts and the contents of all IO operations, the hypervisor could precisely target any race condition in the kernel. This could lead to high success probabilities in attacks that previously were almost impossible due to the precise timing required.

Bibliography

[1] National Security Agency. *Ghidra*. 2024. URL: `https://ghidra-sre.org/`.

[2] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Sourangshu Bhattacharya. “Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks”. In: *Cryptology ePrint Archive* (2017).

[3] Amazon. *AWS Free Tier Terms*. 2018. URL: `https://aws.amazon.com/free/terms`.

[4] AMD. *AMD Secure Encryption Virtualization (SEV) Information Disclosure*. 2021. URL: `https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1013.html`.

[5] AMD. *Processor Programming Reference*. Rev. 0.50. May 2021. URL: `https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55898_B1_pub_0_50.zip`.

[6] AMD. *Strengthening VM isolation with integrity protection and more*. 2020.

[7] AMD. “System Programming”. In: *AMD64 Architecture Programmer’s Manual* 2 (2023), pp. 497–619.

[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. “SpecTLB: a mechanism for speculative address translation”. In: *ISCA*. 2011.

[9] Dan Boneh, Richard A DeMillo, and Richard J Lipton. “On the importance of checking cryptographic protocols for faults”. In: *EUROCRYPT*. 1997.

[10] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. “ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture”. In: *USENIX Security Symposium*. 2022.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. “Software grand exposure: SGX cache attacks are practical”. In: *WOOT*. 2017.

[12] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. “One glitch to rule them all: Fault injection attacks against amd’s secure encrypted virtualization”. In: *CCS*. 2021.

[13] Stefano Carnà, Serena Ferracci, Francesco Quaglia, and Alessandro Pellegrini. “Fight Hardware with Hardware: Systemwide Detection and Mitigation of Side-channel Attacks Using Performance Counters”. In: *Digital Threats: Research and Practice* 4.1 (2023).

[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution". In: *EuroS&P*. 2019.

[15] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. "Defeating Speculative-Execution Attacks on SGX with HyperRace". In: *DSC*. 2019.

[16] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races". In: *S&P*. 2018.

[17] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. "Detecting privileged side-channel attacks in shielded execution with Déjá Vu". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 2017.

[18] Victor Costan and Srinivas Devadas. "Intel SGX explained". In: *Cryptology ePrint Archive* (2016).

[19] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks". In: *CHES* (2018).

[20] Francis Davidson. *GitHub – Theldus/Tiny2FA: A small C library that implements TOTP, compatible with Google Authenticator*. 2023. URL: `https://github.com/Theldus/Tiny2FA`.

[21] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. "On the feasibility of online malware detection with performance counters". In: *ACM SIGARCH computer architecture news* (2013).

[22] drweasel. *GitHub – drweasel/totp: Basic C++ TOTP implementation based on libsodium*. 2023. URL: `https://github.com/drweasel/totp`.

[23] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. *Secure Encrypted Virtualization is Unsecure*. 2017.

[24] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *ASPLOS*. 2018.

[25] Agner Fog. "The microarchitecture of Intel, AMD and VIA CPUs". In: *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering* (2023).

[26] Google. *Google Cloud Platform Terms of Service*. 2024. URL: `https://cloud.google.com/terms`.

[27] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache attacks on Intel SGX". In: *EuroSys*. 2017.

[28] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: a fast and stealthy cache attack". In: *DIMVA*. 2016.

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache template attacks: Automating attacks on inclusive Last-Level caches". In: *USENIX Security Symposium*. 2015.

[30] gurushida. *GitHub – gurushida/totp: A C program to generate 6 digit TOTP codes like Google Authenticator*. 2019. URL: `https://github.com/gurushida/totp`.

[31] Felicitas Hetzelt and Robert Buhren. "Security analysis of encrypted virtual machines". In: *ACM SIGPLAN Notices* (2017).

[32] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. "Bluethunder: A 2-level directional predictor based side-channel attack against SGX". In: *CHES* (2020).

[33] David Kaplan, Jeremy Powell, and Tom Woller. *AMD memory encryption*. 2016.

[34] Tomas Kislan. *GitHub – tkislan/base64: Base64 encoding and decoding for C++ projects*. 2020. URL: `https://github.com/tkislan/base64`.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[36] kokke. *tiny-bignum-c*. 2022. URL: `https://github.com/kokke/tiny-bignum-c`.

[37] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *WOOT*. 2018.

[38] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. *Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing*. 2017. eprint: `1611.06952`.

[39] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. "A systematic look at ciphertext side channels on AMD SEV-SNP". In: *S&P*. 2022.

[40] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "Crossline: Breaking "security-by-crash" based memory isolation in amd sev". In: *CCS*. 2021.

[41] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *USENIX Security Symposium*. 2019.

[42] 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.

[43] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. "TLB Poisoning Attacks on AMD Secure Encrypted Virtualization". In: *ACSA*. 2021.

[44] Linaro. *MBed TLS*. 2024. URL: https://www.trustedfirmware.org/projects/mbed-tls/.

[45] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. "PLATYPUS: Software-based power side-channel attacks on x86". In: *S&P*. 2021.

[46] Ganapathy Mani, Vikram Pasumarti, Bharat Bhargava, Faisal Tariq Vora, James MacDonald, Justin King, and Jason Kobes. "Decrypto pro: Deep learning based cryptomining malware detection using performance counters". In: *Autonomic Computing and Self-Organizing Systems (ACSOS)*. 2020.

[47] Robert Martin, John Demme, and Simha Sethumadhavan. "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks". In: *ACM SIGARCH computer architecture news* (2012).

[48] Microsoft. *Azure Free Trial — Microsoft Azure*. 2024. URL: https://azure.microsoft.com/en-us/pricing/offers/ms-azr-0044p/.

[49] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "Cachezoom: How SGX amplifies the power of cache attacks". In: *CHES*. 2017.

[50] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *USENIX Security Symposium*. 2020.

[51] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "Severed: Subverting AMD's virtual machine encryption". In: *EuroSys*. 2018.

[52] Gal Motika and Shlomo Weiss. "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard". In: *Computer Standards & Interfaces* (2012).

[53] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based fault injection attacks against Intel SGX". In: *S&P*. 2020.

[54] Minus Nolldag. *GitHub – minusnolldag/totp: TOTP implementation in C*. 2022. URL: https://github.com/minusnolldag/totp.

[55] Rene Nyffenegger. *GitHub – ReneNyffenegger/cpp-base64: base64 encoding and decoding with c++*. 2022. URL: https://github.com/ReneNyffenegger/cpp-base64.

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES". In: *CT-RSA*. 2006.

[57] patzol768. *GitHub – patzol768/cpp-otp: A One Time Password (OTP) library in C++, with QR code generation*. 2023. URL: https://github.com/patzol768/cpp-otp.

[58] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Crosstalk: Speculative data leaks across cores are real". In: *S&P*. 2021.

[59] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-oriented programming: Systems, languages, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* (2012).

[60] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-privilege-boundary data sampling". In: *CCS*. 2019.

[61] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware guard extension: Using SGX to conceal cache attacks". In: *DIMVA*. 2017.

[62] Michael Steil. "Inside VMware". In: 2006. URL: `https://fahrplan.events.ccc.de/congress/2006/Fahrplan/attachments/1132-InsideVMware.pdf`.

[63] Paolo Stivanin. *GitHub – paolostivanin/libcotp: C library that generates TOTP and HOTP according to RFC-6238*. 2023. URL: `https://github.com/paolostivanin/libcotp`.

[64] Geoffrey Strongin. "Trusted computing using AMD "Pacifica" and "Presidio" secure virtual machine technology". In: *Information Security Technical Report* (2005).

[65] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. "Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises". In: *RAID*. 2017.

[66] Cody Tilkins. *GitHub – tilkinsc/COTP: A simple One Time Password (OTP) library in C, supports C++*. 2023. URL: `https://github.com/tilkinsc/COTP`.

[67] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. "Intel virtualization technology". In: *Computer* 38.5 (2005), pp. 48–56.

[68] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution". In: *USENIX Security Symposium*. 2018.

[69] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic". In: *CCS*. 2018.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *SysTEX*. 2017.

[71] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling your secrets without page faults: Stealthy page Table-Based attacks on enclaved execution". In: *USENIX Security Symposium*. 2017.

[72] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue in-flight data load". In: *S&P*. 2019.

[73] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV". In: *DIMVA*. 2023.

[74] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. "Single trace attack against RSA key generation in Intel SGX SSL". In: *AsiaCCS*. 2018.

[75] Joseph Werle. *GitHub – jwerle/b64.c: Base64 encode/decode*. 2023. URL: `https://github.com/jwerle/b64.c`.

[76] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "The severest of them all: Inference attacks against secure virtual enclaves". In: *AsiaCCS*. 2019.

[77] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. "Sevurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions". In: *S&P*. 2020.

[78] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. "SEV-Step A Single-Stepping Framework for AMD-SEV". In: *CHES* (2023).

[79] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. "undeserved trust: Exploiting permutation-agnostic remote attestation". In: *Security and Privacy Workshops*. 2021.

[80] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". In: *S&P*. 2015.

[81] Yuval Yarom and Katrina Falkner. "FLUSH+ RELOAD: A high resolution, low noise, l3 cache Side-Channel attack". In: *USENIX Security Symposium*. 2014.

[82] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. "CacheWarp: Software-based Fault Injection using Selective State Reset". In: *USENIX Security Symposium*. 2023.

Appendix A.
Code listings

```
/**
 * __svm_sev_es_vcpu_run - Run a SEV-ES vCPU via a transition to SVM
    guest mode
 * @vmcb_pa:  unsigned long
 */
SYM_FUNC_START(__svm_sev_es_vcpu_run)
  push %_ASM_BP
#ifdef CONFIG_X86_64
  push %r15
  push %r14
  push %r13
  push %r12
#else
  push %edi
  push %esi
#endif
  push %_ASM_BX

  /* Move @vmcb to RAX. */
  mov %_ASM_ARG1, %_ASM_AX

  /* Enter guest mode */
  sti

1:  vmrun %_ASM_AX

2:  cli

  pop %_ASM_BX

#ifdef CONFIG_X86_64
  pop %r12
  pop %r13
  pop %r14
  pop %r15
#else
  pop %esi
  pop %edi
#endif
  pop %_ASM_BP
  RET

3:  cmpb $0, kvm_rebooting
  jne 2b
  ud2
  _ASM_EXTABLE(1b, 3b)
SYM_FUNC_END(__svm_sev_es_vcpu_run)
```

Listing A.1: Linux 5.19 guest entry code (truncated).