UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Injection Attacks on Secure Encrypted Virtualization

Injektionsangriffe auf Secure Encrypted Virtualization

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Luca Wilke

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Jan Wichelmann

Lübeck, den 9ten Januar 2020

Abstract

Data privacy concerns are one of the main aspects that stops customers from making use of cloud hosted applications, since the hosting provider can inspect the content of the virtual machines (VMs) used for hosting them. File- or disk encryption solutions only partially solve this problem, as the data remains unprotected when it is at use. The most prevalent solution to this problem is AMD's Secure Encrypted Virtualization (SEV), which allows to encrypt the RAM of VMs with a key, that is not accessible to the hosting provider. This is achieved, by managing it in a separate, trustworthy hardware component called Secure Processor (SP). The latest extension SEV-ES also protects the Virtual Machine Control Block (VMCB), that contains the state of the VM. This way the hosting provider can neither learn the content, nor any relevant part of the state of the VM.

Several previous works demonstrated how to move data into or out of the VM, by manipulating I/O operations. However, the I/O manipulation makes them vulnerable to detection and is a known problem to AMD. In this thesis we show how to inject arbitrary code into SEV secured VMs, simply by copying ciphertext blocks, that are already inside the VM, to new locations. This allows us to completely take over the VM without relying on any I/O, which makes our attack very steatlhy. We identified SEV's missing integrity protection as the main cause for this vulnerability.

In an independent result, we discovered that AMD Epyc 3xx1 CPUs use a new, enhanced encryption mode and show that it is still vulnerable to our attacks.

Kurzfassung

Datenschutz ist einer der Hauptgründe, der Kunden davon abhält Clound-Anwendungen zu nutzen, da der Hosting-Anbieter den Inhalt der virtuellen Maschinen (VMs), die zum hosten dieser genutzt werden, inspizieren kann. Datei- oder Festplattenveschlüsselungs Verfahren lösen dieses Problem nur teilweise, da die Daten weiterhin ungeschützt sind, wenn sie sich in Verwendung befinden. Die am meisten verbreitetste Lösung für dieses Problem ist AMDs Secure Encrypted Virtualization (SEV), welche es erlaubt den RAM von VMs mit einem, für den Hosting-Anbieter nicht lesbaren, Schlüssel zu verschlüsseln. Dies wird dadurch erreicht, dass der Schlüssel von einer separaten, vertrauenswürdigen Hardwarekomponente verwaltet wird. Die neueste Erweiterung, SEV-ES, sichert außerdem den Virtual Machine Control Block (VMCB) ab, der den Zustand der VM beinhaltet. Auf diese Weise kann der Hosting-Anbieter weder den Inhalt noch relevante Details über den Zustand der VM lernen.

Mehrere vorangegangene Arbeiten haben gezeigt, wie man durch Manipulation von I/O-Operationen Daten in die VM kopieren kann. Diese I/O Manipulationen machen es jedoch möglich, solche Angriffe festzustellen und sind ein AMD bekanntes Problem. In dieser Arbeit zeigen wir, wie man beliebigen Code in mit SEV gesicherte VMs injizieren

kann, in dem man Ciphertext-Blöcke, die sich bereits in der VM befinden, an neue Positionen kopiert. Dies erlaubt uns, die VM vollständig zu übernehmen ohne von I/O abzuhängen, was es sehr schwierig macht unseren Angriff zu detektieren. Wir haben SEVs fehlenden Integritätsschutz als die Hauptursache für diese Schwachstelle identifiziert. In einem separatem Ergebnis, haben wir entdeckt, dass AMD Epyc 3xx1 CPUs einen neuen, verbesserten Verschlüsselungsmodus verwenden und zeigen, dass auch dieser für unsere Angriffe anfällig ist.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, den 9ten Januar 2020

Acknowledgements

I want to thank Prof. Thomas Eisenbarth and Jan Wichelmann for their excellent supervision of this thesis. In addition, I want to thank Mathias Morbitzer for fruitful discussions.

Contents

1 Introduction **1**

2 Background **5**

2.1 Virtualization Technology 5

2.2 The Page Fault Side Channel 10

2.3 Memory Encryption using Tweakable Block Ciphers 11

2.4 AMD's Memory Encryption Solutions 13

3 Related Work **17**

3.1 Previous Attacks 17

3.3 Intel's Memory Encryption Solutions 22

4 Reverse Engineering the Encryption Mode **25**

4.1 Analysis of AMD Ryzen and AMD Epyc 7xx1 CPUs 25

4.2 Updated Encryption Mode for newer Epyc 3xx1 CPUs 28

5 Fault Injection Attacks **31**

5.2 Tracking Guest Execution 31

5.3 Placing Partially Controlled Plaintext 32

5.4 Code Injection 33

5.5 Executing Arbitrary Code 38

6 Comparison to Related Work **45**

7 Countermeasures **49**

8 Conclusion & Outlook **55**

References **57**

1 Introduction

In the modern IT infrastructure, more and more services no longer run locally but, on remote servers, to make them available to more people at once and allow for easier management. Furthermore, to make better use of the hardware, one server usually runs multiple services. From a security point of view, this is problematic because a vulnerability in one of the services may also affect the other services, as they run on the same system. Thus, every new service reduces the security of the system as a whole. Another problem of this approach is, that the computing power required by a service, like a webshop, often shows high fluctuation, while setting up new physical hardware requires some time. To cope with that, a certain amount of resource over-commitment is needed to handle peaks, that remains unused during periods of low load and thus creates unnecessary costs.
Virtual Machines (VMs) are a very popular solution to these problems. The physical server runs a software called *hypervisor*, that allows the creation of multiple VMs on the same server, which can be treated as independent computers from the user's point of view. The hypervisor provides one-sided isolation, as it prohibits software running inside the VM from accessing the host system itself or other VMs. This way, a security breach in one of the VMs does not have a severe impact on other components running on the same host.
Since VMs have a near-instant setup time, load fluctuations can easily be coped with by increasing or decreasing the number of VMs used for running the service. Instead of renting or buying hardware to host VMs, the customer directly rents VMs from a hosting provider, with enough hardware resources to allow the creation of new VMs at any time. These two properties make hosting services with rented VMs a very popular use-case in the current server market.
As mentioned earlier, the hypervisor only provides one-sided isolation, i.e. the VM cannot access the host or other VMs while the hypervisor has full control over the VM's hardware and thus can arbitrarily read from/write to the RAM used by the VM. Thus the hypervisor can spy on sensitive data in the VM or manipulate the code of loaded programs to take over the VM. It is very hard for the VM to detect such inspections/manipulations; it simply has to trust the hosting provider. That's why potential customers still cite data privacy concerns toward cloud service providers as the main reason not to adopt such solutions, especially in cases where the hosting location within a given jurisdiction cannot be guaranteed. Security solutions like full disk encryption only partially address this issue, since data is still vulnerable when being decrypted and stored in the RAM at run time.

There are two approaches to this problem: Homomorphic encryption and hardware-based security solutions that prohibit the hosting provider from accessing the users' data and are able to prove this to the customer.

Homomorphic encryption [GB09] allows to perform computations directly on encrypted data. Instead of uploading data in plaintext, the user encrypts it with such a scheme before uploading it. This way, the service processing the data never learns the actual content, which in turn prohibits the hypervisor from spying on it. However, these schemes are quite complex and have a high-performance overhead for general computations [AAUC18].

Hardware-based security solutions for providing full isolation between the hypervisor and the VM have been studied extensively by researchers as well as industry [AGJS13, KPW16, Kap17, JACH11, XLC13, Int19]. The general idea is to use a hardware component with higher privilege levels than the hypervisor or the host Operating System (OS) which enables it to protect the data via access right restrictions or cryptography. The hardware component itself proves its authenticity via asymmetric cryptography. This way the customer only has to trust the hardware manufacturer, which he has to do regardless of using a VM or his own dedicated hardware.

Intel Software Guard Extensions (SGX) [AGJS13, HLP+13, MAB+13] was the first widely available solution for protecting data in RAM. However, it only can protect a small chunk of RAM, not the VM as a whole [Gue16]. In 2016, AMD introduced Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) [KPW16] to protect the entire system's main memory. SME provides drop-in, AES based RAM encryption. SEV extends this for VMs by using different encryption keys per VM, in order to prohibit the hypervisor from inspecting the VM's main memory. This was a first step towards full isolation. The Linux kernel support for SEV was mainlined in early 2018 [AMD]. In February 2017, AMD introduced SEV Encrypted State (SEV-ES) [Kap17], which offers additional protection against manipulating the state of a VM during context switches. While SEV-ES does not need new hardware, it requires extensive modification of the Linux kernel. According to AMD, the corresponding patches to the Linux kernel are mostly finished, however support for SEV-ES has not been mainlined, yet [Len19]. Intel is also working on a solution similar to SME/SEV, called Total Memory Encryption (TME)/Multi-Key Total Memory Encryption (MKTME) [Int19], but did not yet publish corresponding processors.

This thesis focuses on AMD's solutions for providing full isolation between hypervisor and VM, as it is the most prevalent full memory encryption. Prior attacks still working under SEV-ES either used a firmware bug [BWS19] or I/O to move known plaintext into encrypted pages or extract encrypted data [DYM+17, MHHW18, MHH19, LZL19]. However, the I/O channel is well known problem to AMD. We demonstrate that our attack

vector is available even without user-controlled I/O or access to unprotected I/O operations.
This thesis is structured as follows. First, we explain the current state of virtualization technology, review the official information on AMD's memory encryption and discuss related work. Next, in Chapter 4, we analyze the encryption mode used by SME/SEV as there is very little official information on it. We show, that the AMD Epyc 3xx1 product line uses an updated encryption mode and prove, that it is still vulnerable to the previous attacks.
Using this knowledge, we demonstrate in Chapter 5, that minimal information about the system is enough to compromise and completely take over the VM. To achieve this, we exploit some weaknesses of the encryption mode, to construct a primitive that allows us to inject data into the encrypted RAM of the VM, simply by copying ciphertext blocks with known plaintext to a new memory location. However, in the basic variant this only allows us to control 4 plaintext bytes of every two continuous 16-byte ciphertext blocks. Thus, in the next step we build on this primitive to bootstrap an encryption oracle from just a few megabytes of known plaintext, that is able to control full 16-byte blocks, allowing us to place and execute arbitrary code in the VM. In addition, we present that code execution can be used to build a decryption oracle.
Finally, we compare our results to related work and discuss possible countermeasures. We identify the lack of integrity protection as the main weakness of AMD's memory encryption solutions and postulate that full security against our attacks can only be achieved by integrating a proper integrity protection scheme into the currently used encryption mode.

2 Background

To understand the requirements for fully isolating hypervisor and VM, we first need to understand how they interact with each other. Thus this section starts with an introduction to virtualization technology. While Intel's and AMD's approaches to virtualization share many similar concepts, we will focus on AMD's variant when it comes to the details, as this thesis analyzes the AMD exclusive features SME/SEV. Next, we will briefly explore theoretical considerations for memory encryption, before reviewing the official information on AMD's memory encryption technologies.

2.1 Virtualization Technology

As discussed in [TB15, p. 471 et seq.], the goal of virtualization is to to create and run multiple VMs on a single physical computer in such a way, that they are isolated from each other. As mentioned in the introduction, this has multiple advantages like improved security when hosting multiple services, reduced hardware costs and easier load balancing. The VMs are created and managed by a software called *hypervisor*. In order to isolate VMs from each other and from the hypervisor itself, the hypervisor emulates a set of virtual hardware components for each VM. On a technical level, this is achieved by intercepting and manipulating certain operations issued by the VM. We will call this one-sided isolation, as it does not isolate the VM from the hypervisor, but isolates the hypervisor from its VMs, as well as the VMs from each other. In the following, we are going to explore how to virtualize the CPU, memory accesses and accesses to I/O devices.

2.1.1 Virtualizing the CPU

First of all, the hypervisor has to emulate a CPU to allow its guests to execute code, which boils down to executing assembly instructions. However, the hypervisor must prevent its guests from executing instructions that would allow it to manipulate the hypervisor or other VMs. AMD CPUs support a hierarchical privilege-levels concept [Adv19, p. 96], ranging from the most privileged level *ring 0* to the least privileged level *ring 3*. Some assembly instructions, so-called *privileged instructions*, can only be executed in ring 0. The host OS/hypervisor runs at ring 0, while user software usually runs in ring 3.
The hypervisor must run at a higher privilege level than its VMs, to be able to isolate itself from them. If a privileged instruction is executed in a ring different from ring 0, it causes

a `trap` which transfers the control flow to a handler function of the host OS/hypervisor. This handler is provided with certain information like the process that tried to execute the privileged instruction. Based on this it can decide if the instruction should actually get executed or not. Afterwards, the control flow is restored to the calling application. In the non-virtualization context, this mechanism allows the OS to enforce privilege separation. Similar, the hypervisor can use this mechanism to isolate itself from the VM as well as different VMs from each other, by restricting or emulating certain instructions issued by the VM. This is also called instruction interception.

Unfortunately, on x86 some instructions behave differently based on the ring that they are executed in, without causing a trap like privileged instructions. If we virtualize a software, like an OS, that would usually run in ring 0, this causes unexpected behavior, as the instruction is now executed at a different privilege level. As each VM must run an OS, this is very problematic. Early approaches to virtualization solved this by scanning the assembly code executed in VMs for such instruction and replaced them with a semantically equivalent instruction sequence causing a trap. The downside of this approach is that it introduces a performance overhead.

To overcome these problems, in 2005 both Intel and AMD introduced an instruction set extension to enable hardware-assisted virtualization. AMD's solution is called Secure Virtual Machine (SVM) [Adv05] and Intel's solution is called VT-x [UNR+05].

SVM adds two additional privilege layers called *host mode* and *guest mode*, as well as new instructions for switching between them. Each mode has ring 0 to 3, as before. Thus virtualized applications can again run in their intended ring nullifying the problem of instructions that behave differently depending on the ring they are executed in. However, in guest mode, certain instructions are forbidden/restricted. If an exception, interrupt or trap occurs while in guest mode, the CPU performs a context switch to the hypervisor. Figure 2.1 contains a schematic overview of the rings, modes and the control flow transfer between them.

In order to start a VM, the host executes the `VMRUN` instruction with a reference to the Virtual Machine Control Block (VMCB) of the VM. This instruction first saves the CPU state information of the host, before loading the VM's state from the VMCB and switching the CPU to guest mode. Afterwards, the CPU runs the VM in guest mode until exceptions, interrupts or traps occur, triggering a `#VMEXIT`. If this happens, the CPU writes back the current state of the CPU to the VMCB, switches the CPU to host mode at privilege level zero and resume the execution at the saved host state. The VMCB primarily contains the state of the CPU at the last time the VM was executed. This especially includes register values. In addition to instructions that must trap to the hypervisor in order to allow proper functioning of the VM, SVM also allows configuring certain other instructions to

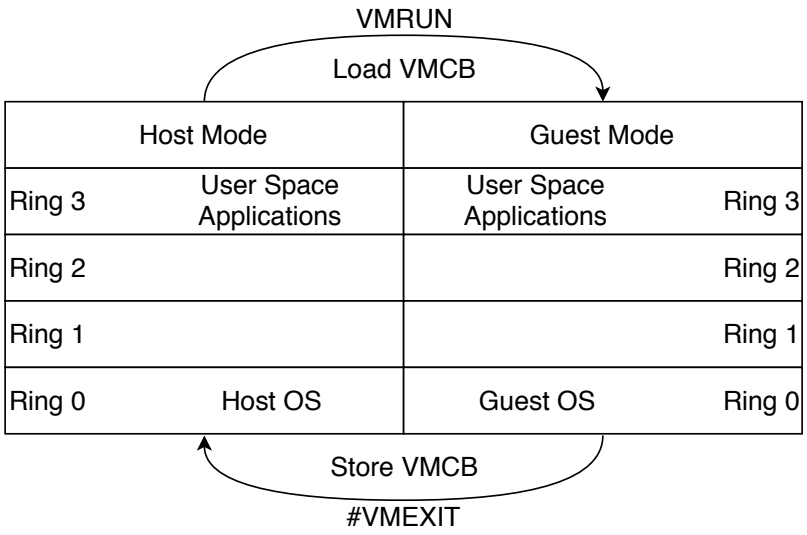


Figure 2.1: Schematic overview of CPU rings and modes used for virtualization on the AMD x86 architecture. Switching between modes is done via the `VMRUN` instruction and the `#VMEXIT` exception. Ring 0 has the highest rights. Rings 1 and 2 are seldomly used.

trap, allowing the hypervisor to emulate them. Two prominent examples for this are the `cpuid` and the `rdtsc` instructions. The `cpuid` instruction allows querying a wide span of CPU information, including an accurate model number, a list supported features and the system topology. Trapping this instruction, allows the hypervisor to change the returned registers values, allowing it fine-grained control over the hardware features it exposes to the guest. The `rdtsc` instruction returns the current state of the CPU core-private timestamp counter. OSs and other programs may use this counter for cycle-level time measurements. If a VM is live-migrated from one host to another, the `cpuid` and `rdtsc` values on both machines might be different. Emulating these instructions allows the hypervisor to convey a consistent picture of the system state. Whether such instructions are intercepted or not, can be configured in the VMCB [Adv19].

In summary, handling instructions executed in the guest, that could potentially break the isolation between hypervisor and VMs is the most important part of CPU virtualization. To achieve isolation, the hypervisor can intercept such instructions before they are executed, which allows it to perform additional emulation. Next, we will discuss the virtualization of memory accesses.

2.1.2 Memory Accesses

Before we explore the virtualization of memory accesses, we give a brief overview of how memory accesses work on a non-virtualized system.
As discussed in [TB15], on modern OSs, like Windows, Linux and MacOS, processes do not directly address the physical memory to access data. Instead, an additional indirection layer called Virtual Address (VA) is used. If a program accesses a VA, the CPU tries to translate it to a Physical Address (PA) via a data structure called page table. Furthermore, this data structure also contains additional information like access rights and various status bits for each entry. In order to speed up the translation, page table entries do not refer to a single byte of physical memory, but to a larger chunk of memory called a *page*, which usually has a size of 4KB. To achieve this, only the upper bits of a VA are used to translate it to a PA, while the lower 12 bits are used as an offset inside the 4KB page without any translation. In addition, the results of page table lookups are stored in a cache called Translation Lookaside Buffer (TLB), speeding up subsequent accesses. Furthermore, the actual translation process is done in hardware. The PA of the top-level page table is stored in a well-known hardware register and is managed by the OS.
VAs allow for more flexible addressing, as they enable the OS to create and manage different page tables for each process. This way, each process has its own address space and can use the memory, as if it is the only process running on the system. For example, two processes A and B can both manipulate the content of the VA 0x1000 without seeing each other's results, as the OS takes care of mapping these VAs to different PAs. This also greatly improves the security and stability of the system, as process A cannot access the memory of process B without requesting the OS to add the respective mappings to its page table. If a program accesses a VA, for which it does not have a mapping in its page table, the hardware issues a *page fault*, which gets processed by a handler in the OS. The OS can then decide, whether it wants to allow or deny this access. In the former case, it creates a new mapping in the page table and re-executes the instruction that caused the page fault. In the latter case, the program that caused the page fault gets aborted.
Virtualizing memory accesses is more difficult, as the VM itself also has to run an OS that expects to manages its own top-level page table. However, the hypervisor clearly cannot allow the VM to create entries for arbitrary physical memory pages in its page table, as this would allow the VM to read from/write to memory pages belonging to the hypervisor or other VMs running on the physical machines. This breaks the one-sided isolation we want to achieve. To overcome this problem, an additional layer of indirection is introduced. The page table of the VM only translate Guest Virtual Addresses (GVAs) into Guest Physical Addresses (GPAs), which are then translated to the actual PA, also known as Host Physical Addresses (HPAs) in this context. The translation of GPAs to

HPAs is done via an additional data structure called Nested Page Table (NPT) [Adv08]. The NPT is managed by the hypervisor, allowing it to control which physical pages are used for the memory requests of the VM. This process is completely transparent for the VM, i.e. it does not know that its GPAs do not actually point to physical memory. In the context of virtualization, AMD tags the TLB entries with a so-called Address Space Identifier (ASID) in order to allow the hardware to distinguish between entries originating from different VMs or the host [Adv05]. This way, the TLB does not need to be flushed before performing a context switch between the hypervisor and theVM, which improves the performance.

In summary, the use of NPTs ensures that the hypervisor has full control over the physical memory pages that a VM can use. Therefore, the hypervisor can ensure that the VM cannot read any memory that is used by the hypervisor itself or other VMs, if the hypervisor does not explicitly allow it.

2.1.3 I/O Devices

Finally, we want to discuss how VMs access I/O devices like network cards or hard drives. According to [TB15, p. 490 et seq.], the VM probes for connected I/O devices on startup. These instructions are trapped to the hypervisor, which can then decide whether it wants to expose a device to the VM or not. Interfacing with hardware components can be done via hardware control registers. The instructions for accessing these register, also trap to the hypervisor, which then can decide, whether it wants to copy the data to/from the real hardware registers or not. This also allows the hypervisor to emulate devices that are not physically attached to it. A classical example for this are hard drives. The operating systems running in the VM expect access to a hard drive partition. However, actually creating partitions on the real hard drive that is attached to the physical machine is cumbersome. Instead, the hypervisor stores all of the VM's contents in a regular file on and only emulates to the VM, that it accesses a real hard drive.

Another important aspect are DMA operations, which allow hardware components to read from/write to the RAM without using the CPU, which greatly reduces the system load. On modern x86 systems, a hardware component called I/O Memory Management Unit (IOMMU) is responsible for mapping DMA capable hardware components into the address space used by the OS. In order to support virtualization, the hypervisor can program the IOMMU to use the NPT of the VM to translate the GPAs of the VM into HPA before performing memory accesses. Directly using the GPA as a PA would break the isolation, as the VM could exploit the IOMMU to access memory areas, that were not assigned to it by the hypervisor.

The preceding sections have shown, that the one-sided isolation of the hypervisor from its VMs rests on two main pillars. The first is, that the hypervisor runs in a higher privilege mode on the physical CPU and thus can prevent its VMs from breaking out of their lower privilege level, by restricting the use of certain instructions. The second pillar is, that the hypervisor controls the physical memory which can be accessed by its VM via the NPT. This way, a malicious VM is unable to modify the hypervisor itself or other VMs running on the same physical machine. However, this one-sided isolation does not protect the VMs against a malicious hypervisor. This is mainly due to the fact, that the hypervisor can manipulate the VM's RAM content, as it has control over all of the physical memory. Therefore, the hypervisor can manipulate the code executed in the VM and extract sensitive data when it gets loaded into the RAM for processing. As the VM cannot detect or prevent such accesses, it simply has to trust, that the hypervisor respects its privacy. This may be acceptable in traditional use-cases, where a company uses its own physical machine, which is located at their premises, to run multiple VMs. However, as already noted in the introduction, in modern use-cases companies or persons rent VMs from third-party providers. Thus they do not have any control over the hypervisor or the physical machine and have to trust the hosting provider to not spy on their VM. As we will see in Section 2.4, AMD's SEV technology tries to achieve full isolation between VM and hypervisor, by encrypting the memory content of the VM with a key, that is not known to the hypervisor.

2.2 The Page Fault Side Channel

The page fault side channel was first explored in the context of Intel SGX [XCP15]. When SEV is active, the page table of the guest is encrypted and thus not accessible by the hypervisor. However, as the hypervisor is responsible for managing the NPT, it can infer information about the VM's memory assignment by monitoring the entries in the NPT. Since the NPT cannot be accessed by the VM, it cannot prevent the hypervisor from overwriting permissions in the NPT. A schematic overview of how the hypervisor can exploit this is shown in Figure 2.2.

As discussed in Subsection 2.1.2, the VM contains the GVA to GPA page table while the host manages the GPA to HPA translation in the NPT. Both page tables contain status bits that control whether a page is present, writeable or executable. However, the hardware only honors the status bits from the NPT, triggering a page fault on illegal accesses. If the VM causes a page fault, the corresponding handler in the hypervisor is called. It is provided with the GPA of the fault as well as the reason for the fault in order to handle it. A malicious hypervisor can manipulate the status bits in the NPT to get notified if the

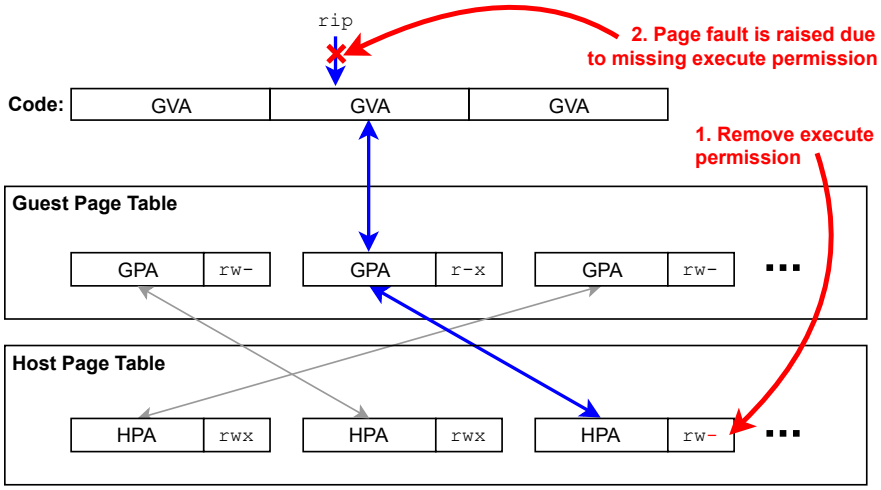


Figure 2.2: Schematic overview of the page fault side channel. When the VM tries to execute an instruction, the GVA to which the program counter (`rip`) points, has to be resolved to a HPA. This is accomplished by performing a walk through the NPT, while checking the respective permission flags. The hypervisor can force a page fault by removing the execute flag, in the corresponding NPT entry. Subsequently, the hypervisor learns which page the VM tried to execute. The same method can be used for detecting memory reads or writes by clearing the respective flags instead.

VM tries to read, write, or execute a page. Multiple attacks make use of this possibility to gather information about where the VM stores critical data [HB17, MHHW18, BHP18, LZL19, MHH19].

2.3 Memory Encryption using Tweakable Block Ciphers

As the name suggests, block ciphers only operate on a fixed-length data block. However, the data that should be encrypted/decrypted is usually larger than one block and not a multiple of the block size. The latter can be solved with padding or ciphertext stealing [Aum17, p. 69-70]. The easiest way to encrypt multiple blocks of data is the Electronic Codebook Mode (ECB) mode, which simply applies the cipher independently to each block. This method is very fast, as blocks can be encrypted and decrypted in parallel, but has various weaknesses like the fact that equal plaintext blocks encrypt to the same ciphertext blocks, allowing to recover information on the plaintext.
Other block modes overcome this problem by making the encryption of a plaintext block depend on other plaintext or ciphertext blocks, like discussed in [Aum17, p. 65 et seq.]. For example, in the Cipher Block Chaining (CBC) mode the plaintext of block i gets XORed with the ciphertext of block $(i-1)$ before encryption. While this mitigates the problem of

equal plaintexts encrypting to the same ciphertexts (or more general: ciphertext frequency analysis), it also prevents parallelizing the encryption process. However, applications like encrypting the RAM or hard drives require very fast encryption and decryption, as any introduced latency slows down the whole system.

Another approach to this problem is using Counter Mode (CTR) mode, which converts the block cipher into a stream cipher, by using it to encrypt a counter value and XORing the result with the plaintext. Using a good block cipher like AES, the encryption of counter values i and $i + 1$ appear independent. In addition, encryption as well as decryption are fully parallelizable and, if no counter value is used twice, the encrypted blocks appear independent regardless of the plaintext.

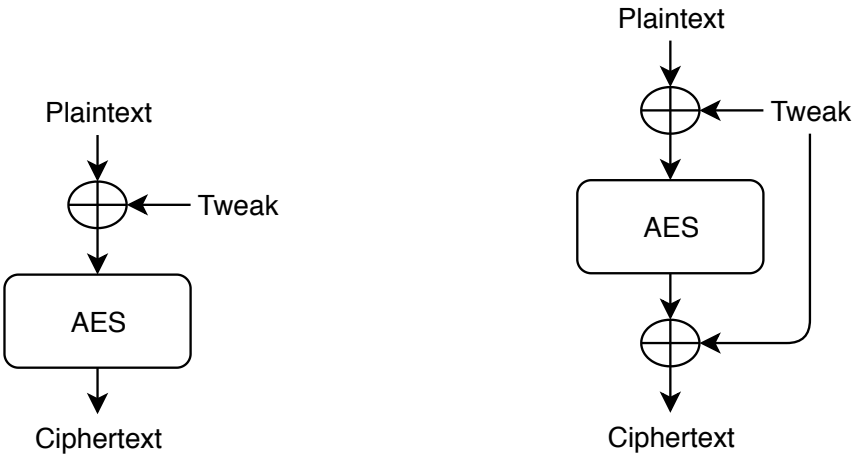
However, CTR mode is not well suited for RAM encryption or full disk encryption: In order to prevent using the same counter value twice the latest counter value has to be stored for each memory location/disk sector. In addition, flipping one ciphertext bit only flips the corresponding plaintext bit, instead of changing the whole block in an unpredictable manner, like it is the case for block ciphers. An attacker with some knowledge of the system could use this for directed manipulations, e.g. to flip an access right bit in a kernel data structure.

Tweakable block ciphers, on the other hand, provide full parallelization for encryption and decryption without the need to store a counter value or risking ciphertext frequency analysis. In addition to the key, the ciphertext produced by a tweakable block cipher is also influenced by a so-called *tweak*, that allows to securely change the behavior of the block cipher, similar to instantiating it with a new key, but with less overhead. Unlike the key, the tweak does not need to be secret [LRW02].

If the ECB style encryption is used with a different tweak for each block, equal plaintext blocks no longer encrypt to equal ciphertext blocks. Instead, the blocks appear as if they were independent. Nonetheless, without proper integrity protection several attacks remain possible, like randomizing the plaintext (by altering the ciphertext) and replaying old values.

AES-XTS [XTS19] is a tweaked version of AES, that is very popular for storage encryption, as it includes ciphertext stealing, which allows expansion-free block encryption for arbitrary-length plaintexts by using previous ciphertext for padding. The tweak is usually a function of the logical disk block address. AES-XTS is used in Apple's FileVault, MS Bitlocker and Android's File-based Encryption.

However, the RAM and the uncore part of the CPU (components that are not part of the CPU core but closely connected to it like the L3 cache) are always handled in 32 or 64 byte blocks which are multiples of the 16-byte block size of the AES [Adv17]. Thus, RAM encryption does not require padding or ciphertext stealing. Rogaway [Rog04] introduces



(a) XE mode (b) XEX mode

Figure 2.3: Flowchart of the XE and XEX construction, that turns a regular block cipher like AES, into a tweakable block cipher.

the Xor-Encrypt (XE) and the Xor-Encrypt-Xor (XEX) modes, which turn a block cipher such as AES into a tweakable block cipher, by XORing a tweak-derived value with the plaintext before encryption (and again after encryption in the case of XEX), as shown in Figure 2.3.
Rogaway proposes special tweak functions for XE and XEX, which allow very efficient computation, especially if the tweak function is evaluated in a sequential manner, e.g. multiple consecutive disk sectors or consecutive memory addresses are encrypted/decrypted.

2.4 AMD's Memory Encryption Solutions

Currently AMD offers three main memory encryption solutions, that build on each other This section presents them in a hierarchical order. Figure 2.4 contains a schematic overview of the encryption/decryption process as well as the interaction between the different components.

2.4.1 AMD SME

AMD Secure Memory Encryption (SME) was first introduced in a whitepaper in 2016 [KPW16]. It is a hardware-based security feature, which allows encrypting data before it is stored in RAM in order to guard against Direct Memory Access (DMA) or cold boot attacks [HSH+08, BGF16, YADA17]. According to [KPW16], it works as follows: The mem-

ory controller includes an AES encryption module to allow high-speed encryption/decryption. However, the used block mode is not specified by AMD. They claim to apply an address based tweak to the plaintext to protect against "cipher-text block move attacks".
The key K for the encryption is managed by the AMD Secure Processor (SP), an ARM based co-processor. It is regenerated during the boot process, using a NIST SP 800-90 compliant random number generator. The key is never stored outside the SP. Due to this, it cannot be recovered by cold boot or DMA attacks.
To allow more flexibility, SME uses a special bit in the page table entries – the so-called C-bit – to encode whether a page should be treated as encrypted or not. For example, this can be used to only encrypt memory locations containing secrets like full disk encryption or RSA keys.
However, changing the C-bit does not perform in-place en- or decryption; i.e., to encrypt the data inside an initially unencrypted page, the data has to be rewritten after changing the C-bit. There is no coherency between mappings of the same memory location with different C-bit values or different encryption keys. Thus, changing the encryption status requires flushing all involved CPU caches, to write back and invalidate all cached entries. Otherwise, the next access might return a cached result, that does not reflect the new encryption status [Adv19].
Managing the C-bit requires support from the OS. In case the OS does not support SME, Transparent SME (TSME) can be used: In TSME mode, all memory pages are encrypted independently from the value of the C-Bit.

2.4.2 AMD SEV

AMD SEV, which was also introduced in [KPW16], builds on SME, to provide full cryptographic isolation between the hypervisor and the VM. This means that the VM owner does not have to trust the hypervisor.
To achieve this, each VM is assigned an encryption key by the SP, which is used to encrypt the VM's main memory with the SME technology. The mapping from VM to encryption key is based on the ASID of the VM. Like it is the case for SME, the key never leaves the SP. Thus, even an attacker with hypervisor privileges is unable to obtain the key.
Like with SME, the VM has page-wise control over the RAM encryption via the C-bit. If the C-bit is set, the page is encrypted with the VM-specific key; else, the key of the hypervisor is used. This allows the VM to set up shared pages (pages accessible by the VM and the hypervisor).
The SP offers an API to manage SEV secured VMs [Adv18]. This is used by the hypervisor, to perform operations requiring knowledge of the key, like encrypting the initial RAM content during VM startup. To prove to the VM owner, that the hypervisor has not

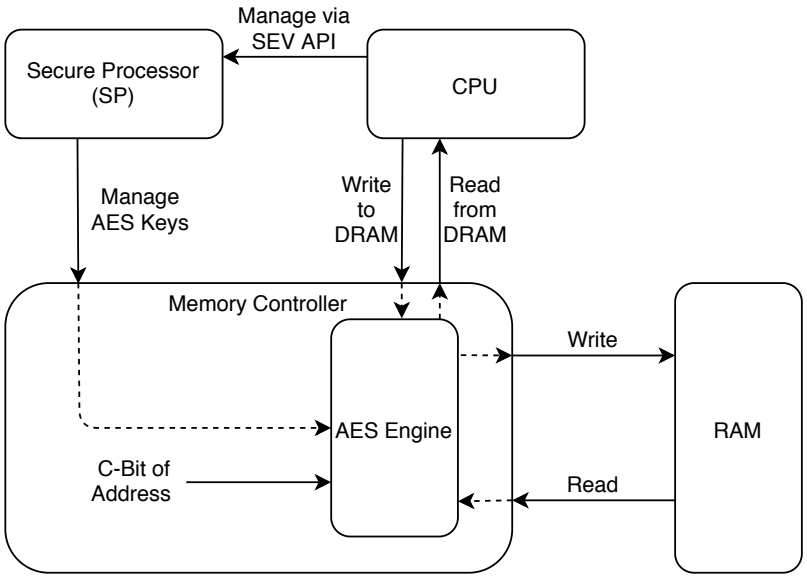


Figure 2.4: Schematic overview of the encryption and decryption process of SME/SEV. Memory accesses are routed through a hardware-based AES engine. However, data is only encrypted/decrypted if the C-Bit of its address is set. Key management is handled by the SP. For SEV, the SP exposes an API allowing the CPU to to perform certain key-related operations without exposing the key to the CPU. For example, the CPU can request that a new key should be created in order to start a new SEV secured VM.

tampered with the initial content before loading it via the SP API, the owner can request a signed hash of the initial RAM content of his VM from the SP. For the signature the SP uses an asymmetric key pair, that is signed by AMD, to prove its authenticity.
In addition, the asymmetric key pair can also be used to establish a secure channel between the VM owner and the SP. This way, the VM owner can transfer secrets, like full disk encryption keys or OpenSSH private keys, into the VM in a secure manner.

2.4.3 AMD-ES

A known problem of SEV is, that it does not protect the VMCB before switching from the VM's context to the hypervisor's context. The VMCB describes the state of a VM, like the values of the VM's general purpose registers. It is also used for communication between the hypervisor and the VM. For example, if the VM exits due to an interrupt it can store additional information about the interrupt reasons in this structure.
To address these issues, AMD introduced SEV-ES [Kap17] as an extension for SEV. SEV-ES splits the VMCB into two areas: The control area and the save area. The unencrypted control area contains information that must always be available to the hypervisor in order

to manage the VM, e.g. flags for interrupt injection. The save area contains all of the other information from the VMCB, and is protected against access or manipulation from the hypervisor by encrypting it when the VM exits. However, since certain operations require the VM to share data from its save area with the hypervisor (e.g. reading and writing certain registers when emulating `cpuid`), AMD introduced the Guest Hypervisor Communication Block (GHCB), which basically is a shared page, allowing communication between guest and hypervisor. In addition, they introduced a new exception that gets triggered by operations that require the VM to share information with the hypervisor, allowing the guest to copy the required data from the VMCB to the GHCB before the context switch. When the VM is resumed, it can copy the data back to its VMCB.

3 Related Work

This chapter is split into two sections. The first one discusses existing attacks on AMD's memory encryption technology. The second section provides a summary of Intel's memory encryption solutions, in order to provide an overview of current technologies for solving the hypervisor trust problem on the x86 architecture.

3.1 Previous Attacks

This section summarizes already known attacks on SEV. If not stated otherwise, all attacks assume a malicious hypervisor. In addition to this, Chapter 6 contains a detailed comparison of our results with the ones from Du et al. [DYM+17] and Li et al. [LZL19], as both of them construct encryption oracles.

3.2 VMCB based Encryption/Decryption Oracles

Hetzelt and Buhren [HB17] explore the idea of manipulating the general purpose registers stored in the VMCB to create an encryption/decryption oracle. They leverage, that prior to SEV-ES, the VMCB was stored unencrypted and without integrity protection during context switches between hypervisor and VM. Therefore, they can manipulate the VM's register values, that are stored in the VMCB.
However, in order to create an encryption oracle, they have to make sure that the register values are written to RAM when the VM resumes. To achieve this, they construct corresponding gadgets via a ROP style attack. Instead of manipulating return addresses on the stack, they directly manipulate the unprotected value of the `rip` register containing the GVA of the next instruction that gets executed. Similarly, they construct a decryption oracle by constructing gadgets that load data from memory into general-purpose registers before forcing a context switch.

3.2.1 NPT Manipulation

Hetzelt and Buhren [HB17] also constructed an attack that leverages the page fault side channel to launch a replay attack against an OpenSSH server running inside the VM. During the login process, OpenSSH stores the user's password in RAM for a short period of time, to perform the authentication. They show, that an attacker can identify this memory

location and create a copy of it before it gets overwritten for security reasons. Afterwards, the attacker starts a login process via SSH with the user account, whose password was just captured and replays the captured page to pass the password check.
Since the memory is encrypted, the attacker cannot deduce the correct location of the page containing the password from the memory content itself. Furthermore, with SEV-ES he is also unable to observe the state of an application running inside the VM via the data from the VMCB, as it is encrypted. Instead, Hetzelt and Buhren determine the right timing by observing system calls and memory accesses of the VM. The latter are immediately leaked by the page fault side channel. The former requires knowledge of the VM's kernel version and the GPA that it is loaded to. By removing the execute rights from the pages containing the system call handlers of the guest OS, the page fault side channel can be used to track the VM's system call usage.
For the replay part, the attacker has to remap the GPA where OpenSSH expects the password, to the HPA at which the password was captured, before overwriting its content. The remapping is necessary due to the HPA based tweak function. Replaying the ciphertext to another HPA changes the plaintext it decrypts to.
In their implementation, they achieved a successful replay attack in 23% of all logins attempts. They ascribe the low success rate to two circumstances. The first is, that the offset of the password inside the memory page changes during login attempts. The second is, that the correct identification of the page via the system call and read/write access patterns is only successful in 86% of the cases.
Since SEV was not available at this time, they tested their implementation on an unencrypted VM and argue that the information they used is also available on a SEV secured VM.

3.2.2 I/O based Encryption/Decryption Oracles

Morbitzer et al. [MHHW18], construct a decryption oracle based on the hypervisors ability to manipulate the NPT as well as self-generated network I/O. They require a network reachable service running in the VM, that returns some kind of resource on request. A good example for such a service is an HTTP server, serving a static web page. The basic idea is, that the attacker can manipulate the GPA to HPA mapping in the NPT, to change the content returned by the service, when the resource is requested.
The attack is split into two phases. First, the attacker has to identify the GPAs of the memory page(s) containing the data returned by the service. In order to do so, he requests the resource from the service, while monitoring the page accesses of the VM via the page fault side channel, until he receives the result of this request over the network. However, due to other processes running inside the VM, there are multiple page accesses in this time

span. To identify the page accesses related to the resource, the attacker queries the server multiple times and only marks pages occurring every time as relevant. After sufficient repetitions the set of relevant pages converges toward the page containing the resource returned by the server.

In the second step, the attacker manipulates the NPT to change the HPA to which the GPA containing the resource is mapped. Thus on the next request to the service, the data located at this HPA is returned instead of the original data. If the original resource returned by the service spans at least one page, this process can be iterated to send all of the VM's RAM content over the network. Otherwise, only the offsets of each page that were used by the original resource can be sent. Due to the address based tweak function, the attacker cannot simply copy the other parts of the page to this part to get the rest of the page's content.

Morbitzer et al. implemented the attack on an AMD Epyc 7251, which has full SEV support. For the network reachable service, they evaluated Apache, Nginx and OpenSSH. In all cases, the returned resource spanned at least a whole page. To simulate a real environment up to 50 random requests per second were simulated during the page access measurements. For Apache and Nginx the set of possible pages converged to a number of less than five in at most 22 iterations that required at most 23 seconds. For OpenSSH more than 100 iterations were required, that took at most 5 minutes.

They achieved a transfer rate of $79.4\,\mathrm{KB}\,/\,\mathrm{sec}$ in the case of Apache and Nginx and a rate of $41.6\,\mathrm{KB}\,/\,\mathrm{sec}$ for OpenSSH. In their follow up paper [MHH19], Morbitzer et al. showed how to use the page fault side channel to identify pages that are likely to contain secrets like encryption keys.

Du et al. [DYM+17] were, to the best of our knowledge, the first first to discover the original encryption mode of SME as well as the tweak values. They combine their knowledge of the tweak function with a specific memory management behavior of an Nginx server running inside the VM, to build an encryption oracle.

In the Nginx version used by them, parts of a client's request get stored in a memory page that is not overwritten after the request has been processed. They call this the bridge page. To get the GPA of the bridge page, they leverage the fact that parts of the request get stored in continuous 16-byte blocks inside the bridge page. They exploit this, by preparing the request in such a way, that the data stored at these offsets nullifies the effect of the tweak function. Thus these offsets encrypt to the same ciphertext, as if ECB was used without a tweak function, making them easily detectable in a memory dump. Furthermore, they observed that the GPA of the bridge page does not change during requests. Thus, subsequent requests can be used to encrypt arbitrary data. Furthermore, they show that the encrypted data can be moved to a different memory location, if the plaintext is prepared

in such a way that it nullifies the effect of the different tweak values used at these memory addresses. In order to demonstrate this, they overwrite parts of OpenSSH's `.text` section with code that opens a remote shell without requiring a login.

Since SEV was not available at the time, they used SME to for a self-built, simulated version of SEV. They do not provide performance measures. The attack is not mitigated by SEV-ES. Chapter 6 contains a comparison of this attack with our encryption oracle from Section 5.5.

Li et al. [LZL19] show, how to use unprotected DMA operations to create a decryption as well as an encryption oracle. According to them, VM's mainly use DMA to perform I/O operations with (virtual) hardware like network cards. As discussed in Section 2.1, this is done via a hardware component called IOMMU, which maps the I/O buses of the (virtual) hardware into memory. They observed, that this hardware unit does have support for different memory encryption keys. If the hypervisor wants to provide DMA capabilities to different VMs, the IOMMU can at most use the single memory encryption key that is used to encrypt the host's RAM. Thus, the pages used for DMA are shared between the hypervisor and the VM. If the VM wants to read from a device via DMA, the data gets copied into a shared page (a page encrypted with the shared hypervisor key) p_s. Afterwards the VM copies the data from p_s into a private page p_p (a page encrypted with its own key) before further processing the data. Writing to a device is done by the inverse process. They demonstrate their attack based on network packets related to an OpenSSH server running inside the VM, as network packets get sent via DMA.

The construction of the encryption oracle is quite similar to [DYM+17]. The attacker uses the page fault side channel to find the private page p_p to which the VM copies the data from p_s during a read operation. To encrypt arbitrary data, the attacker replaces the original data before the VM copies it from p_s to p_p. For the decryption oracle, the attacker manipulates DMA write operations issued by the VM. This time, the content of p_p is replaced with the content of the page that should be decrypted, before the VM copies the data to p_s. Again the page fault side channel is used to get the correct timing.

In their implementation, they were able to identify p_p with an average precision of 0.956 and an average recall of 0.847. They achieve a throughput of 200 B/s for their decryption oracle. They do not give performance measures for the encryption oracle, but due to the symmetric construction, its throughput should be similar. Chapter 6 contains a more detailed discussion of this attack as well as a comparison with our encryption oracle from Section 5.5.

3.2.3 Data Faults

In [BGN+17], Buhren et al. explore the idea of performing classical fault attacks on application data, by flipping a bit in a ciphertext block, in order to create garbled plaintext. They demonstrate this, by performing a fault attack on a GnuPG version without a fault resistant RSA CRT implementation. They implemented their attack for SME, as SEV was not available at that time, thus their attacker model is different. They require, that the attacker is able to run an unprivileged application and can perform DMA memory access. They used the unprivileged application to perform a prime and probe cache attack, to infer the state of the GnuPG application, allowing them to inject the fault at the correct time. In order to get the memory location containing the data used in the RSA CRT calculations, they used some Linux kernel specific memory management behavior, as well as the pagemap interface (which now requires root permissions).

3.2.4 IBS based Fingerprinting

Werner et al. [WMA+19], showed two independent results. First, they use the unencrypted VMCB to reconstruct code executed in the VM by single-stepping the VM while observing the changes to the unencrypted register values in the VMCB. Like [HB17], they also use the unencrypted VMCB to encrypt/decrypt data. This is mitigated by SEV-ES. In their second result, they show a novel approach to fingerprint applications running inside the VM. For this they leverage a performance counter subsystem called Instruction Based Sampling (IBS). This subsystem leaks information on instructions executed in the VM as well as their GVA. They show, that the distance between return statements (measured in their GVA), uniquely identifies specific versions of applications and even allows to distinguish between different compilers. One major problem they had to overcome, was that the IBS subsystem does neither distinguish between instruction executed in different applications nor between instructions executed in userspace and kernel space. Thus the returned data has to be filtered and assigned to the application they originated from, to apply the fingerprinting technique. They claim, that the guest cannot detect whether IBS is activated. This result holds under SEV-ES.

3.2.5 AMD SP security

In [BWS19], Buhren et al. explore another attack vector. They examine the security of the AMD SP, which forms the root of trust for SEV. The SP only executes signed firmware images. However, they found a bug in the signature check mechanism of the firmware, allowing them to execute manipulated firmware on the SP. While newer firmware versions fix that bug, there is no rollback prevention mechanism. Thus an attacker can just load a

vulnerable firmware version. Using a modified firmware, they are able to extract the private key, used by the SP to authenticate itself as an AMD device. Among other things, this allows them to fake the presence of an authentic AMD SP.

3.3 Intel's Memory Encryption Solutions

Intel has two different memory encryption solutions: SGX and TME/MKTME. SGX is intended to protect small parts of applications and is available in current Intel CPUs. TME provides encryption of the entire RAM with a single key, while MKTME builds on that, adding the possibility to use multiple keys. However, TME and MKTME are not available yet.

3.3.1 SGX

Intel SGX [AGJS13, HLP+13, MAB+13, CD16] was introduced in 2013. Its goal is to create a protected enclave, that allows private computations without trusting the OS, the hypervisor or any hardware that is attached to the computer's system bus. SGX ensures the confidentiality as well as the integrity of an enclave. However, it is designed to protect only parts of an application. In addition, existing software must be modified to make use of SGX.
All data related to an SGX enclave, is stored inside the Enclave Page Cache (EPC), which is a subset of the Processor Reserved Memory Range (PRM), a special, continuous memory area reserved for SGX. The EPC is divided in 4KB pages and each page can only be used by a single enclave. I.e., EPC pages cannot be used to share data between enclaves. The size of the EPC is very limited, but it is possible to use paging to temporarily remove pages. However, this requires additional OS support, as special SGX instructions must be used for evicting and restoring pages.
There are two mechanisms to protect data in the EPC. The first is, that the EPC's memory pages cannot be accessed by software outside of the enclave, removing the need to trust the OS's page tables to prevent such memory accesses. The access checks are performed in hardware. In addition to this access right based protection, SGX uses cryptography to ensure the confidentiality, integrity and freshness of all pages in the EPC. The data only gets decrypted when it enters the CPU.
An existing enclave can essentially be in three states: "uninitialized", "initialized and in use" and "initialized and not in use". Enclaves are managed by newly introduced instructions. If and only if an enclave is in the "uninitialized" state, untrusted software can load data into the enclave, via the `EADD` or `EEXTEND` instructions. The loaded data is hashed. If the untrusted software is done loading, it executes the `EINIT` instruction. At this point

the hash gets finalized and signed, to allow attestation of the initial enclave content. This way, the creator of the enclave can check whether the untrusted software, that performed the loading process, has tampered with the data. If the attestation is successful, the enclave enters the "initialized and not in use" state. During the lifetime of an enclave the state cannot be reverted to "uninitialized", implicating that untrusted software cannot use the loading mechanism anymore.
Entering and exiting enclaves works via special instructions, similar to entering and exiting VMs. The state of an enclave is stored in a data structure called State Save Area (SSA) which resides in the EPC. The state is loaded into the CPU upon entering the enclave and stored before exiting it. This is comparable to the VMCB mechanism in the context of virtualization. In order to prevent leaking secrets from the enclave, the CPU's state-related registers are overwritten with dummy values, before the enclaves exits.
SGX also includes a remote attestation feature, to prove to a remote party, that it is talking to an SGX secured enclave.
Intel SGX has two drawbacks in comparison to AMD's SME/SEV technology. The first is the limited size of the EPC, as it is designed to only protect small parts of an application, instead of a whole VM, like SEV. However, there has been research to protect whole applications or even Docker containers with SGX [ATG+16, BWG+16, TPV17, KPM+16, KCV18, KPM+16]. The second drawback is, that software must be modified in order to make use of SGX. AMD's SEV on the other hand, only requires modifications to the OS, but not to userland software running in the SEV secured VM. For SME, there even is the TSME mode which does not require any software modifications at all. However, the major advantage of SGX is that it does not only provide confidentiality like SME/SEV, but also integrity and freshness. A more detailed comparison between Intel SGX and AMD's memory encryption technologies can be found in [MZLS18].
There have been multiple attacks on Intel SGX that use side channels, like the CPU cache, to extract secret data from enclaves [BMD+17, GESM17, LSG+17, WCP+17, SCNS16]. Other attacks [BCD+18, LJJ+17] exploited memory corruption vulnerabilities of code running inside enclaves, to perform Return Oriented Programming (ROP) attacks.

3.3.2 Intel TME/MKTME

Intel TME and MKTME [Int17] are planned features, that will be supported in future Intel processors. TME provides encryption of the entire RAM, like AMD's SME. It requires a small change in the BIOS to be globally activated and thus does not need software modifications. It will use AES-XTS with a 128-bit key, that gets randomly generated upon boot and is not accessible by software.
Intel MKTME builds on TME but allows the usage of multiple keys on a per-page basis.

The decision, which key should get used to access a page is stored in the page table entries and is thus manageable by software. Intel claims, that this enables hypervisors to isolate whole VMs or containers from each other, by encrypting their memory with different keys. However, it is unclear, whether MKTME can also be used to protect against a malicious hypervisor. In addition to keys generated by the hardware, MKTME will also support software provided keys.

4 Reverse Engineering the Encryption Mode

In order to predict how the plaintext that corresponds to a ciphertext block changes, when the ciphertext block gets copied to a new memory location, we need to reverse engineer the AES encryption mode, particularly the address based tweak function. Only with this knowledge, we are able to inject meaningful data into the VM via ciphertext moving.

As shown in [DYM+17], AMD uses a tweaked AES encryption to avoid that a ciphertext block appears multiple times due to an identical plaintext. If AMD would not have added any randomization, it would have been trivial to move ciphertext blocks, and easy to fingerprint applications by detecting certain repeating patterns in memory, e.g. alignment bytes between functions, or zeroed pages.

Since an encrypted block does not have any kind of tag or temporal information, AMD uses a function of its physical address to compute the associated tweak value. In the following, we summarize our findings on that function and verify and extend the results from [DYM+17].

We performed our initial analysis on an AMD Ryzen 1950X which only supports SME. Later we reproduced these results (with the exception of some minor changes on an AMD Epyc 7251 with full SEV support. Finally, we analyze the newer AMD Epyc 3151 and show, for the first time, that it uses a new enhanced encryption mode. However, the mode is still vulnerable to previous attacks.

4.1 Analysis of AMD Ryzen and AMD Epyc 7xx1 CPUs

According to [DYM+17], AMD uses a fixed array of 16-byte *tweak constants* t_i for $i \geq 4$. Given a physical address p, where $\mathrm{bit}(p, i)$ represents its i-th least significant bit for $i \geq 0$, the *tweak value* $T(p)$ is defined as

$$T(p) \coloneqq \bigoplus_{i=4}^{n-1} \mathrm{bit}(p, i) \cdot t_i,$$

so for each physical address bit the respective tweak constant is XORed, if that bit is 1. As the lowest 4 address bits index inside a 16-byte AES block, they are not used by the tweak function.

A 16-byte plaintext block $m \in \{0,1\}^{128}$ with physical address p is then encrypted as

$$\mathrm{Enc}_K^{\mathrm{XE}}(m,p) \coloneqq \mathrm{AES}_K\left(m \oplus T(p)\right).$$

Similarly, decryption of a ciphertext c uses the inverse transformation:

$$\mathrm{Dec}_K^{\mathrm{XE}}(c,p) \coloneqq \mathrm{AES}_K^{-1}(c) \oplus T(p).$$

This construction is a variant of the XE mode of operation [Rog04].

In order to verify, that the tweak function is indeed $T(p) = \bigoplus_i \mathrm{bit}(p,i) \cdot t_i$, as well as to provide a universal method for computing the tweak values, we modeled the effect of the tweak function with a system of linear equations. For simplicity we assume 64-bit addresses.

$$\begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{pmatrix} \cdot \begin{pmatrix} t_{63} \\ t_{62} \\ \vdots \\ t_0 \end{pmatrix} = \begin{pmatrix} T(p_1) \\ T(p_2) \\ \vdots \\ T(p_n) \end{pmatrix}.$$

Unfortunately, it is not possible to directly observe the tweak values for a given address due to the AES encryption. However, by moving a ciphertext block to another address before decrypting it, we are able to observe the XORed tweak values for two different addresses. If q is the address in which m gets encrypted, then moving its ciphertext to another address p and decrypting it there yields

$$\begin{aligned} &\mathrm{Dec}_K^{\mathrm{XE}}(\mathrm{Enc}_K^{\mathrm{XE}}(m,q),p) \\ &= \mathrm{Dec}_K^{\mathrm{XE}}(AES_K(m \oplus T(q)),p) \\ &= \mathrm{AES}_K^{-1}(AES_K(m \oplus T(q))) \oplus T(p) \\ &= m \oplus T(q) \oplus T(p). \end{aligned}$$

Since we are able to observe $m \oplus T(q) \oplus T(p)$, we can repeat this experiment with a constant ciphertext at address q and varying destination addresses p_i, until the destination addresses form a basis of the address space (or some subspace of it). This allows us to recover the tweak values by solving the following system of linear equations. To simplify

Table 4.1: The first three tweak constants on an Epyc 7251 processor. We denote the first one as t4 , since there are no dedicated constants for the least significant bits 3 to 0. This also implies that each tweak constant has a length of 16 bytes.

t_4	82	25	38	38	82	25	38	38	82	25	38	38	82	25	38	38
t_5	ec	09	07	9c	ec	09	07	9c	ec	09	07	9c	ec	09	07	9c
t_6	40	00	00	18	40	00	00	18	40	00	00	18	40	00	00	18

the equations, we omit m as it is a known, constant value.

$$\begin{pmatrix} q \oplus p_1 \\ q \oplus p_2 \\ \vdots \\ q \oplus p_n \end{pmatrix} \cdot \begin{pmatrix} t_{63} \\ t_{62} \\ \vdots \\ t_0 \end{pmatrix} = \begin{pmatrix} T(q) \oplus T(p_1) \\ T(q) \oplus T(p_1) \\ \vdots \\ T(q) \oplus T(p_n) \end{pmatrix}.$$

Note that q can be chosen arbitrarily without altering the solution space of the system of equations since it is simply a linear combination of

$$\begin{pmatrix} q \\ q \\ \vdots \\ q \end{pmatrix} \cdot \begin{pmatrix} t_{63} \\ t_{62} \\ \vdots \\ t_0 \end{pmatrix} = \begin{pmatrix} T(q) \\ T(q) \\ \vdots \\ T(q) \end{pmatrix} \text{ and } \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} \cdot \begin{pmatrix} t_{63} \\ t_{62} \\ \vdots \\ t_0 \end{pmatrix} = \begin{pmatrix} T(p_1) \\ T(p_2) \\ \vdots \\ T(p_n) \end{pmatrix}.$$

The data required for solving these systems of linear equations can be generated by a Linux kernel module. This allows recovering the tweak values at run time, with very low overhead.

The first few constants are shown in Table 4.1. Each constant consists of a repeating pattern of 4 bytes, thus reducing its entropy to at most 32 bits.

The tweak constants on our Epyc 7251 mostly equal those from [DYM+17], who used a Ryzen 7 1700X. This suggests that AMD hardcoded these values, or at least uses a fixed seed to generate them on startup. However, even fully randomizing these values on boot would not add any security, since they are shared across VMs and the hypervisor thus could easily compute them in advance, as shown above.

We also performed these experiments on an AMD Ryzen 1950X, which only has SME support. On our first measurements, we found that $t_8 = t_9 = 0$, which leads to four offsets inside each page, that share the same tweak value and thus encrypt to the same ciphertext if the plaintexts are equal. Sometime later, after applying several OS and BIOS updates, the tweak values t_8 and t_9 changed, removing those patterns. This leads us to the conclusion that the tweak values are influenced by firmware. We used a MSI x399 SLI

Plus Motherboard. The BIOS Version after the update is E7B09AMS.A70.
After our initial experiments on the AMD Ryzen 1950X we built a fingerprinting mechanism for the Linux kernel based on these patterns, as, at that point in time, we were unaware that $t_8 = t_9 = 0$ does not hold on the SEV enabled AMD Epyc product line. In an offline phase, we built a fingerprint that states which pages of the kernel's `.text` section contain equal plaintext at the four offsets that share the same tweak value due to $t_8 = t_9 = 0$. In the online phase, we dumped the system's memory and tried to find a page sequence that matches the fingerprint (note, that the Linux kernel is loaded to continuous GPA's in a VM).
We evaluated this approach using the Linux kernel versions 4.9.0-3 to 4.9.0-9 from Debian "Stretch" and were able to identify them without any errors. This result also highlights the importance of linear independent tweak values (especially for the page offset bits), as linear dependencies between them would also lead to memory locations that share the same tweak value. However, as we will see in Chapter 5, linear independent tweak values in combination with missing integrity protection, strengthen our ability to inject code into the VM.
In summary, these results show that XE schemes in combination with missing integrity protection leak information about the tweak function. This is problematic especially in the context of RAM encryption, where the tweak function is required to have low computational complexity. The techniques presented in this section show a fundamental approach to reverse engineer such tweak functions.

4.2 Updated Encryption Mode for newer Epyc 3xx1 CPUs

The Epyc Embedded 3151 processor was released about 8 months after the Epyc 7251, and features full SEV support. We conducted the same experiments as on the Epyc 7251, and found that the system of linear equations does not have any solutions on the new processor, i.e. AMD must have changed the encryption mode.
To reverse engineer the new encryption mode, we assumed that AMD did not greatly deviate from their previous implementations, and thus conducted a few experiments with slightly modified functions which used the same tweak values as before. This approach proved successful and yielded the new encryption function

$$\mathrm{Enc}_K^{\mathrm{XEX}}(m, p) \coloneqq \mathrm{AES}_K\left(m \oplus T(p)\right) \oplus T(p),$$

and the matching decryption function

$$\mathrm{Dec}_K^{\mathrm{XEX}}(c, p) \coloneqq \mathrm{AES}_K^{-1}\left(c \oplus T(p)\right) \oplus T(p).$$

As these equations show, AMD chose to use the XEX [Rog04] mode of operation, where a second tweak value is XORed to the AES encrypted ciphertext; in this case, both tweak values are identical.

The altered encryption function significantly complicates the calculation of the tweak constants, since simply decrypting a ciphertext at a different position does not yield usable results anymore:

$$\begin{aligned}&\mathrm{Dec}_K^{\mathrm{XEX}}\left(\mathrm{Enc}_K^{\mathrm{XEX}}\left(m,p\right),q\right)\\&\quad=\mathrm{AES}_K^{-1}\left(\mathrm{AES}_K\left(m\oplus T(p)\right)\oplus T(p)\oplus T(q)\right)\oplus T(q).\end{aligned}$$

Instead, the attacker needs to guess $T(p) \oplus T(q)$ and add this number to the ciphertext before decryption. He can then check his guess by computing

$$\begin{aligned}&\mathrm{Dec}_K^{\mathrm{XEX}}\left(\mathrm{Enc}_K^{\mathrm{XEX}}\left(m,p\right)\oplus T(p)\oplus T(q),q\right)\\&\quad\stackrel{?}{=}\mathrm{AES}_K^{-1}\left(\mathrm{AES}_K\left(m\oplus T(p)\right)\right)\oplus T(q)\\&\quad=m\oplus T(p)\oplus T(q).\end{aligned}\tag{4.1}$$

If all 128 bits of the tweak constants were chosen randomly, this operation would become infeasible; however, AMD still uses the repeated 4-byte pattern, so each tweak constant has only 32 bits of entropy.

Guessing these tweak constants is still computationally expensive, since one has to flush the respective TLB entry and the CPU caches when changing the encryption status of a page. Only performing one guess per flush operation, would require more than a day, to brute force a single tweak value. The number of guesses per flush operation can be increased with the following strategy.

Let us assume we want to brute force the tweak for bit i and already know the tweaks t_4 to (including) t_{i-1}. First, we encrypt our initial plaintext at an address q, which has bit i set, but does not have set any bits with a larger index. This allows us to use all 16-byte blocks, whose address p does have set any bit with an index larger than $i - 1$, to test a different guess for t_i. Figure 4.1 shows a visualization of this memory layout. In practice, some of these blocks might already be in use. To do so, we first copy the original ciphertext from address q to p. As we choose the addresses q and p in such a way, that $q \oplus p$ has bit i set, but does not have set bits with a larger index, we can calculate the tweak difference of q and p except for the influence of bit i, for which we guess a tweak value. Afterwards, we can test our guess by decrypting the moved ciphertext blocks, as shown in Equation 4.1. If we do not know any tweak values, we need to use a slightly less efficient strategy, as the addresses p used for guessing tweak values may only differ in bit i. However, we can

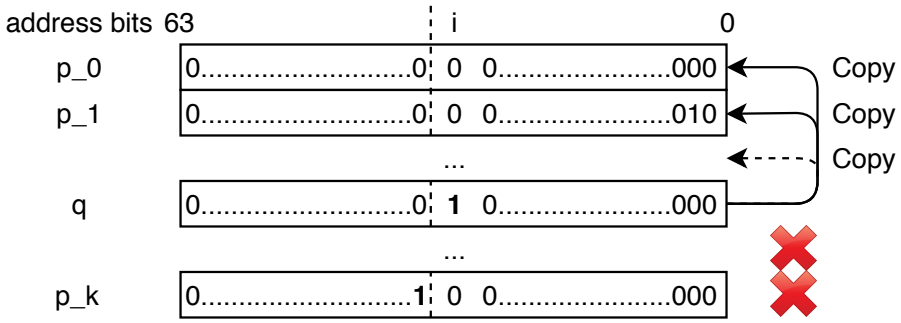


Figure 4.1: Memory layout for brute-forcing tweak t_i, when t_4 to t_{i-1} are already known. The original ciphertext gets encrypted at block q and is copied to blocks with a smaller address. This way, we can calculate the tweak difference of q and p_i except for the influence of bit i, for which we want to guess the tweak value.

work around this, by encrypting our initial plaintext at multiple addresses.
Using these strategies, we only need around 30 minutes for each tweak constant. Given that even the newer CPUs still use the same tweak constants for every VM, the hypervisor can pre-compute the table once in advance, so the slightly higher computation time becomes negligible in terms of security. Even changing the tweak values each boot would not mitigate this as the decision to reboot is controlled by the attacker.
In summary, we showed that on the recent Epyc 3xx1 product line AMD implemented the well-known XEX encryption mode. However, the tweak values have very low entropy and depend linearly on the physical memory addresses, enabling a malicious hypervisor to compute the entire table of tweak constants nevertheless. In the next chapter, we will exploit this fact and show how known plaintext can be used to place arbitrary code and data in the encrypted VM.

5 Fault Injection Attacks

As we have seen in the previous chapter, we can compute the tweak values for any physical address. In this chapter, we show how a malicious hypervisor can use the knowledge of the tweak values together with known plaintext and the missing integrity protection, to place 16-byte blocks containing some consecutive, controlled bytes. This narrow attack vector already suffices to insert early returns in functions and skip parts of code, as shown in Subsection 5.4.1. In Section 5.5 these byte sequences are exploited to build a full 16-byte encryption oracle, which allows us to execute arbitrary code on the highest privilege level within the VM. Contrary to previous work [DYM+17, LZL19], which has used network I/O to create an encryption oracle, we do not need any control over the plaintext that gets loaded into the VM in order to inject arbitrary data/code: Instead, we simply use the plaintext that is already inside the VM anyway.

5.1 Attacker Model

We assume that the attacker controls the hypervisor, which implies control over the NPTs and the ability to modify the VM's RAM. The attacker knows at least parts of the binary of the guest kernel, which might be due to the unencrypted `/boot` partition or by using fingerprinting (see Section 7.4). We assume that the VM is secured by SEV-ES, implicating that the initial VM image cannot be tampered with and the VMCB is protected. We do not require, that the VM communicates over the network or uses disk I/O.

5.2 Tracking Guest Execution

To be able to make the VM execute hypervisor-supplied code while being in a known state, we need to follow and eventually suspend its execution. We achieve this by using the page fault side channel, as explained in Section 2.2.
Our attacks require computing tweak values, which in turn depend on HPAs, so we have to infer the latter for both the source and destination GPAs. The NPTs provide this translation. Since we aim at injecting and executing code in the VM, we need to find GPAs that are mapped as executable inside the guest. We cannot directly inspect the page tables inside the VM, but we can acquire this information by monitoring for page faults due to missing execute permissions via the page fault side channel.

The guest kernel is a suitable target for code injection attacks because it is executed with the highest privileges. However, similar to Address Space Layout Randomization (ASLR) for the virtual addresses of userspace applications, the kernel's GPA and GVA are also randomized. This is called Kernel Address Space Layout Randomization (KASLR) and intended to harden the kernel against memory corruption based attacks. In contrast to user space applications, the Linux kernel is loaded to consecutive GPAs and GVAs. This makes it significantly easier to locate it based on page fault information as the GPA of a single function/symbol of the kernel is enough to break the randomization. We present a method for finding the guest kernel in Subsection 5.4.1 and discuss alternatives in Section 7.4.
As mentioned in the attacker model, the kernel code can be assumed to be entirely known to the attacker and thus also serves as a reliable source for ciphertext blocks with known plaintext, which can be copied to other places in the kernel to trigger malicious behavior.

5.3 Placing Partially Controlled Plaintext

Knowing the destination address in VM memory, we can now start to construct our attack primitive. Since SEV lacks any integrity protection, the hypervisor can modify the contents of the entire guest's memory. Randomly guessing ciphertexts is rather unlikely to yield meaningful plaintext and will, especially in the case of code, most probably crash the VM. However, since we can compute the tweak values for any given address, we can re-use existing ciphertext blocks after applying slight adjustments.
We assume that we want to place a 16-byte block m at address p. We then need to find an address q holding a known 16-byte plaintext block m', which satisfies the following property:

$$\begin{aligned} & m \oplus T(p) = m' \oplus T(q) \\ \Leftrightarrow \quad & m' = m \oplus T(p) \oplus T(q). \end{aligned}$$

Copying the corresponding ciphertext block from q to p and decrypting it, yields the desired plaintext block m:

$$\begin{aligned} & \mathrm{Dec}_K^{\mathrm{XE}}\left(\mathrm{Enc}_K^{\mathrm{XE}}\left(m', q\right), p\right) \\ & \quad = \mathrm{AES}_K^{-1}\left(\mathrm{AES}_K\left(m' \oplus T(q)\right)\right) \oplus T(p) \\ & \quad = \left(m' \oplus T(q)\right) \oplus T(p) \\ & \quad = \left[m \oplus T(p) \oplus T(q) \oplus T(q)\right] \oplus T(p) \\ & \quad = m. \end{aligned}$$

To target the XEX encryption mode of the AMD Epyc 3xx1 product line, the copied ciphertext block needs to be slightly adjusted, by adding an additional $T(p) \oplus T(q)$ to cancel out the effect of the additional XOR:

$$\begin{aligned}
&\mathrm{Dec}_K^{\mathrm{XEX}}\left(\mathrm{Enc}_K^{\mathrm{XEX}}(m', q) \oplus T(p) \oplus T(q), p\right) \\
&\quad = \mathrm{Dec}_K^{\mathrm{XEX}}\left(\mathrm{AES}_K\left(m' \oplus T(q)\right) \oplus T(q) \oplus T(p) \oplus T(q), p\right) \\
&\quad = \mathrm{Dec}_K^{\mathrm{XEX}}\left(\mathrm{AES}_K\left(m \oplus T(p)\right) \oplus T(p), p\right) \\
&\quad = m.
\end{aligned}$$

The complexity of the bit sequences a malicious hypervisor is able to create with this method is limited by several factors. The first is the diversity of the known plaintext blocks, i.e. whether they have enough entropy. The next limitation is the 32-bit periodicity of the tweak values (which we can control by choosing the HPA a GPA gets mapped to), so we can expect to able to control at most 4 bytes of any 16-byte block in a reliable way. Finally, for the 28 tweak values we reverse engineered on our processor, we found that all of them are linear independent, so for each guest page, the hypervisor can choose from up to 2^{28} different base addresses which yield different ciphertext blocks. This suggests a rough upper bound of 3 bytes per block, which an attacker is likely able to fully control, if given enough plaintext. Given more memory, there might even be more independent tweaks (but no more than 32 as the 128-bit tweaks are 32-bit periodic).
In our experiments, we found that we can very reliably find a fitting pair m'/q for any sequence of two bytes, given about 8 MB of known plaintext. We obtained them by copying the `.text` (code) section of the Linux kernel bootstrapper as it gets loaded into memory, which can be easily located due to the lack of randomization of its load GPA. In addition, as discussed in the previous section, we can also use the `.text` section of the kernel binary itself as a known-plaintext source.

5.4 Code Injection

We now show how the two controlled bytes per block can be used to modify existing VM code, allowing us to redirect control flow and to insert arbitrary 2-byte instructions.
Instructions on x86-64 have variable length and might share prefixes, so we have to consider whether we change or break an existing instruction when injecting our 16-byte block. Also, we have to ensure that the uncontrolled bytes of our block do not get executed since we cannot control their effect which most likely results in a crash. The easiest way to achieve this is by finding a 16-byte aligned instruction and overwriting it with a short

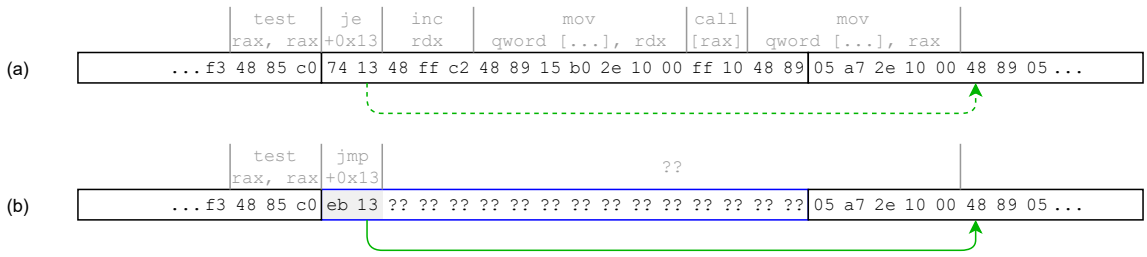


Figure 5.1: Example for changing execution flow by replacing one 16-byte block of code. While the base program (a) branches conditionally depending on the value of the `rax` register, the patched version (b) has this branch replaced by an unconditional one. The remainder of the inserted block consists of uncontrolled bytes, which are not expected to form a sequence of meaningful instructions.

branch instruction like `ret` or `jmp`, like show in Figure 5.1. This simple modification already suffices to completely disable KASLR, as we will see in Subsection 5.4.1.

Finding a 16-byte aligned instruction for an injection point is rather easy for 64-bit code: For performance reasons, most compilers align functions and frequently used chunks of functions to an architecture-specific value, which usually happens to be 8 bytes on x86-64, so we can expect around every second function to be aligned to a 16-byte boundary.

To avoid executing the uncontrolled bytes of a block, we always have to insert a jump instruction – which takes both usable bytes of a block, so this method only allows us to skip small parts of the underlying code. For inserting other instructions, we propose the layout shown in Figure 5.2. First, we inject a `jmp` at a 16-byte aligned instruction. With this, we jump to offset 14 of the following block, where we can place an arbitrary two-byte instruction. We will call this the *payload*. Then we can use the first two bytes of the following block to again jump to the next payload location. This way we maximize the number of consecutive bytes that we can control.

In Subsection 5.4.1, we successfully use this method to disable KASLR and illustrate a fast `cpuid`-based 16-byte encryption oracle. Finally, in Section 5.5 we build another 16-byte encryption oracle which solely relies on ciphertext block moving and a synchronization mechanism. We show how the latter can be implemented based on intercepted instruction or the page fault side channel. The 16-byte encryption oracle does not depend on the capability of the hypervisor to modify the `cpuid` registers, so it is hard to mitigate without introducing proper integrity protection. Both encryption oracles allow us to execute arbitrary code within the VM.

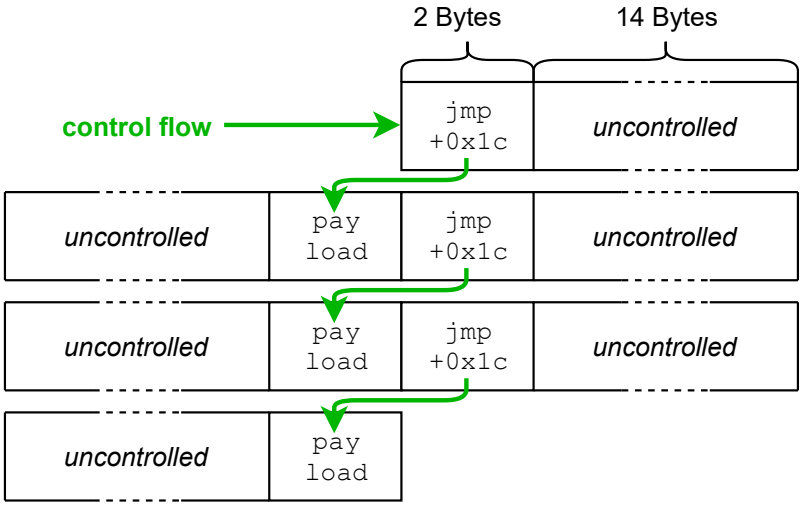


Figure 5.2: A sequence of consecutive 16-byte blocks are chained together to get small contiguous chunks of code, which are connected by unconditional 2-byte `jmp` instructions to avoid executing the uncontrolled bytes in between. Thus two bytes of every second block can be used to execute arbitrary 1-byte or 2-byte instructions (payload). The last payload may either redirect to original code (e.g. by returning) or enter a loop.

5.4.1 Attack Case Studies

This section shows example applications of the just created injection primitive. As we will see, even very subtle modifications to an application's control flow can have severe effects.

Disabling KASLR

To be able to track the VM's execution, the hypervisor needs to know the base GPA of the kernel, which is randomized by KASLR, as discussed in Section 5.2 . In order to perform a first demonstration of our attack primitives, we disable KASLR using the one-block code injection method from Section 5.4, effectively placing the kernel at a well-known, constant GPA.

To perform KASLR, the function `choose_random_location` is called, while the kernel bootstrapper loads the actual kernel code. The function checks whether the user provided the *nokaslr* kernel command-line option; if this is the case, it returns immediately. Else the function computes random physical and virtual offsets and adds them to the standard base addresses for the kernel. Therefore, it is sufficient to place a `ret` at an early location in the function, to disable KASLR.

For a successful modification we need to perform the attack after the code has been loaded into memory, but before it gets executed. To get the right timing, we utilize the page fault

side channel as explained in Section 2.2. We remove the write permissions to the physical page *after* the first part of the targeted function is copied. This causes a page fault to be triggered as soon as the boot loader is done copying the first part of the function and tries to copy the next one. When handling the page fault, the hypervisor places the block containing the `ret` instruction and then resumes execution.

CPUID Encryption Oracle

Our code injection primitive can be combined with the hypervisor-interceptable `cpuid` instruction to gain control over certain general purpose registers and build a high-performance 16-byte encryption oracle.

As explained in Subsection 2.4.3, the content of the VMCB gets encrypted and integrity protected upon a `#VMEXIT` in case SEV-ES is enabled. This prevents a malicious hypervisor from manipulating its content; however, in order to emulate instructions like `cpuid`, the value of certain registers is still shared via the GHCB. While the guest owner may disable instruction emulation, they are an important virtualization feature that allows for fine-grained control over exposed hardware features as well as keeping the VM's environment consistent in case of live migration.

OS kernels frequently call the `cpuid` instruction during startup to retrieve information about system capabilities and topology. Since the results of these calls often get directly stored in memory for caching purposes (e.g. the vendor string), this poses an easy target for injection attacks. First, we determine the HPA of the `cpuid` call and the associated memory store; then we inject a block containing an unconditional jump to the `cpuid` instruction after the memory store in order to create a loop. On each `cpuid` call, the hypervisor sets the return registers, resumes execution and waits for the next `cpuid` call. When this call occurs, the data from the last call has been stored, so the hypervisor can copy the encrypted data to the desired location.

We implemented this exploit in the `get_model_name` function (`arch/x86/kernel/cpu/common.c`) of the Linux kernel, since it writes the `cpuid` result to a contiguous block of 16-bytes, which can be directly used as an encryption oracle. One could also use our basic injection attack to create such a `cpuid` loop. The results can be stored on the program's stack memory, whose GPA can be determined by the stack detect gadget which will be presented in the next section. Since we only need one context switch between VM and hypervisor per 16-byte block, this channel is very efficient: We encrypted 1'000'000 blocks (16 MB) within around 37.5 seconds, suggesting a bandwidth of around 3.41 MBit/s or 426.67 KB/s.

Breaking OpenSSH Privacy

SSH is a very popular remote access solution under Linux. While there are various implementations, OpenSSH is the most popular one [GHC14]. In the SSH handshake, the client and server first negotiate a shared secret using a Diffie-Hellman (DH) handshake. Next, the server proves its authenticity via the Digital Signature Algorithm (DSA) or Elliptic Curve Digital Signature Algorithm (ECDSA). We show that we can compromise the shared secret, as well as the private key used for the DSA signature, by altering the control flow in the random number generators used for generating the randomized values in the respective algorithms. For our experiments, we used Ubuntu with OpenSSH_7.9p1 and OpenSSL 1.1.1c.

In our OpenSSH version, the x25519 variant of the Elliptic Curve Diffie-Hellman (ECDH) handshake is used, which is implemented directly in the OpenSSH server. The random number for the server's private part of the handshake is generated by a ChaCha20-based random number generator. The content of a global variable *rs* is used as a key for the ChaCha20 algorithm. During the startup of the OpenSSH process, *rs* is seeded from the OpenSSL random number generator. OpenSSH uses its own process for every connection. The keystream generated by ChaCha20 is stored in a buffer and used as random data on request. If all data from the buffer has been read, ChaCha20 is called again to reseed *rs* with new random data.

The functions for seeding *rs* on startup are 16-byte aligned and have no return value. Thus we can abort them by injecting a `ret` instruction to prevent seeding. This means that *rs* (as a global C variable) is still initialized to zero when it is used as a key for ChaCha20.

This allows the attacker to calculate the shared secret, as he knows the private value used by the server and can easily obtain the client's public value by monitoring the VM's network traffic.

The attack on the signature algorithm works in a similar fashion. The OpenSSH server uses OpenSSL's implementation of DSA/ECDSA to prove its identity. Both variants use an ephemeral key value for the signature, that must remain secret. Otherwise, the private key can be calculated given the message and the signature. Using our injection primitive we can reduce the randomness to an uninitialized stack array by inserting a `jmp` instruction. With the OpenSSL 1.1.1c version, this value was always constant. We also tried this with the OpenSSL versions shipped in Debian Buster (stable) and Debian Bullseye (next stable). In these versions about 32 bits of the value were randomized.

In OpenSSH, the message that gets signed to prove the server's authenticity, contains the shared secret generated by the DH handshake and is never explicitly sent over the network. Instead, both parties calculate them from the transmitted values. Thus an attacker either has to initiate a handshake himself or combine this attack with the attack on the

shared secret, to be able to infer the message that gets signed. The latter has the advantage that the attacker does not need to generate network packets himself. In [HB17, MHH19], it is shown how to identify pages containing specific data of user space processes with the page fault side channel. This should also be applicable to identifying the GPA of OpenSSH's code pages, in order to perform our attack.

5.5 Executing Arbitrary Code

The previous two examples have shown that even little modifications to the control flow can have a severe effect on the system's overall security. However, our ultimate goal is to execute arbitrary code, without having to rely on the ability to control register contents through an intercepted instruction, or use of I/O. We will advance the 4-byte block chaining method from Section 5.4, to inject a program into the VM, which writes arbitrary data into a 16-byte block of memory. This block encryption oracle enables us to execute arbitrary code with kernel privileges inside the VM. We show that the oracle can easily be used to construct a decryption oracle as well.
The basic idea is to inject a small code gadget into the VM, that performs some computations in order to write 4 bytes of plaintext into a 32-bit register. Next, we push this register onto the stack, to get an encrypted version of our plaintext; this serves as an intermediate 4-byte encryption oracle, so we are able to control $2+4 = 6$ consecutive bytes. We then use this increased payload size to repeat the same process with 64-bit registers, finally giving us control over the full 16 bytes of a block.

5.5.1 Synchronisation with Hypervisor

The proposed attack needs careful synchronization between VM and hypervisor, such that the hypervisor can suspend execution at a precise point in time and modify guest memory. We propose two different mechanisms to achieve this. The first mechanism utilizes the `cpuid` instruction, which can be intercepted by the hypervisor and features a 2-byte opcode: Each time `cpuid` is executed, the hypervisor is called to emulate it. So, by interleaving the injected instructions with `cpuid` calls, we can precisely redirect execution to the hypervisor.
The `cpuid` calls clobber the `eax`, `ebx`, `ecx` and `edx` general purpose registers, so they are not usable for the constructed gadgets. Also, the `eax` register (which determines the requested leaf ID) should be cleared beforehand to avoid calling additional handling logic in the hypervisor – leaf 0 just returns the vendor ID.
It is convenient to use the `cpuid` instruction because it has a simple handler in the KVM hypervisor. As stated in Section 2.1, the interception of instructions like `cpuid` or `rdtsc`

is important, as it allows fine-grained control over exposed hardware features, as well as providing the VM with consistent information after a live migration. However, the guest owner can configure which instructions are interceptable by the hypervisor via the VM's VMCB.
A more complex alternative to the `cpuid`-based execution transfer is the usage of the page fault side channel, which also allows a precise interruption of the VM. If we want to interrupt the VM between two injected instructions, we ensure that they reside on two different pages p_1 and p_2, and remove the execute permission in the hypervisor's NPT. This way, the VM gets interrupted before the first instruction in p_2 gets executed. We then remove the execute permission for p_1, such that the hypervisor gets triggered another time to remove execute permissions for p_2 once again.
For simplicity, we use the `cpuid` instruction if we want to express that the VM should be interrupted at a certain point. However, the same results can be achieved by only using the page fault side channel, which is more cumbersome to use, but also more difficult to mitigate.

5.5.2 Locating the stack memory

In order to use the stack for our encryption oracle, we need to get the HPA of the related stack pages. We solve this problem by combining the `cpuid` method and the page fault side channel.
We use the attack primitive from Section 5.4 to construct an instruction sequence `cpuid; push rdi`. `cpuid` triggers the hypervisor, which then removes write access from all memory pages belonging to the VM, and resumes execution. The following `push rdi` tries to write to the non-writable stack memory page and subsequently raises a page fault in the hypervisor. The page fault exception information yields the corresponding GPA and thus the HPA of the stack.
If the write address of the `push rdi` is near the end of a page, the hypervisor may issue an extra `pop rdi` instruction to ensure that the next stack operation writes to the same page. This also significantly eases restoring original execution after inserting the encryption oracle code.

5.5.3 4-Byte Encryption Oracle

Originally, x86 only supported 16-bit and 32-bit operands. When the CPU vendors implemented support for native 64-bit operations, they did not add new opcodes for every general-purpose instruction (e.g. arithmetic and memory-to-register/register-to-memory); instead, they introduced the *REX* prefix, which, when put before an instruc-

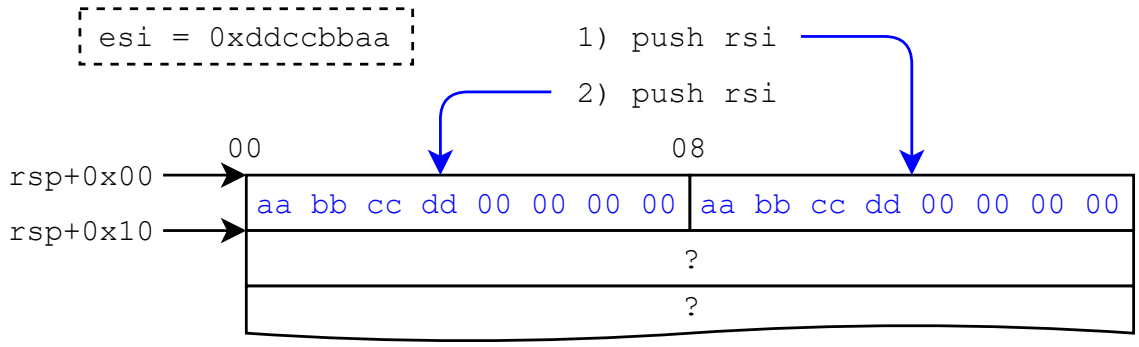


Figure 5.3: Layout of stack after pushing the 32-bit `esi` register. The stack pointer is decreased when pushing a register, so, depending on the stack pointer's original alignment, we might have to push the register another time to set the lower address part of a 16-byte block. Since this is a 64-bit operation, the (zeroed) higher 32-bits of `rsi` are pushed as well. Due to the endianness of x86 the lower significant bytes end up first, the higher bytes last. We thus finally get a 16-byte block where we control the first 4 bytes.

tion's opcode, upgrades its operands to 64-bit mode. Since in 32-bit mode most general-purpose instructions are encoded using at least 2 bytes, this prefix extends them to 3 bytes – but our attack primitive only supports 2 bytes of payload. However, when adding 64-bit support, the 1-byte `push` *`reg`* instructions were redefined to only support 64-bit registers, so we can use the payload to perform stack writes. We thus can use 32-bit instructions to control the lower half of some registers, and then push those onto the stack. Hence we can control the lower 4 bytes of a 16-byte block, so the possible payload is doubled, enabling us to use 64-bit instructions for the next step.

x86 is a little-endian system, so when we push a register to the stack, its bytes are stored in reversed order. This means, if we set the least significant 32 bits of a register and push it to the stack, those bits will be placed at lower addresses (Figure 5.3). If the stack pointer has been 16-byte aligned before our first push, the controlled bytes will then reside in the middle of the 16-byte block, where we cannot chain them with another block. So we have to push the register *a second time* – now the stack pointer is 16-byte aligned, and the payload resides at the block beginning. Depending on the stack page offset and the number of blocks being created, one might have to add some `pop` instructions to free up stack space before proceeding with the next block.

As the last building block, we need a gadget to place an arbitrary 32-bit value into a register. This gadget can be constructed via a simple combination of increments and left shifts: First, the register is cleared by XORing it with itself (this also automatically clears the upper 32-bit of the corresponding 64-bit register). To add a 0 bit, the register is just shifted; to add a 1 bit, the register is incremented and then shifted. This will take at most 31 rounds

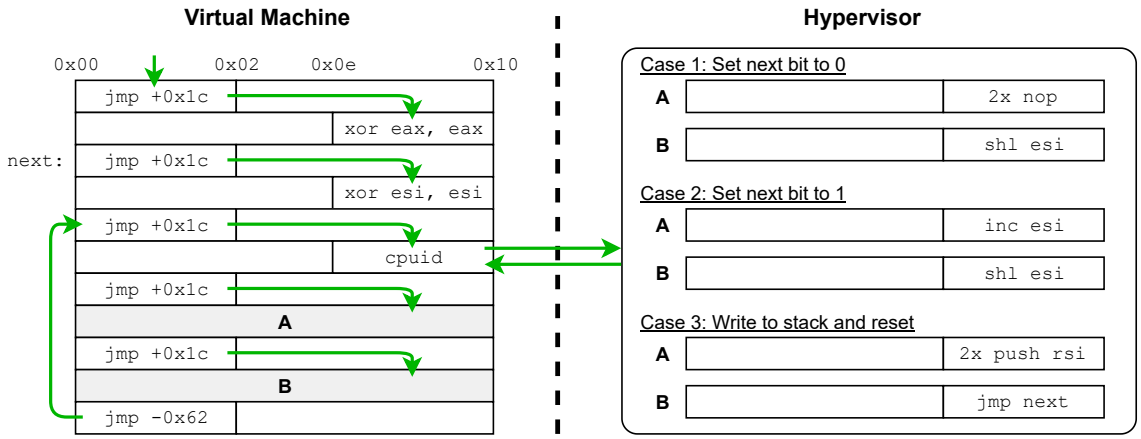


Figure 5.4: Schematic of the 4-byte encryption oracle. Each row represents a 16-byte block (not to scale), control flow jumps are denoted by arrows. On each call of `cpuid`, the hypervisor replaces blocks A and B depending on the desired action: It may either shift 0s and 1s into `esi`, or push the `rsi` register two times to the stack to get an encrypted 16-byte block. This process can be repeated arbitrarily often.

until the most-significant bit has been set. All the involved instructions have 2-byte opcodes. The final block layout forming the 32-bit oracle is shown in Figure 5.4. To move the payload from the location where it gets encrypted to another memory location, we need to consider the XOR difference of the tweaks used at these two memory locations and XOR it with our payload before using the 4-byte oracle.
In summary, we are now able to control 4 byte per 16-byte block. In the next paragraph we show that this is sufficient to inject a program allowing us to control a whole 16 byte block.

5.5.4 16-Byte Encryption Oracle

The 16-byte encryption oracle works very similar to the 4-byte encryption oracle. First, we ensure that the stack is 16-byte aligned; if we used the described process for creating the 4-byte encryption oracle, we already have this information. Then we use the same strategy as in the 4-byte encryption oracle to load the two 64-bit chunks of our plaintext into 64-bit registers and push them onto the stack. Since we made sure that the stack was 16-byte aligned before the first push operation, we now have an entire 16-byte aligned 16-byte block in memory, which only needs to be copied to the desired location.
The formerly introduced 4-byte oracle allows us to use 6-byte instruction gadgets, so after subtracting the necessary `jmp` instructions we can use 4 bytes of payload. This is sufficient for most 64-bit register-to-register arithmetic. Though there might be more efficient methods for assigning hypervisor-defined values to a 64-bit register, we reuse the incre-

Figure 5.5: Example for injection of a 3-byte opcode payload followed by an unconditional jump, using a block created with the cipher block moving primitive, and one block from the 4-byte encryption oracle. It is desirable to fully use the 4 bytes from the encryption oracle, since finding a fitting block for the cipher block moving primitive requires more complexity, when the number of payload bytes increases.

ment/shift method for sake of simplicity.
The implementation is very similar to the 4-byte oracle: All instructions involving the target register (`rsi`) are extended to 64-bit using the REX opcode prefix. Additionally, instead of pushing `rsi` twice, it is only pushed once and another iteration is started to push another value. This way we can fully control all 128 bits of the plaintext block. Figure 5.5 shows an excerpt of a gadget using a 3-byte opcode payload.
In summary, we are able to encrypt arbitrary 16-byte values, by injecting a program into the VM that performs some computations in order to write data into encrypted memory owned by the VM.
Finally, we show how to implement the gadgets from the previous paragraphs without the use of intercepted instructions like `cpuid`. We demonstrate this with the 4 byte encryption oracle. We first split the instructions shown in Figure 5.4 between two pages $p1, p2$ as follows: everything above the `cpuid` instructions is stored at the end of $p1$ while everything below this instructions gets stored at the start of $p2$. The `cpuid` instruction itself is removed. Before entering the gadget, we remove the exec rights from $p2$. Thus, the VM gets interrupted before the first instruction of $p2$ is executed. When receiving the interrupt, in addition to deciding which blocks should get inserted at A and B, we also remove the exec rights from $p1$. This way, the VM gets interrupted again, when we jump back to $p1$ via the `jmp` instruction after the placeholder. B, allowing us to again remove the exec rights from $p2$. This concludes the first round. Note that we are back at the initial state.

5.5.5 Code Execution allows stealthy Decryption

Throughout this section, we have shown how to execute arbitrary code via a self-bootstrapping, non I/O dependent encryption oracle. This of course raises the question, if it is possible to create a decryption oracle with a similarly low set of requirements. We now show how a decryption oracle can be constructed by extending an idea of Hetzelt

and Buhren [HB17].
As explained in Subsection 2.4.1, the encryption status of a page can be controlled via the C-bit, in each page table entry. This allows the VM to share pages with the hypervisor. Hetzelt and Buhren show that using an encryption as well as an decryption oracle, the hypervisor can insert a shared page into the page table of a process running inside the VM. The hypervisor can then copy the content of an encrypted page into the shared page. In their approach, they use a decryption oracle in order to find a free entry in the page table of a victim process running in the VM. We do not need a decryption oracle for this approach: Allocating a shared page, as well as copying some data to it, can be done via an injected program instead.
Thus, we conclude that the existence of an encryption oracle immediately implies a decryption oracle. Furthermore, this method is very stealthy compared to using loggable network communication to extract data, like in [MHHW18, LZL19]. In addition, this allows for very high throughput, as the copy rate of the injected program is only limited by the VM's ability to write to RAM.

6 Comparison to Related Work

First, we present a performance analysis of our 16-byte encryption oracle, before comparing it to encryption oracles constructed in related work.

6.1 Throughput of our oracle

We performed the experiments related to the encryption oracle on an AMD Epyc 3151, running Ubuntu 19.04 with Linux kernel version 5.0.18 as a host OS and 16 GB of RAM. The guest was running Ubuntu 19.04 with kernel 5.0.0-27-generic and was configured with 1 GB of RAM. The used QEMU version was 2.12.0. We made use of the SEVered framework [MH19] to inject page faults into the VM.
To evaluate the performance of our encryption oracle, we set up a program that waits for a trigger before calling the function in which we injected our gadgets. First, we bootstrap the 16-byte encryption oracle via the stack detect gadget and the 4-byte encryption oracle. Then we use it 1000 times to encrypt 16-bytes of payload data. On our unoptimized prototype, the setup part takes 0.62 seconds and the payload encryption needed 75.86 seconds. This translates to a throughput of 211 Bytes per second for the 16-byte oracle.
Our prototype implementation focuses on ease of implementation and debugability, thus the performance can be improved by writing more than one bit to the `rsi` register before interrupting the computation with a call to `cpuid`. A sequence of zeroes could be written by inserting an x-bit left shift (4-byte opcode), instead of performing x rounds with a single bit left shift. Furthermore, we could simply increase the number of instructions we execute each round, to decrease the number of interrupts/context switches and write operations which require expensive flushes.

6.2 Comparison

In this section, we compare our results to the encryption/decryption oracles constructed by Du et al. [DYM+17] and Li et al. [LZL19]. If not stated otherwise all attacks assume a malicious hypervisor. An overview can be found in Table 6.1.
Du et al. [DYM+17] were, to the best of our knowledge, the first to discover the original encryption mode of SME as well as the tweak values. Their experiments were performed on an AMD Ryzen CPU without SEV support (AMD Epyc 7xx1 CPUs were not readily

Table 6.1: Comparison of different approaches for encryption oracles. [1]Li et al. [LZL19] only specify the decryption rate, but it should be similar to the encryption rate due to the similar construction.

	Du et al. [DYM+17]	Li et al. [LZL19]	`cpuid`	Cipher Block Moving
Needs service in VM	yes	no	no	no
Relies on I/O	yes	yes	no	no
Needs instruction emulation	no	no	yes	no
Encryption rate (B/s)	unknown	200[1]	426670	211

available at that time). They constructed an encryption oracle for a self-built simulation of SEV. Their attack requires knowledge of the tweak values, an Nginx server running in the VM and is not mitigated by SEV-ES.

They found that Nginx stores parts of the data sent to it in consecutive 16-byte blocks at fixed offsets inside a page. Building on this, they send an HTTP packet whose payload is designed in a way, that the parts going to these offsets contain exactly the tweak values of said offsets. This way, the data encrypts to the same ciphertext, making it easily detectable in a memory dump.

They use this to encrypt code and execute it in the VM. In contrast to our encryption oracle, they rely on self-generated network traffic getting processed by an Nginx web server inside the VM as well as the discussed memory management behavior of Nginx. It is unclear whether different services, or even different versions of Nginx, show similar exploitable behavior. They do not give performance measures.

Li et al. [LZL19] showed how to create an encryption/decryption oracle by leveraging unprotected DMA operations, knowledge of the tweak function and control over the NPTs. For the demonstrated attack, they also require network traffic, whose frequency linearly scales with the throughput of their oracles. Their attack works with SEV-ES.

According to them, DMA is the most common method used by VMs to perform I/O operations. They exploit, that current IOMMU hardware (which is responsible for performing DMA) only supports one memory encryption key, while SEV uses one key for the hypervisor as well an additional key per VM. Thus, all DMA operations must be performed on memory pages p_s that are shared between the hypervisor and the VM i.e. encrypted with the hypervisor's encryption key. This means if the guest wants to write data via DMA, it first needs to prepare the content in a private page p_p before copying the content into p_s. Reading data via DMA works the other way around.

The general idea for their decryption oracle is to manipulate the content of p_p, before its

content is copied to p_s. For their decryption oracle, they use DMA write operations. To decrypt the memory at address q they copy it into p_p, before it gets copied to p_s. In order to get the GPA of p_p, they use the page fault side channel. They demonstrated their ideas based on DMA operations related to OpenSSH network traffic.
For the decryption oracle, they are limited to the packets sent by the VM. Further, they show that they can make their oracle harder to detect by only overwriting parts of p_p that contain known metadata spanning at least a whole 16-byte aligned block. This way, they can restore the overwritten parts before sending the package over the network. Assuming a packet rate of 10 packets per second they showed that their decryption oracle has a throughput of about 200 B/s.
For the encryption oracle, they can also use self-generated network packets. Since only the VM can decide whether it wants to process a package or not, all network traffic addressed to it gets copied into the VM. If there is no service listening for a packet, the VM simply drops the packet. Thus, their encryption oracle can also be used if there is no network service running in the VM. However, a high rate of dropped packages might arouse suspicion. They did not give any data for the throughput of the encryption oracle. But since the construction is similar to the decryption case, its throughput should scale in a comparable manner with the packet rate.
For the encryption oracle, they do not state whether the idea of replacing the payload with known metadata can be applied. If this is not possible, the VM can observe the packages that get destroyed by the encryption oracle. Our encryption oracle is not affected by such problems, because we take over the control flow that processes the data, instead of trying to manipulate data used by the regular control flow.
Like we have shown above, our encryption oracle reaches a slightly higher throughput with our prototype implementation, although they based their measurements on an SSH packet rate of 10 pps, which is quite high for user-generated input (one packet roughly equals one keystroke). Since we do not depend on I/O, we can achieve our throughput independently of the rate of network packages. While they claim that their approach can be applied to any DMA I/O performed by the VM, it is unclear which of them sport known metadata that spans at least a 16-byte aligned memory block in order to make the attack stealthy.

7 Countermeasures

Our code injection attacks, as well as the injected 16-byte encryption oracle, build on the missing integrity protection, the reverse-engineered tweak values, known plaintext and the page fault side channel. The high-performance `cpuid` encryption oracle from the case study in Subsection 5.4.1 also requires that the `cpuid` instruction is interceptable by the hypervisor. In the following sections, we discuss how changes in these areas influence our attack.

7.1 Integrity Protection

With cryptographic integrity protection, the cryptosystem could detect blocks created with the cipher block moving approach. This would prevent us from injecting code/data into the VM, mitigating the attacks presented in this thesis, as well as all of the related work mentioned in Chapter 6 with the exception of the application fingerprint presented in [WMA+19] and the attacks on the AMD SP from [BWS19]. In a recent talk, [Kap19] David Kaplan (AMD) provides first information on a planned extensions of SEV, called SEV Secure Nested Paging (SEV-SNP). Instead of adding strong, cryptographic integrity protection, they propose a new mechanism called Reverse Map Table (RMP) to prevent the hypervisor from writing to pages used by the guest. In order to manage this structure, they plan to introduce an x86 instructions set extension. However, at the time of writing, no whitepaper with precise technical information on SEV-SNP exists.

7.2 Tweak Function

Without the knowledge of the tweak values, we could no longer predict the effect of a cipher block move. [LZL19] claims that "Future versions of the tweak function will be implemented as $T(k, a)$ where a is the physical address and k is a random input that changes after every systems boot". For the non XEX version of the encryption scheme, considered by them, this would not make any difference, since our method from Chapter 4 can be implemented in a kernel module to recalculate the tweak values at run time, with very little overhead. For the XEX version, discovered by us, we demonstrated in Section 4.2 how to brute force the tweak values at run time, as long as they stay 32 bit periodic (or have a similarly low periodicity). While the tweak recovery process takes about 30 min-

utes per tweak, we want to stress that the decision to reboot is under the control of the malicious hypervisor. However, we are unaware of any method to directly calculate the tweak values, like it was possible with the previous version. We believe that using 128-bit randomized tweak values are a mitigation to this attack vector.

7.3 Fixing the Page Fault Side Channel

Currently the hypervisor can manipulate the NPT to provoke page faults. In theory, the page faults should contain the full GPA, which is the Guest Frame Number (GFN) as well as the page offset where the faults occurred. In our experiments, we were only able to get the page offset for write faults. In our opinion, completely removing the hypervisor's ability to observe the page faults of the VM is not realistic, since the hypervisor needs this information for memory management purposes. However, we believe that there are two realistic approaches to reduce the amount of information leaked by this side channel.
For the first approach, the VM no longer shares the page offset, but only the GFN of the fault with the hypervisor. The memory management of the host OS should not be affected by this, as it works at the page level anyway. However, this change would only make our attack slightly more complicated. While there are some cases where the offset is fixed and known, like for our attack on KASLR, this is not the case for the offset of the stack, which we need for our injected 16-byte oracle. In order to work around a masked page offset, we can modify the stack detect gadget from Subsection 5.5.2 to the following:

```
1 cpuid
2 push rdi
3 cpuid
4 push rdi
5 push rdi
6 cpuid
```

Lines 1 to 3 are equal to the original stack detect gadget, but now only yield the GFN of the stack. Before we resume from the second `cpuid` in line 3, we take a copy of the page to which the just obtained GFN points. Next, we push two 8-byte values to the stack at lines 4 and 5. When we reach the third `cpuid` at line 6, we simply compare our copy of the page to its current state to get the offset of the stack pointer inside the page. Although we can only observe changes with a 16-byte granularity (as we can only see ciphertext blocks), this is sufficient since the stack is either 8- or 16-byte aligned. If the two push operations from lines 4 and 5 only result in one changed ciphertext block, the stack was 16-byte aligned at the changed block. If two ciphertext blocks have changed, the stack was 8-byte aligned at the first changed block. The usage of the page fault side channel as a replacement for intercepted instructions is not affected by these changes.

For the second approach, we assume that the hypervisor's ability to manipulate the NPTs gets restricted, so that it is no longer possible to manipulate certain bits in the NPTs, like the ones responsible for write or execute permissions. This way we could no longer provoke page faults, but only observe page faults that are "naturally" triggered by the VM. Implementing this approach would however most likely need major architectural changes, like instruction set extensions. On the other hand, it would make our attack significantly harder or even infeasible, depending on the availability of intercepted instructions as well as the RAM size of the VM. For the stack detection gadget, we could still use the same strategy as in the previous paragraph. But, since we are no longer able to provoke a page fault, allowing us to at least get the GFN of the stack, we would now have to dump all of the VM's RAM that has ever been written to. Furthermore, we may have to change a larger portion of the stack to reliably identify the GPA of the stack. Depending on the RAM size, this could take a significant amount of time. In order to interrupt the VM at precise points in time, we would now dependent on intercepted instructions. Another problem is detecting whether the VM has started executing our gadget, if we are no longer able to remove the execute permission. A possible solution could be to issue a unique sequence of intercepted instructions, to mark the start of our gadget.
We thus conclude, that closing the page fault side channel most likely requires introducing major architectural changes. However, even in this case, the hypervisor's control over physical memory could still be exploited to track the VM's memory usage, by observing "natural" page faults.

7.4 Availability of Known Plaintext

For our attack, we used the Linux kernel itself as a source of known plaintext. We split our analysis in two parts: Knowing the plaintext and finding it in the memory.
As customers most likely use a common Linux distribution, it can be assumed that they are running the kernel supplied by the respective distribution. Furthermore normal disc encryption setups do not encrypt the `/boot` partition from which the kernel gets loaded at boot, allowing the attacker to read the kernel binary in plaintext.
Another approach is using the technique presented in Werner et al. [WMA+19] (see related work in Chapter 6). They showed how to use IBS to reliably fingerprint specific application versions running in SEV-ES secured VMs based on the distance (measured by the GVA) of executed return instruction of an application. One of the problems they had to overcome was separating GVAs/instruction tuples by the instruction that issued them, based on the GVAs. In their algorithm for this, they state that it is easy to filter the instructions belonging to the Linux kernel, due to the large addresses. While they only

evaluated their approach for user space applications, their result is still applicable to the Linux kernel, and it could also potentially perform even better due to the easier data separation. Another approach could be to apply their technique to kernel functions, whose GPAs gets leaked by the page fault side channel.

The bootstrap part of the kernel gets loaded to a fixed GPA. Thus we can use the page fault side channel to infer when it is loaded into memory in order to copy it into our database. The load address of the Linux kernel is randomized by KASLR. There are several strategies to break KASLR in the malicious hypervisor scenario. In Subsection 5.4.1 we showed how to completely disable KASLR with an injection attack. However, the guest might get suspicious if the kernel is always loaded to the same address. In our experiments, we observed that the naturally occurring page faults during VM startup are enough to locate the kernel in memory. So, even if the hypervisor can no longer manipulate the NPTs to provoke page faults, we are able to locate the kernel and use it as a plaintext source. A third method is presented in [HB17]. They show that the kernel location can be detected at runtime, by removing the execute permissions from all pages, injecting an interrupt and observing the occurring page faults. However, this is only possible if we are still able to manipulate the NPTs.

7.5 Emulated Operations

Whether an instruction like `cpuid` or `rdtsc` is intercepted by the hypervisor can be configured in the control area of the VMCB [Adv19]. The VMCB gets encrypted and integrity protected upon a `#VMEXIT`. Furthermore, it is part of the initial attestation [Kap17]. Thus it cannot be manipulated by a malicious hypervisor. The high-performance encryption oracle we built in Subsection 5.4.1, by using the exposed registers of the `cpuid` operation during instruction emulation, can thus be mitigated by disabling interception of the `cpuid` instruction [Adv19]. However, as already mentioned in Section 2.1, emulation of instructions is an important virtualization feature since it allows fine-grained control over exposed hardware features as well as simulating a consistent environment after live migration. A better-suited approach would be to check the returned values for plausibility. [Kap19] suggests that future SEV versions offer such functionality.

Like explained in Section 5.5, the injected 16-byte encryption oracle can be implemented without relying on hypervisor emulated operations. For this, we split the instructions over two pages and use the page fault side channel with active manipulation of the NPTs.

7.6 Detection

Another important aspect, besides direct countermeasures, is how hard it is to detect an attack in the first place. In the scenario of a malicious hypervisor spying on its VMs, the detection of an attack could be enough for the guests to switch to another service or pursue legal matters. We analyze two possible ways of detecting our attack. The first is detecting the changed code itself, the second is detecting abnormal behavior.
Since SEV itself does not provide any integrity protection for the RAM content, this must be done in software and inside the guest. However, the injection could take place at any memory location that gets executed during the lifetime of the VM. This makes for a very large search space. Furthermore, our injected code is only temporarily visible, as we can take a copy of the original memory content before performing our injection attack. This allows us to restore the original state after we are done. Finally, the program that inspects the guests' RAM content, in order to find changed code, cannot be certain that its own code is unchanged.
The second approach is to detect abnormal behavior, like the unusual kernel base address when disabling KASLR. However, detecting more transient abnormal behavior, like a random number generator that got faulted for a short period of time, or a mapping for a shared page, is more difficult. Like for detecting changed code, the attack surface for changed behavior is quite large.
Finally, we want to stress that while our injected code hijacks the original control flow, we are able to restore it. For this, we simply need to jump back to the location where we took over the control flow, after restoring the original content. Since the jump back itself obviously cannot be overwritten before being executed, it must be at a memory location that is not immediately executed after resuming the original control flow. For example, it could be located in the block right before the one that we jump back to. This way, we have enough time to overwrite the jump back instruction itself before the natural control flow return to this location. The content of registers can be stored on the stack before they are modified and restored at the end of the injected code.

8 Conclusion & Outlook

In this thesis, we have analyzed the security properties of AMD's SME/SEV memory encryption technology. The main goal of SEV is to provide full isolation between the hypervisor and its VMs by encrypting the VM's RAM and the VMCB with a key, that is not accessible to the hypervisor. This way, customers could use rented VMs to process sensitive data, without having to trust the hosting provider not to spy on their data. While VM based cloud solutions have been a major success, data privacy concerns are one of the major reasons why some customers do not want to use them. However, in its current state, SEV fails to achieve full isolation, as related work and this thesis have shown that is possible to construct encryption and decryption oracles for SEV secured VMs. This way, a malicious hypervisor can read sensitive data from the VM's RAM or move his own data into the VM, breaking the full isolation. Nonetheless, SEV makes such attacks significantly harder, compared to the previous state of affairs, where the hypervisor could simply read from/write to all of the VM's RAM in plaintext. The attacks from related work were either based on a firmware bug of the SP or manipulated I/O operations performed by the VM. Some of them even require that the attacker has to generate the I/O traffic himself, which greatly increases the risk of detection.

In this thesis, we have demonstrated that the manipulation of I/O operations is not necessary to construct an encryption or decryption oracle. In fact, we do not require the VM to perform any I/O operations, making our attack hard to detect and applicable in a broad range of scenarios. Instead, we have shown in Chapter 5 that it suffices to simply move ciphertext blocks containing known plaintext, from one memory location to another to construct a first encryption oracle, that allows us to partially control the plaintext to which a 16-byte ciphertext block decrypts. For this we exploited, that it is possible to reverse engineer the address-based tweak values used in SEV's XE-based encryption mode, as we have shown in Chapter 4. While this was partially known for the older AMD 7xx1 product line, we have provided additional detail on the analysis. Besides, we have leveraged the fact that SEV does not provide any integrity protection. Furthermore, we have discovered, that the newer AMD 3xx1 product line uses a new, enhanced XEX-based encryption mode. Nevertheless, we have been able to prove that it is still vulnerable to our attack, as it is still possible to reverse engineer the tweak values. However, computing them now requires a much higher computational effort. By knowing the address-based tweak values, we are able two predict how the plaintext of a ciphertext block changes, when copying the

ciphertext block to a new memory location. Given about 8 MB of known plaintext, this enables us to control 4 consecutive plaintext bytes every two 16-byte ciphertext blocks. Based on this, we have bootstrapped a full encryption oracle that can control all 16-bytes of a ciphertext block. This allows us to move data into the VM, which in turn enables us to execute arbitrary code inside the VM. Moreover, we discuss how code execution can be used to build a decryption oracle.

In the late summer of 2019, AMD released the 7xx2 product line of Epyc processors, that is based on their new Zen 2 architecture. Our first analysis has shown that it uses a much stronger tweak function, as the tweak values are no longer 4-byte periodic and seem to be regenerated on boot. However, due to the limited scope of a master thesis, we did not have enough time to perform a full analysis. Anyhow, we do not expect that it is possible to fully mitigate our attack vector, without implementing cryptographic integrity protection, which is able to detect that a ciphertext block is not decrypted at its original location. Without integrity protection, an attacker can always exploit the ability to decrypt a ciphertext block at an arbitrary address, to gain some information on the tweak values. Even without knowing the tweak values, an attacker might be able to place less complex data, simply by trial and error. Since the decryption of a modified AES ciphertext can be modeled as a random mapping, placing 1-byte assembly instructions by copying arbitrary ciphertext blocks, should succeed with a probability of 1/256.
Another approach to decrease our attack's dependency on knowing the tweak values, could be to exploit paging. The main idea is to modify the hypervisor in a way, that it only uses one (or a small number) of HPAs for all of the VM's main memory. As the tweak is dependent on the HPA, this would provide us with many valid ciphertext blocks for each page offset. If we know the plaintexts of these blocks, we could again try to inject instructions, by swapping such blocks. However, we leave exploring the feasibility of this approach to future work.

References

[AAUC18] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):79, 2018.

[Adv05] Advanced Micro Devices. Secure Virtual Machine Architecture Reference Manual. Technical report, Advanced Micro Devices, 2005.

[Adv08] Advanced Micro Devices. Nested Paging. `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`, 2008.

[Adv17] AMD64 Architecture Programmer's Manual Volume 1: Application Programming. `https://www.amd.com/system/files/TechDocs/24592.pdf`, 2017.

[Adv18] Advanced Micro Devices. Secure Encrypted Virtualization API Version 0.16. `http://support.amd.com/TechDocs/55766_SEV-KM%20API_Specification.pdf`, 2018.

[Adv19] AMD64 Architecture Programmer's Manual Volume 2: System Programming. `https://www.amd.com/system/files/TechDocs/24593.pdf`, 2019.

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.

[AMD] AMD SEV Resource Center. `https://developer.amd.com/sev/`. Accessed: 2019-11-26.

[ATG+16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, pages 689–703. USENIX, 2016.

[Aum17] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 2017.

[BCD+18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel {SGX}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1213–1227, 2018.

[BGF16] Johannes Bauer, Michael Gruhn, and Felix C Freiling. Lest we forget: Cold-boot attacks on scrambled ddr3 memory. *Digital Investigation*, 16:S65–S74, 2016.

[BGN+17] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault Attacks on Encrypted General Purpose Compute Platforms. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 197–204. ACM, 2017.

[BHP18] Robert Buhren, Felicitas Hetzelt, and Niklas Pirnay. On the detectability of control flow using memory access patterns. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 48–53. ACM, 2018.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[BWG+16] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, page 14. ACM, 2016.

[BWS19] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19. ACM, 2019.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX Explained. `https://eprint.iacr.org/2016/086.pdf`, 2016.

[DYM+17] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *arXiv preprint arXiv:1712.05090*, 2017.

[GB09] Craig Gentry and Dan Boneh. *A fully homomorphic encryption scheme*, volume 20. Stanford University Stanford, 2009.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EUROSEC*, pages 2–1, 2017.

[GHC14] Oliver Gasser, Ralph Holz, and Georg Carle. A deeper understanding of ssh: results from internet-wide scans. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9. IEEE, 2014.

[Gue16] Shay Gueron. Memory encryption for general-purpose processors. *IEEE Security & Privacy*, 14(6):54–62, 2016.

[HB17] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 129–142, New York, NY, USA, 2017. ACM.

[HLP+13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11, 2013.

[HSH+08] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *usenix-security*, 2008.

[Int17] Intel. Intel Architecture Memory Encryption Technologies Specification. `https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf`, 2017.

[Int19] Intel Architecture Memory Encryption Technologies Specification, April 2019.

[JACH11] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 272–283. IEEE, 2011.

[Kap17] David Kaplan. Protecting VM Register State with SEV-ES. White Paper, February 2017.

[Kap19] David Kaplan. Upcoming x86 technologies for malicious hypervisor protection. `https://static.sched.com/hosted_files/lsseu2019/65/SEV-SNP%20Slides%20Nov%201%202019.pdf`, November 2019. Accessed: 2019-11-23.

[KCV18] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with intel software guard extension (intel sgx). *arXiv preprint arXiv:1802.00508*, 2018.

[KPM+16] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N Asokan, Andrew Simpson, and Robin Ankele. Exploring the use of intel sgx for secure many-party applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, page 5. ACM, 2016.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. Technical report, Advanced Micro Devices, 2016.

[Len19] Thomas Lendacky. Improving and expanding sev support. `https://static.sched.com/hosted_files/kvmforum2019/61/KVM_Forum_2019_SEV.pdf`, 2019. Talk at KVM Forum 2019, Accessed: 2019-11-26.

[LJJ+17] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 523–539, 2017.

[LRW02] Moses Liskov, Ronald L Rivest, and David Wagner. Tweakable block ciphers. In *Annual International Cryptology Conference*, pages 31–46. Springer, 2002.

[LSG+17] 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. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.

[LZL19] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[MH19] Mathias Morbitzer and Manuel Huber. Github - SEVered Framework. `https://github.com/Fraunhofer-AISEC/severed-framework/`, 2019.

[MHH19] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting Secrets from Encrypted Virtual Machines. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, pages 221–230. ACM, 2019.

[MHHW18] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security*, EuroSec'18, pages 1:1–1:6, New York, NY, USA, 2018. ACM.

[MZLS18] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of intel sgx and amd memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, page 9. ACM, 2018.

[Rog04] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 16–31. Springer, 2004.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[TPV17] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017.

[UNR+05] 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. *Computer*, 38(5):48–56, 2005.

[WCP+17] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, New York, NY, USA, 2017. ACM, ACM.

[WMA+19] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, pages 73–85. ACM, 2019.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[XLC13] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–257. IEEE, 2013.

[XTS19] IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2018 (Revision of IEEE Std 1619-2007)*, pages 1–41, Jan 2019.

[YADA17] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 313–324. IEEE, 2017.