SEVurity: No Security Without Integrity
Breaking Integrity-Free Memory Encryption with Minimal Assumptions

Luca Wilke*, Jan Wichelmann*, Mathias Morbitzer†, Thomas Eisenbarth*
*University of Lübeck, Lübeck, Germany
{l.wilke,j.wichelmann,thomas.eisenbarth}@uni-luebeck.de
†Fraunhofer AISEC, Garching, Germany
{mathias.morbitzer}@aisec.fraunhofer.de

***Abstract*—One reason for not adopting cloud services is the required trust in the cloud provider: As they control the hypervisor, any data processed in the system is accessible to them. Full memory encryption for Virtual Machines (VM) protects against curious cloud providers as well as otherwise compromised hypervisors. AMD Secure Encrypted Virtualization (SEV) is the most prevalent hardware-based full memory encryption for VMs. Its newest extension, SEV-ES, also protects the entire VM state during context switches, aiming to ensure that the host neither learns anything about the data that is processed inside the VM, nor is able to modify its execution state. Several previous works have analyzed the security of SEV and have shown that, by controlling I/O, it is possible to exfiltrate data or even gain control over the VM's execution. In this work, we introduce two new methods that allow us to inject arbitrary code into SEV-ES secured virtual machines. Due to the lack of proper integrity protection, it is sufficient to reuse existing ciphertext to build a high-speed encryption oracle. As a result, our attack no longer depends on control over the I/O, which is needed by prior attacks. As I/O manipulation is highly detectable, our attacks are stealthier. In addition, we reverse-engineer the previously unknown, improved Xor-Encrypt-Xor (XEX) based encryption mode, that AMD is using on updated processors, and show, for the first time, how it can be overcome by our new attacks.**

I. Introduction

Virtual Machines (VMs) are a very important part of today's cloud computing market. They significantly ease technical aspects of hosting. On the customer side they allow for flexible resource scaling, due to their low setup time. Furthermore, multiple VMs can run on the same physical machine, because the hypervisor – the software that manages the virtualization – provides one sided isolation, by preventing the VM from accessing other software running on the host. Thus, the hosting provider can make better use of its hardware.

Apart from that, potential customers still cite data privacy concerns toward cloud service providers as a main reason not to adopt cloud solutions, especially in cases where the hosting location within a given jurisdiction cannot be guaranteed. Performing sensitive computations in VMs requires the customer to fully trust the hypervisor, since the hypervisor has direct access to all virtualized resources.

Security solutions like full disk encryption only partially address this issue, since the data is still vulnerable when being decrypted and stored in the RAM at run time.

Providing full isolation between the hypervisor and the VM has been studied extensively by researchers as well as industry [9, 21, 22, 23, 24, 34]. Intel Software Guard Extensions (SGX) [9, 20, 27] 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 [17]. In 2016, AMD introduced Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) [24] to protect the entire system 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 [8]. In February 2017, AMD introduced SEV Encrypted State (SEV-ES) [23], 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 modifications to the Linux kernel. According to AMD, the corresponding patches are mostly finished, however support for SEV-ES has not been mainlined, yet [25]. Intel is also working on a solution similar to SME/SEV, called Total Memory Encryption (TME)/Multi-Key Total Memory Encryption (MKTME) [21], but did not yet publish corresponding processors. A detailed comparison between Intel SGX and AMD's memory encryption can be found in [28].

This work focuses on AMD's solutions for providing full isolation between hypervisor and VM, as it is the most prevalent full memory encryption. All prior attacks have either been mitigated by SEV-ES or used I/O to move known plaintext into encrypted pages. We show that our attack vector is available even without user-controlled I/O or access to unprotected I/O operations. Instead, we only require minimal knowledge about the system to compromise and completely take over the VM. To achieve this, we bootstrap an encryption oracle from just a few megabytes of known plaintext, allowing us to place and execute arbitrary code in the VM. We identify the lack of integrity protection as the main reason for this weakness, and postulate that full security against our attacks can only be achieved by implementing a proper integrity protection scheme. As an independent contribution, we also show that AMD's updated XEX-based memory encryption mode is still vulnerable to the previous attacks.

A. Our Contribution

- We exploit the missing integrity protection of SEV to place arbitrary code in a SEV-ES secured VM, *without relying on any I/O operations*.

- For this we bootstrap an encryption oracle just by moving existing ciphertext within the VM's memory.
- We show the security impact of the emulated `cpuid` instruction by abusing it to create a high-performance encryption oracle.
- We reverse engineer the new XEX encryption mode, that is used on updated processors, and infer the associated tweak values. We show that this mode is just as vulnerable as the previous XE-based encryption mode which has been exploited by prior attacks.
- We discuss previously proposed and new countermeasures, and evaluate their impact on our attacks.

II. Background

A. AMD Memory Encryption

Secure Memory Encryption (SME) To protect against an attacker with physical access to a system, AMD introduced SME in 2016 [24]. SME encrypts data before writing it to RAM, which renders it useless for an attacker attempting to access the data, e.g., via a cold boot attack or Direct Memory Access (DMA) [10, 18, 36]. The encryption and decryption are controlled by the Secure Processor (SP), an ARM-based co-processor.

A special bit in the page table – the so-called C-bit – is used to indicate whether a page should be encrypted [24]. However, changing the C-bit does not change the content of the page, only whether it is interpreted as encrypted or not. 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. In case the Operating System (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.

When the system boots, a random memory encryption key is created and stored in the SP [24]. Subsequently, memory writes are encrypted, and memory reads are decrypted. For both the encryption and decryption, the SP uses AES in conjunction with a physical address-based tweak.

Secure Encrypted Virtualization (SEV) The SEV technology was introduced together with SME in 2016. The goal of SEV is to protect a VM from a malicious or compromised hypervisor. This is achieved by using the memory encryption technology of SME with a different encryption key for each VM and the hypervisor itself. All keys are stored in the SP and are not accessible by any other party.

Due to the different encryption keys, a hypervisor attempting to access data of a VM would get a decrypted version using the hypervisor's key, rendering the generated plaintext useless. However, the VMs can use the hypervisor's key to intentionally share information.

A known issue of SEV is the lack of encrypting the Virtual Machine Control Block (VMCB), which is a data structure describing the state of a VM. It includes information like the VM's configuration and register contents. It also provides means for communication between the hypervisor and the VM: For example, if the VM exits due to an interrupt, the processor stores appropriate metadata (e.g., a memory address) in this structure. The lack of encryption can be exploited to manipulate the execution flow of the VM and leak sensitive information like [19, 33] have shown.

SEV Encrypted State (SEV-ES) To address these issues, AMD introduced SEV-ES [23] 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. They introduced a new exception, which 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 `#VMEXIT`. When the VM is resumed, it can copy the data back to its VMCB.

Encryption mode Like SME, SEV and SEV-ES provide drop-in memory encryption, but for VMs. The memory encryption has no ciphertext expansion, which means that structure and size of the memory remain unchanged with and without encryption. Similar techniques like Intel SGX [14] store Message Authentication Code (MAC) tags for each memory block, which allows for strong integrity protection, but comes with significant overhead. In contrast, AMD does not store any integrity protecting metadata, which is very convenient for the user in terms of transparency and space-efficiency.

AMD achieves an implicit block level integrity protection through the encryption: Changing any bits in a ciphertext block results in a garbled and for the attacker unpredictable plaintext block. Also, the usage of an address-based tweak should make it difficult to decrypt a valid ciphertext at another address and get a meaningful plaintext. One thus has to assume that the VM execution will eventually halt if it encounters random data blocks (i.e. invalid opcodes or state variables) – there are no means of reliably detecting whether the ciphertext has been tampered with.

Since both SME and SEV use the same technique for the encryption process, they suffer from the same problems regarding integrity. These problems are even more severe in the case of SEV, because an attacker with hypervisor permissions can easily manipulate or copy the RAM content of a VM.

B. Memory Encryption using Tweakable Block Ciphers

One popular method for storage encryption are tweakable block ciphers, such as AES-XTS [1], which is, e.g., used in Apple's FileVault, MS Bitlocker and Android's file-based encryption. Tweakable block ciphers provide encryption

without data expansion as well as some protection against plaintext manipulation. A *tweak* allows to securely change the behavior of the block cipher, similar to instantiating it with a new key, but with little overhead. If the tweak is, e.g., a function of the block address, the encryptions for each block appear independent, which prevents numerous attacks on the ciphertext, such as frequency analysis, block moving and several more. However, without proper integrity protection, several attacks remain possible, i.e., randomizing the plaintext (by altering the ciphertext), replaying old values and traffic analysis by monitoring location-specific changes.

AES-XTS includes ciphertext stealing, which allows expansion-free block encryption for arbitrary-length plaintexts by using previous ciphertext for padding. Memory in RAM and the uncore part of the CPU is always handled in blocks of 64 bytes, which is a multiple of the 16 byte block size of AES. Thus, ciphertext stealing is not needed, reducing the XTS mode to the original Xor-Encrypt-Xor (XEX) mode by Rogaway [32]. XEX and Xor-Encrypt (XE) are methods to turn a block cipher such as AES into a tweakable blockcipher, where a tweak-derived value is XORed with the plaintext before encryption (and XORed again after encryption in the case of XEX).

C. Nested Paging

On most common OSs, processes use Virtual Addresses to access data [15]. Those VAs are translated into Physical Addresses, which determine where the data is located in the physical memory. The mappings between VAs and PAs are stored in the page table.

On virtualized systems, two different page tables are used. Within the VM, the VA used by the guest, the Guest Virtual Address (GVA), is translated to the Guest Physical Address (GPA). The GPA is the address which the VM considers to be the PA. However, on the host system itself another page table is introduced, to allow multiple VMs to run on the same physical machine. This second page table is called the Second Level Address Translation (SLAT), or Nested Page Table (NPT) [5]. The NPT translates the GPA into the Host Physical Address (HPA), the actual address of the data in physical memory.

When SEV is active, the page table in the guest is encrypted and thus not accessible by the hypervisor. However, the hypervisor is still responsible for managing the NPT. This allows the hypervisor to infer information about the VM's memory assignment by monitoring the entries in the NPT. The NPT cannot be accessed by the VM. It is therefore not possible for the VM to prevent the hypervisor from overwriting permissions in the NPT. Multiple attacks make use of this possibility to gather information about where the VM stores critical data [12, 19, 26, 30, 31].

D. Instruction Interception

In a virtualized environment, there are two reasons which cause a VM to trigger a `#VMEXIT`, which hands control back to the hypervisor. One reason are interrupts and exception handlers, which need hypervisor assistance, e.g., page faults due to swapped pages.

The second reason is the interception of special instructions [6]. Two prominent examples for this are the `cpuid` and the `rdtsc` instruction. The `cpuid` instruction allows querying a wide span of CPU information, including an accurate model number, a list of supported features and the system's topology. Changing the returned registers allows the hypervisor fine grained control over the hardware features it exposes to the guest. The `rdtsc` instruction returns the current state of the 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 an instruction is intercepted or not can be configured in the VMCB.

E. Previous Attacks on SEV

Manipulating VMCB Hetzelt and Buhren [19] explore the idea of manipulating the general purpose registers stored in the VMCB to create an encryption/decryption oracle. In order to move data from memory into a register or vice versa, they manipulate the `RIP` register in the VMCB, to construct corresponding gadgets. SEV-ES mitigates these attacks.

I/O-based attacks Du et al. [16] build an encryption oracle, which is based on self-generated network traffic. They require that an Nginx web server is running in the VM and exploit its memory management behavior, allowing them to locate the content of specifically crafted, self-generated HTTP packets in the VM's RAM.

Morbitzer et al. [31] leverage the hypervisor's control over the NPT in order to swap GPA mappings. In combination with a network service running inside the VM, which returns some resources on request, they build a decryption oracle. In the first phase, they locate the GPA where the response of the network service is stored by repeatedly sending requests and monitoring the page fault side channel. In the second phase, they manipulate the NPT so that the GPA of the returned resource points to another memory location. Thus, the content of this memory location is returned on the next request. In their follow-up work [30], they show how to locate GPAs that might contain secret data, like encryption keys.

Li et al. [26] exploit the fact that DMA operations issued by the VM are currently performed via an unencrypted bounce buffer. They demonstrate that this can be used in combination with network I/O, to create an encryption/decryption oracle. If the VM performs network I/O, the packets are copied to the bounce buffer, before they are processed by the network card. To create an encryption oracle, they manipulate incoming data in the bounce buffer before the VM copies it into its private memory. For the decryption oracle, they manipulate the data that the VM wants to send before it gets copied from the VM's private memory into the bounce buffer. To detect the memory locations and hit the correct timing, they use the page fault side channel.

Fingerprinting Applications Werner et al. [33] showed two independent results. First, they use the unencrypted VMCB to reconstruct code executed in the VM by singlestepping the VM while observing the changes to the unencrypted register values in the VMCB. Like [19] they also use the unencrypted VMCB to encrypt/decrypt data. This is mitigated by SEV-ES. In their second result, they show how to use a performance counter subsystem called Instruction Based Sampling (IBS) to detect which applications are running in the VM. They leverage that IBS leaks the GVA of return statements, and show, that the distance between return statements uniquely identifies specific versions of applications. They claim that the guest cannot detect whether IBS is activated. This result holds under SEV-ES.

Security of AMD-SP In [13] Buhren et al. take 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, 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.

Data Faults In [11], Buhren et al. explore the idea of performing classical fault attacks on application data in memory. They flip a bit in a ciphertext block, in order to create garbled plaintext. They demonstrate how to use this to perform a fault attack on RSA CRT. They implemented their attack for SME and required that the attacker is able to run an unprivileged application and can perform DMA memory access. However, it should also be possible to migrate this attack to SEV.

III. 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 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 [16], 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 [16].

A. XE Encryption Mode

According to [16], the processor contains 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

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

This means, that for each physical address bit the respective tweak constant is XORed, if that bit is 1.

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

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

Similarly, decryption of a ciphertext c uses the inverse transformation

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

This construction is a variant of the XE mode of operation [32].

We can exploit the missing integrity protection, to compute all tweak constants t_i: We encrypt a block m with physical address p, copy the ciphertext to other addresses q_j and decrypt it there. By doing this, the tweak values of the source address p and the target addresses q_j are XORed:

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

This allows us to build a system of linear equations, whose solution are the tweak constants:

$$\begin{pmatrix} p \oplus q_1 \\ p \oplus q_2 \\ \vdots \\ p \oplus q_{n-4} \end{pmatrix} \cdot \begin{pmatrix} t_{n-1} \\ t_{n-2} \\ \vdots \\ t_4 \end{pmatrix} = \begin{pmatrix} m \oplus T(p \oplus q_1) \\ m \oplus T(p \oplus q_2) \\ \vdots \\ m \oplus T(p \oplus q_{n-4}) \end{pmatrix}.$$

TABLE I. The first three tweak constants on an Epyc 7251 processor. We denote the first one as t_4, 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 first few constants are shown in Table I. 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 [16], 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 led to repeating patterns within an encrypted page, if the plaintext was all zeroes; some time later, after applying several operating system 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.

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.

B. Updated XEX Encryption Mode

We conducted the same experiments on an Epyc Embedded 3151 processor, which was released about 8 months after the Epyc 7251, and on an Epyc 7401P processor, which was released together with the Epyc 7251. On both processors, the system of linear equations did not have any solutions, 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

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

and the matching decryption function

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

As these equations show, AMD chose to use the XEX [32] 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 calculating the tweak constants, since simply decrypting a ciphertext at a different position does not yield usable results anymore:

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

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

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

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. We managed to partially work around this penalty by parallelizing our guesses, taking only 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.

In summary, we showed that 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 two sections, we will exploit this fact and show how known plaintext can be used to place arbitrary code and data in the encrypted VM.

IV. Cipher Block Moving Attack

As we have seen in the previous section, we can compute the tweak values for any physical address. In this section we show how a malicious hypervisor can use the knowledge of the tweak values together with known plaintext and 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 Section V. In Section VI, 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 [16, 26], 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.

A. 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 guest kernel's binary, which might be due to the unencrypted `/boot` partition or by using fingerprinting (see Section VIII-D). 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.

B. 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, which has first been introduced in the context of Intel SGX [35]. A schematic overview can be found in Figure 1. Since we control the hypervisor, and therefore the host page table, we can mark the relevant VM pages as *not writable* or *not executable*. If the VM then tries to

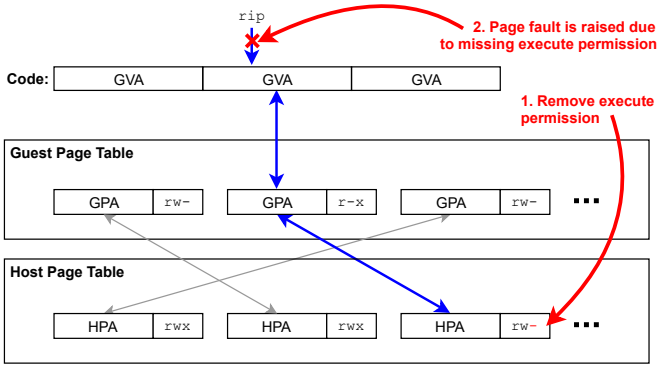

Fig. 1. Page fault side channel. When the VM tries to execute an instruction, the GVA which the program counter (`rip`) is pointing to has to be resolved to a HPA. This is accomplished by performing a walk through the NPTs, while checking the respective permission flags. The hypervisor can force the VM to page fault by removing the execute flag in the host page table entry. Subsequently, the hypervisor learns which page the VM tried to execute. The same method can be used for detecting memory writes, by clearing the write flag instead.

issue a memory write or execute an instruction, a page fault is triggered and the corresponding hypervisor interrupt handler is called. The page fault exception contains the GPA where the fault occurred. The execution of the VM can be resumed by marking the VM's page as *writable* respectively *executable* within the host page table.

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 and is loaded to consecutive GPAs. On modern Linux kernels, the base GPA, to which the kernel gets loaded, is randomized by Kernel Address Space Layout Randomization (KASLR). We present two methods for finding the guest kernel when KASLR is active. The Linux kernel is booted in two steps. First a small bootstrapper is loaded (to a fixed GPA) that, amongst other setup tasks, is responsible for loading the actual kernel and for performing the KASLR. In the first approach, we use our cipher block moving attack to modify the code of the bootstrapper, such that the KASLR code is never executed. That way, the kernel is always loaded to a fixed GPA. We present a more detailed description in case study V-A. For the second approach, we monitor the naturally occurring page faults during VM startup. We exploit, that the kernel is loaded to continuous GPAs and that the memory accesses before loading the kernel to a randomized address, are quite deterministic. This allows us to identify the memory accesses related to loading the kernel, which gives us the GPAs of the kernel.

At the moment, AMD's patched Linux kernel cannot be complied to run as a SEV-ES guest and use KASLR at the same time, as the compile time options for these features exclude each other (see CONFIG_SEV_ES_GUEST and CONFIG_RANDOMIZE_BASE in arch/x86/Kconfig in AMD's kernel repository [2]). We are not aware of any fundamental conflict between these two features and thus suspect that this is only a temporary implementation issue.

We tested these methods on SEV secured machines, but as they do not rely on an unencrypted VMCB they should also work with SEV-ES. As mentioned in the attacker model, the kernel code can be assumed to be entirely known to the attacker and thus 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.

C. 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:

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

To target the XEX encryption mode the copied ciphertext block needs to be slightly adjusted, by adding $T(p) \oplus T(q)$:

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

Decrypting this at address p will then yield m.

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 be able to control at most 4 bytes of any 16-byte block in a reliable way. Finally, for our processors we found that only 28 tweak constants 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 to be able to fully control, if given enough plaintext and memory.

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 copied 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, we can use the `.text` section of the kernel binary itself, after locating it in memory with one of the previously described methods.

D. Code Injection

We now show how the two controlled bytes per block can be used to modify existing VM code, allowing us to redirect the 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. The easiest way to achieve this is by finding a 16-byte aligned instruction and overwriting it with a short branch instruction, like `ret` or `jmp` (Figure 2). This simple modification already suffices to completely disable KASLR (see Section V-A).

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.

To insert other instructions, we propose the layout shown in Figure 3. 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 (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 amount of consecutive bytes that we can control.

In Section V we successfully use this method to disable KASLR and illustrate a fast `cpuid`-based 16-byte encryption oracle. Finally, in Section VI, we build another 16-byte encryption oracle which solely relies on ciphertext block moving and the ability to provoke a context switch between hypervisor and VM at a precise moment in time. We demonstrate how the latter can be achieved by using emulated instructions or page faults. In contrast to the `cpuid`-based 16-byte encryption oracle, this final encryption oracle does not depend on the capability of the hypervisor to modify the result of emulated operations, so it is difficult to mitigate without introducing proper integrity protection. Both encryption oracles allow us to execute arbitrary code within the VM.

V. Attack Case Studies

A. Control Flow Modification for KASLR

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

When loading the kernel, the bootstrapper code calls the function `void choose_random_location(...)`, which is defined in */boot/x86/compressed/kaslr.c*. The function checks whether the user provided the *nokaslr* command line option; if this is the case, it returns immediately. Else the function computes random physical and virtual base addresses for the kernel and writes them into the supplied pointer arguments. So, to disable KASLR, it is sufficient to place a `ret` at an early location in the function.

To find the right point in time to modify `choose_random_location`, we utilize the page fault side-channel as explained in Section IV-B. 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. In our experiments, the targeted function was always located near the end of the bootstrapper's `.text` section, which means that we already have a few MB of known plaintext at this point, depending on the kernel binary. In addition, the `ret` instruction only requires a 1-byte opcode, which greatly reduces the amount of known plaintext that is needed to inject the instruction. As stated above, AMD's patched Linux kernel can currently not be configured to both use KASLR and run as an SEV-ES guest. Thus we only tested this attack on SEV secured VMs (without the ES extension).

B. Using CPUID as an Encryption Oracle

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

As explained in II-A, 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

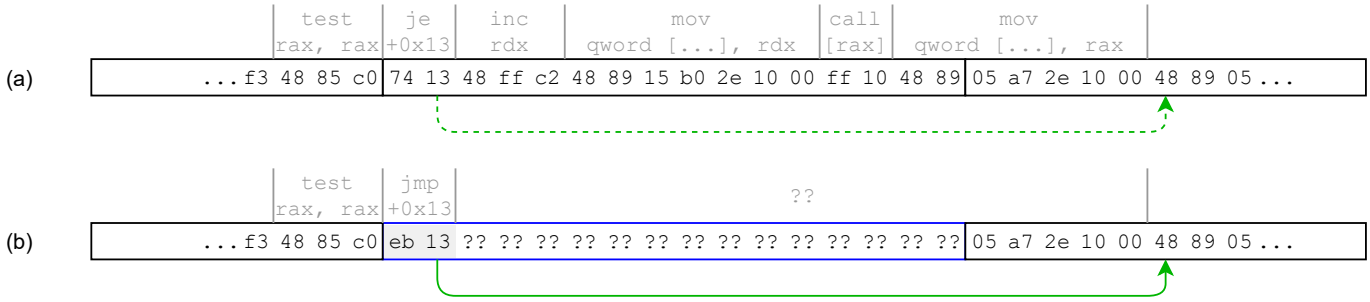

Fig. 2. Example for changing execution flow by replacing one 16-byte block of code. While the base program (a) branches conditionally depending on the value of the `rax` register, the patched version (b) has this branch replaced by an unconditional one. The remainder of the inserted block consists of uncontrolled bytes, which are not expected to form a sequence of meaningful instructions.

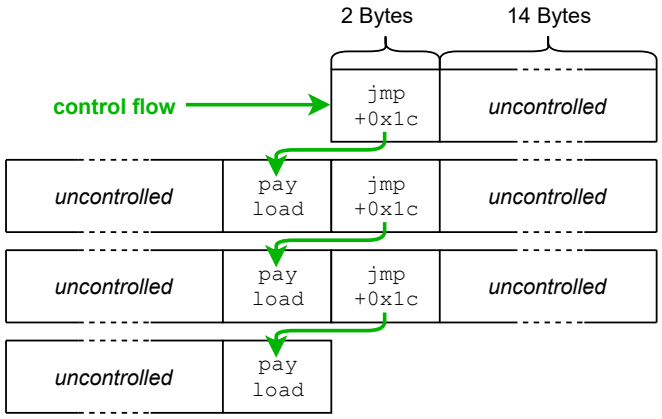

Fig. 3. A sequence of consecutive 16-byte blocks blocks are chained together to get small contiguous chunks of code, which are connected by unconditional 2-byte `jmp` instructions to avoid executing the uncontrolled bytes in between. Thus two bytes of every second block can be used to execute arbitrary 1-byte or 2-byte instructions (payload). The last payload may either redirect to original code (e.g., by returning), or enter a loop.

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 the system's 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 VM/hypervisor context switch 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.

VI. Executing Arbitrary Code

The previous two examples have shown that even little modifications of 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 IV-D, 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 be easily 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.

A. Triggering the 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 is emulated 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. Instead of `cpuid`, we could also use other instructions that are intercepted by the hypervisor and require at most a 2-byte opcode, like `rdtsc` (for a complete list see [6]). As stated in II-D, instruction interception is important, as it, for example, allows fine grained control over exposed hardware features as well as live migration. However, as stated in II-D, the guest owner can choose to disable instruction interception for the VM, with the downside of losing the mentioned functionality.

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.

In the following we will use `<sync>` to express that one of the just described mechanisms must be used, to interrupt the VM at a certain point in time.

B. Finding the Stack

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 synchronisation mechanism with the page fault side channel.

We use the attack primitive from Section IV-D to construct an instruction sequence `<sync>; push rdi; <sync>`. `<sync>` 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. However, on SEV-ES the page offset is masked out. To overcome this, we take a copy of the whole page, while the hypervisor is handling the page fault and compare it with the content of the page at the second `<sync>`. The position of the ciphertext block that was changed by the `push rdi` operation gives us the exact offset of the stack inside the page.

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.

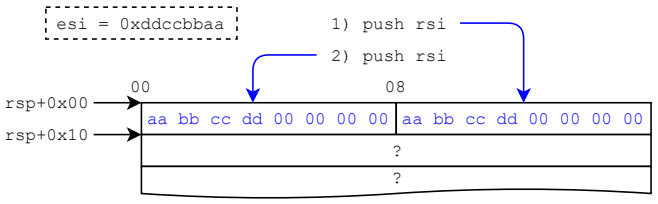

Fig. 4. Layout of stack after pushing the 32-bit `esi` register. The stack pointer is decreased when pushing a register, so, depending on the stack pointer's original alignment, we might have to push the register another time to set the lower address part of a 16-byte block. Since this is a 64-bit operation, 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.

C. 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 instruction'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 4). 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 amount of blocks being created, one might have to add some `pop` instructions to free up stack space before proceeding with the next block.

As a 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 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

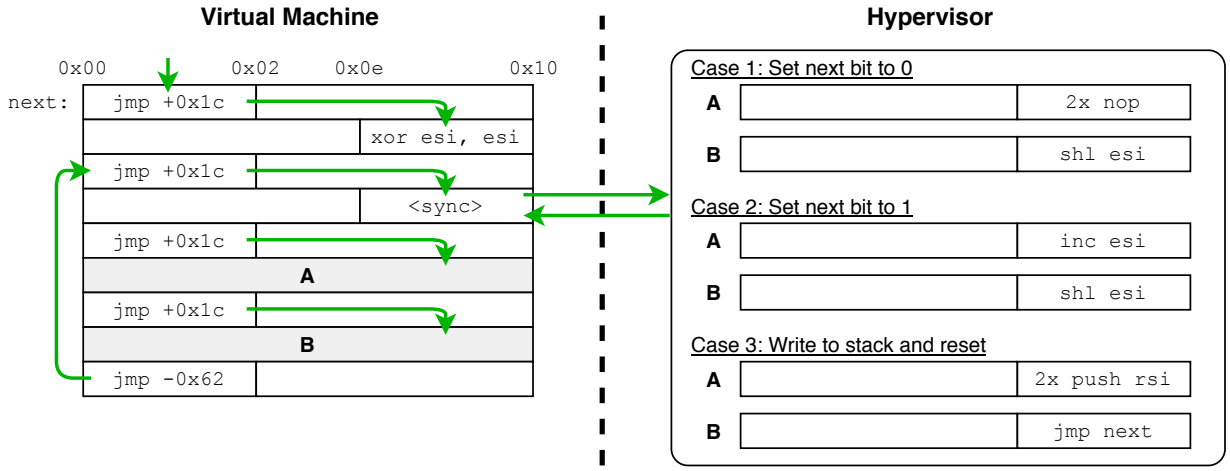

Fig. 5. Schematic of the 4-byte encryption oracle. Each row represents a 16-byte block (not to scale), control flow jumps are denoted by arrows. On each use of the sync mechanism from Section VI-A (for example a `cpuid` call), the hypervisor replaces blocks A and B depending on the desired action: It 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.

Figure 5. 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 bytes 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.

D. 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 increment/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 6 shows an excerpt of a gadget using a 3-byte opcode payload.

Fig. 6. 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.

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.

E. 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 [19].

As explained in Subsection II-A, 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 [26, 31]. 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.

VII. Comparison to Related Work

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

Throughput of our oracle We performed our experiments on two different setups. Initially we used an AMD Epyc 3151 CPU with 16 GB of RAM. The host was running Ubuntu 19.04 with Linux kernel version 5.0.18 and the guest was running Ubuntu 19.04 with kernel 5.0.0-27-generic with 1 GB of RAM. The used QEMU version was 2.12.0. As the BIOS on this machine does not yet support SEV-ES, we also used a machine with an Epyc 7401P processor, that provided the necessary BIOS support, to verify our results under SEV-ES. As mentioned in the introduction, SEV-ES needs extensive software support in the Linux kernel, the QEMU emulator and the UEFI of the VM. We used the versions from AMD's official repositories [2, 3, 4]. We made use of the SEVered framework [29] 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 a thousand 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 via the sync mechanism. 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.

Comparison We compare our results to encryption/decryption oracles constructed in related work. If not stated otherwise all attacks assume a malicious hypervisor. An overview can be found in Table II.

TABLE II. Comparison of different approaches for encryption oracles. [1]Li et al. [26] only specify the decryption rate, but it should be similar to the encryption rate.

	Du et al. [16]	Li et al. [26]	`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

Du et al. [16] 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 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 a constant 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 webserver 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 a similar exploitable behavior. They do not give performance measures.

Li et al. [26] 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 as 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. Furthermore, 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 use self-generated packets, even if there is no service listening. The VM can however observe the amount of dropped packages. 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 a 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.

VIII. 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 V-B also requires that the `cpuid` instruction is interceptable by the hypervisor. In the following paragraphs we discuss how changes in these areas influence our attack.

A. Integrity Protection

With cryptographic integrity protection, the encryption system 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 paper, as well as all of the related work mentioned in Section VII with the exception of the application fingerprinting presented in [33] and the attacks on the AMD SP from [13]. In January 2020, AMD released a whitepaper on a planned future extension called SEV Secure Nested Paging (SEV-SNP) [7]. Instead of adding strong, cryptographic integrity protection, they propose an access right based system called Reverse Map Table (RMP), that assigns each physical memory page either to the hypervisor, a specific VM or to the SP. Only the owner of a page is given write access. The RMP will be manageable by the hypervisor via an instruction set extension. For SEV-SNP secured VMs, the RMP also contains the supposed GPA of the page inside the VM as well as a ”validated” flag, that is always false for new RMP entries. The ”validated” flag can only be manipulated by the VM that the page is assigned to. This way, AMD intends to prevent remapping attacks like the one in [31], as they would require the VM to validate multiple pages for one GPA. While this mechanism is not able to detect that a cipher text block was manipulated, the lack of write access to a page would prevent us from performing our cipher block moving attack. However, given the required architectural changes, it is not foreseeable when SEV-SNP will be available.

B. Tweak Function

Without the knowledge of the tweak values we could no longer predict the effect of a cipher block move. [26] 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 Section III 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 Subsection III-B how to brute force the tweak values at run time, as long as they stay 32 bit periodic (or similarly low periodicity). While the tweak recovery process takes about 30 minutes 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.

C. Fixing the Page Fault Side Channel

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 the amount of leaked information can be reduced, by restricting the hypervisor’s ability to manipulate bits in the NPTs. This way we could no longer provoke page faults, but only observe page faults that are “naturally” triggered by the VM. This 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 general strategy. But, since we are no longer able to provoke a page fault, allowing us to at least get the Guest Frame Number (GFN) of the stack, we would now have to dump all of the VM’s RAM that has ever been written to.

In order to implement a sync mechanism, that allows us to interrupt the VM at precise points in time, we are now dependent on the availability of intercepted instructions.

We thus conclude that mitigating the page fault side channel 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.

D. Availability of Known Plaintext

For our attack, we used the Linux kernel itself as a source of known plaintext. 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.

If the entire boot image is encrypted, one can use the technique presented in Werner et al. [33]: 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 instructions of an application. While they only evaluated their approach for user space applications, their result is also applicable to the Linux kernel. Another approach is building on a method presented in [19]: They show that the kernel location can be detected at runtime, by removing the execute permissions from all of the VM's memory pages, injecting an interrupt and observing the occurring page faults. Given the result of Werner et al., this could also be used to fingerprint the kernel based on the GPAs of the interrupt handler functions.

E. Emulated Operations

Whether an instruction like `cpuid` or `rdtsc` is intercepted by the hypervisor can be configured in the control area of the VMCB [6]. The VMCB gets encrypted and integrity protected upon a `#VMEXIT`. Furthermore, it is part of the initial attestation [23], so it cannot be manipulated by a malicious hypervisor. The high performance `cpuid`-based encryption oracle from Section V-B can thus be mitigated by disabling interception of the `cpuid` instruction [6]. However, as already stated in Section II-D, 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 during live migration.

In theory, the GHCB mechanism used under SEV-ES allows the VM to inspect the hypervisor supplied results of an emulated operation before it continuous its operation. However, AMD's current SEV-ES kernel does not implement such checks (cf. function `vmg_cpuid`). For operations like `rdtsc`, the effectiveness of filtering is uncertain. In the recently released AMD-SNP whitepaper [7] AMD describes a mechanism, that allows the VM to verify the result of the `cpuid` operation by using the SP as a proxy. This is possible, as the hypervisor cannot interfere with the result of operations executed on the SP.

F. Detection

Another important aspect, besides direct countermeasures, is attack detection. In the scenario of a malicious hypervisor spying on VMs, detecting an attack could lead the guests to switch to another service or pursue legal matters.

Since SEV itself does not provide any integrity protection for the RAM content, this must be done by the guest and in software. This is significantly complicated by the large number of possible injection points, and the fact that the injected code is only temporarily present. In addition, the program that inspects the guests RAM content in order to find changed code, cannot be certain that its own code is unchanged.

Another approach is detecting abnormal behavior, like the unusual kernel base address when disabling KASLR. However, detecting more transient abnormal behavior, like a manipulated random number generator, is quite difficult due to the large attack surface.

IX. Responsible Disclosure

We have informed AMD of our findings. In our discussions they suggested that the recently released Zen 2 architecture uses an improved tweak generation, which is no longer 4-byte periodic and uses fresh randomness per boot, which may significantly complicate the described attacks. However, they did not implement an additional integrity protection, yet.

X. Conclusion

In this work we have shown that the lack of proper integrity protection can be exploited to execute arbitrary code within SEV-ES secured VMs. We have reverse engineered the new, XEX-based encryption on updated AMD Epyc processors, and developed a method to control plaintext bytes by moving existing ciphertext blocks. After using this method for bootstrapping a 2-byte encryption oracle, we have shown how to place instructions to control 4 bytes and finally 16 bytes per plaintext block, yielding a 16-byte encryption oracle. In addition, we have shown how to abuse the emulated `cpuid` instruction to build a high performance encryption oracle. Compared to similar attacks, our attacks works with SEV-ES and does not rely on any I/O operations.

We have discussed various countermeasures: A stronger tweak function and disabling instruction interception might significantly complicate our described attacks. However, we do not expect that a full mitigation is possible without implementing a proper integrity protection, which is able to detect modified ciphertext before decryption.

Proof of concept code is available at https://github.com/UzL-ITS/SEVurity/.

Acknowledgments This research was supported by DFG (Grant 427774779). The authors would like to thank Alina Weber-Hohengrund for assisting with reverse engineering the tweak function and Martin Radev for analyzing the page fault sequences at boot time. Furthermore, the authors would like to thank the anonymous reviewers and especially our shepherd Yinqian Zhang for detailed comments and suggestions for improvement.

REFERENCES

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

[2] Advanced Micro Devices, "Github - AMDESE/linux," https://github.com/AMDESE/linux, on branch "sev-es-5.1-v8" at commit "cef9c6a129da1039683604aa9d8d0fd1cbd8c2aa".

[3] ——, "Github - AMDESE/ovmf," https://github.com/AMDESE/ovmf.git, on branch "sev-es-v10" at commit "f74218f93a98455224636a2efd63056dff24c541".

[4] ——, "Github - AMDESE/qemu," https://github.com/AMDESE/qemu.git, on branch "sev-es-v3" at commit "d61d34c70460503aec92ad3a14d10601972c3e31".

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

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

[7] Advanced Micro Devices, "AMD SEV-SNP: Strenghtening VM Isolation with Integrity Protection and More," White Paper, Jan 2020.

[8] "AMD SEV Resource Center," https://developer.amd.com/sev/, accessed: 2019-11-26.

[9] I. Anati, S. Gueron, S. Johnson, and V. 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*, vol. 13. ACM New York, NY, USA, 2013.

[10] J. Bauer, M. Gruhn, and F. C. Freiling, "Lest we forget: Cold-boot attacks on scrambled ddr3 memory," *Digital Investigation*, vol. 16, pp. S65–S74, 2016.

[11] R. Buhren, S. Gueron, J. Nordholz, J.-P. Seifert, and J. Vetter, "Fault Attacks on Encrypted General Purpose Compute Platforms," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 197–204.

[12] R. Buhren, F. Hetzelt, and N. Pirnay, "On the detectability of control flow using memory access patterns," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. ACM, 2018, pp. 48–53.

[13] R. Buhren, C. Werling, and J.-P. Seifert, "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. ACM, 2019.

[14] V. Costan and S. Devadas, "Intel SGX Explained," https://eprint.iacr.org/2016/086.pdf, 2016.

[15] U. Drepper, "Memory Part 3: Virtual Memory," https://lwn.net/Articles/253361/, 2007, Accessed: 2019-05-11.

[16] Z.-H. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang, "Secure encrypted virtualization is unsecure," *arXiv preprint arXiv:1712.05090*, 2017.

[17] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 54–62, 2016.

[18] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys," in *usenix-security*, 2008.

[19] F. Hetzelt and R. Buhren, "Security Analysis of Encrypted Virtual Machines," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. New York, NY, USA: ACM, 2017, pp. 129–142. [Online]. Available: http://doi.acm.org/10.1145/3050748.3050763

[20] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA*, vol. 11, 2013.

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

[22] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2011, pp. 272–283.

[23] D. Kaplan, "Protecting VM Register State with SEV-ES," White Paper, Feb. 2017.

[24] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," Advanced Micro Devices, Tech. Rep., 2016.

[25] T. 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.

[26] M. Li, Y. Zhang, and Z. Lin, "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[27] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *Hasp@ isca*, vol. 10, no. 1, 2013.

[28] S. Mofrad, F. Zhang, S. Lu, and W. 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*. ACM, 2018, p. 9.

[29] M. Morbitzer and M. Huber, "Github - SEVered Framework," https://github.com/Fraunhofer-AISEC/severed-framework/, 2019.

[30] M. Morbitzer, M. Huber, and J. Horsch, "Extracting Secrets from Encrypted Virtual Machines," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '19. ACM, 2019, pp. 221–230.

[31] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "SEVered: Subverting AMD's Virtual Machine Encryption," in *Proceedings of the 11th European Workshop on Systems Security*, ser. EuroSec'18. New York, NY, USA: ACM, 2018, pp. 1:1–1:6. [Online]. Available: http://doi.acm.org/10.1145/3193111.3193112

[32] P. 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*. Springer, 2004, pp. 16–31.

[33] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. 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*, ser. Asia CCS '19. ACM, 2019, pp. 73–85.

[34] Y. Xia, Y. Liu, and H. 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)*. IEEE, 2013, pp. 246–257.

[35] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 640–656.

[36] S. F. Yitbarek, M. T. Aga, R. Das, and T. 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)*. IEEE, 2017, pp. 313–324.