Exploiting Interfaces of Secure Encrypted Virtual Machines

Martin Radev
martin.radev@aisec.fraunhofer.de
Fraunhofer AISEC
Garching near Munich, Germany

Mathias Morbitzer
mathias.morbitzer@aisec.fraunhofer.de
Fraunhofer AISEC
Garching near Munich, Germany

ABSTRACT

Cloud computing is a convenient model for processing data remotely. However, users must trust their cloud provider with the confidentiality and integrity of the stored and processed data. To increase the protection of virtual machines, AMD introduced SEV, a hardware feature which aims to protect code and data in a virtual machine. This allows to store and process sensitive data in cloud environments without the need to trust the cloud provider or the underlying software.

However, the virtual machine still depends on the hypervisor for performing certain activities, such as the emulation of special CPU instructions, or the emulation of devices. Yet, most code that runs in virtual machines was not written with an attacker model which considers the hypervisor as malicious.

In this work, we introduce a new class of attacks in which a malicious hypervisor manipulates external interfaces of an SEV or SEV-ES virtual machine to make it act against its own interests. We start by showing how we can make use of virtual devices to extract encryption keys and secret data of a virtual machine. We then show how we can reduce the entropy of probabilistic kernel defenses in the virtual machine by carefully manipulating the results of the `CPUID` and `RDTSC` instructions. We continue by showing an approach for secret data exfiltration and code injection based on the forgery of MMIO regions over the VM's address space. Finally, we show another attack which forces decryption of the VM's stack and uses Return Oriented Programming to execute arbitrary code inside the VM.

While our approach is also applicable to traditional virtualization environments, its severity significantly increases with the attacker model of SEV-ES, which aims to protect a virtual machine from a benign but vulnerable hypervisor.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; **Virtualization and security**.

KEYWORDS

Trusted execution environments, AMD SEV, encrypted virtual machines, probabilistic kernel defenses, KASLR, stack canaries, virtio, code execution, ROP, data exfiltration, code injection

1 INTRODUCTION

Hardware virtualization is a common approach in cloud environments to securely and efficiently distribute hardware resources among multiple users. Using virtualization, each user can create its own Virtual Machine (VM) to run any Operating System (OS) and software. Each VM uses virtualized hardware, which is managed by the Hypervisor (HV). The HV has direct access to the VM's memory and architectural state. This means that the HV is always able to infer computations performed within the VM. Therefore, the user of the VM has to fully trust the HV, and hence the cloud provider.

In 2016, AMD released the Secure Encrypted Virtualization (SEV) feature to protect a VM from a compromised HV [17]. Since then, also the successors SEV Encrypted State (SEV-ES) [11] and SEV Secure Nested Paging (SEV-SNP) [4] have been announced, which address additional attack vectors. These improvements provide protection and confidentiality by encrypting the VM's memory and state, and limit the possibilities of targeted memory corruption by the HV. The SEV features use a strong attacker model for which the confidentiality of the VM must be guaranteed even if the attacker is in control of the HV.

However, modern OSs, which are also used inside the VMs, have been developed over the years with the consideration that higher privileged layers are always trusted. In comparison, SEV introduces a new attacker model in which the HV may be compromised and malicious [17]. This new attacker model requires to reevaluate the security constraints and to sanitize the no longer trusted interfaces between the VM and HV. Such interfaces include, among others:

- the `virtio` interfaces for communicating data with external devices
- instructions intercepted by the HV
- Model Specific Registers (MSRs) for virtualization

These interfaces exist to provide correct and configurable virtualization, and to match a traditional attacker model in which the VM is considered untrusted and the HV needs to be protected. However, with the attacker model of SEV, these interfaces have to be considered untrusted and carefully validated from both sides.

As long as this sanitization of external interfaces is not performed, the VM is potentially vulnerable to attacks from higher privilege layers. We show that such attacks can be performed by a HV on a VM by adding cryptographic `virtio` devices to the VM's security components, or by providing semantically incorrect information such as CPU capabilities and time. This allows a malicious HV to force the VM to act against its own interest, for example by 1) exposing cryptographic keys and secret data to the HV, 2) reducing the entropy of its randomly generated values, 3) leaking secrets or reading HV-controlled data for most `MOV` memory accesses, 4) decrypting its stack memory. Even though such attacks can be applied on any virtualized environment, they are of much higher criticality on SEV- and SEV-ES-protected VMs due to the different attacker model, which aims to protect the VM from a vulnerable cloud provider [17, 29].

In summary, we make the following contributions:

- We introduce a technique which allows us to manipulate the `/dev/hwrng` randomness source in VMs and to extract encryption keys from the Kernel Crypto API.

- We show how we are able to practically disable probabilistic kernel defenses in encrypted VMs by intercepting `CPUID` and `RDTSC` instructions.
- We demonstrate how a malicious HV can forge Memory-mapped I/O (MMIO) regions to exfiltrate and inject data into encrypted VMs.
- We present a fourth attack which allows us to execute code in encrypted VMs by corrupting the VM's Guest Page Table.

Note that an early report, results and Proof-of-Concepts (PoCs) were responsibly delivered to the AMD security team with which we discussed possible mitigations against the attacks.

2 BACKGROUND

In this section, we give a quick introduction into hardware virtualization and AMD SEV protection mechanisms.

2.1 Hardware virtualization

Hardware virtualization allows to create virtual hardware configurations by using both software and hardware. This technology enables the creation of multiple VMs running on the same physical host. Each VM runs its own OS and can have its own hardware configuration. The VMs are managed by the HV, a combination of user-space and kernel-space code which performs the necessary operations to support the correct execution of the VM. Such operations include the allocation of physical memory for the VM, providing CPU capability information or emulating special instructions.

In order to guarantee the security of the HV and other VMs on the same system, VMs do not have complete access to the underlying hardware. For security and correct emulation, the HV emulates certain CPU instructions such as `IO instructions`, `MSR instructions`, `CPUID` and `RDTSC`. The `CPUID` instruction returns information about CPU features. If a VM issues the `CPUID` instruction, the HV is able to adapt the return values of the instruction. This allows the HV to disable certain CPU features for the VM or to offer features which can be emulated.

The `RDTSC` instruction reads the CPU core's Time Stamp Counter (TSC), a register which counts the number of cycles since the last reset [5]. Each CPU core has its own TSC, which is independent from the TSCs on other cores. To ensure that the different values of the TSCs do not cause any problems when moving a VM to a different core, the HV is able to control the TSC by means of interception or writing to relevant MSRs.

2.2 AMD SEV

In 2016, AMD introduced the SEV feature [17] to protect a VM from a benign but vulnerable HV. SEV achieves this goal by encrypting the VM's memory with an encryption key bound to the instance of the VM. This prevents the HV from accessing the VM's memory in plaintext.

Not all of the VM's memory is encrypted because the VM and the HV need to exchange data such as network packets or disk drive blocks. To facilitate this, the VM can select which physical pages should be encrypted or unencrypted by setting or unsetting the `C-bit` in the Guest Page Table (GPT) [17]. During the boot stage of the VM, the kernel marks almost all memory as encrypted and later marks memory regions used for data communication as unencrypted.

An important drawback of SEV is that it does provide confidentiality to the VM's memory, but it does not guarantee its integrity or freshness. Thus, SEV is vulnerable to attacks which attempt to corrupt [12, 18, 32] or replay the VM's memory as well as memory remapping attacks [16, 21]. Another problem with SEV is that it allows the HV to read and modify the VM's architectural registers during a `VMEXIT`. These registers may contain secret information such as encryption keys and passwords [31], and should therefore not be exposed to the untrusted HV.

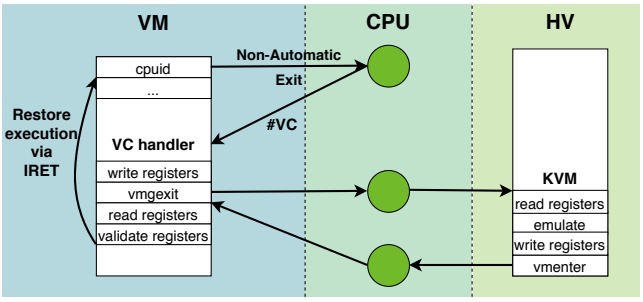


Figure 1: On a Non-Automatic exit, the VM first passes control to its VC handler. Only afterwards, the VC handler gives the control to the HV. After resuming the VM, the VC handler validates the information from the HV before returning to the original instruction.

To reduce the risk of exposing architectural registers to the HV, AMD announced SEV-ES in 2017 [11]. SEV-ES encrypts and integrity protects the architectural state of the VM on a `VMEXIT`. However, the HV still needs to be able to read and modify some of the VM's registers in order to emulate intercepted instructions such as `CPUID` or `RDTSC`. SEV-ES addresses this issue by introducing a new unencrypted region, called the Guest Hypervisor Communication Block (GHCB). The GHCB contains only register values selected by the VM, and is accessible by both the HV and the VM. This method allows for communication of register values.

Figure 1 shows a sequence diagram for executing an intercepted instruction. When the HV intercepts an instruction, the CPU performs a Non-Automatic Exit (NAE). This causes a VMM Communication Exception (`#VC`), which gets handled by the VM's VC handler. The VC handler determines the cause of the exception and decides whether to invoke the HV to emulate the instruction. If the HV needs to be invoked, the VM copies the necessary registers into the GHCB and executes the `VMGEXIT` instruction to invoke the HV.

In case of intercepting for example the `CPUID` instruction, the VM stores the values of the `EAX` and `ECX` registers into the GHCB and executes a `VMGEXIT`. When the HV receives the `VMGEXIT`, it first reads the `EAX` and `ECX` registers. Afterwards, the HV emulates the `CPUID` instruction and writes the new values of the `EAX`, `ECX`, `EBX`, `EDX` registers into the GHCB. To finalize, the HV hands control back to the VM by issuing a `VMENTER`. Once the VM continues execution, it can validate the registers passed by the HV and decide whether to update the architectural state. Then, the VM issues a Return from

interrupt (IRET) instruction to continue execution immediately after the instruction which was originally intercepted.

In 2020, AMD announced an update to SEV-ES named SEV-SNP [4]. SEV-SNP further enhances protection by addressing attacks based on memory corruption and page remapping. SEV-SNP also introduces a `CPUID` reporting feature which allows the VM to verify the available CPU capability features with the firmware [8]. However, this feature does not prevent the HV from disabling certain CPU capabilities for the VM. As SEV-SNP was only recently standardized, we are not aware of any AMD CPU that supports SEV-SNP at the time of writing.

3 UNSAFE VIRTIO DEVICES

In most virtualized environments, the Linux kernel makes use of the `virtio` interface [14] to efficiently communicate data to external devices such as network cards and disk drives. To establish communication via `virtio`, the VM queries the identifiers and configurations of the available virtual devices from the HV. Afterwards, the VM's kernel automatically loads the corresponding device drivers. For example, when the VM attempts to write to disk or to send a network packet, the corresponding `virtio` driver in the VM would consume the request. Afterwards, it writes the data to a suitable guest physical address and signals the HV to propagate the request to the physical hardware device. The added layers of abstraction benefits performance because the `virtio` communication layers are aware of the virtualized environment and can transfer data more efficiently by reducing the number of VM exits.

The improved performance has led to additional `IO` devices making use of `virtio`, including devices essential for the VM's security. In this section, we show how two of those devices — `virtio-rng` [25] and `virtio-crypto` [24] — pose a security risk when considering SEV's attacker model.

The `virtio-rng` device is a Random Number Generator (RNG) device, which allows to add entropy to the kernel's entropy pool [25]. After initialization, the device is accessible within the VM via the `/dev/hwrng` interface. By default, SEV-protected VMs do initialize the `virtio-rng` device [6]. However, considering SEV's attacker model, a device controlled by an untrusted HV providing entropy to the VM represents a critical attack vector. For example, entropy-reliant software in the VM might utilize the `/dev/hwrng` interface when seeding an RNG for cryptographic operations. This would allow the HV to fully control the entropy provided to the software in the VM.

Another device which represents a critical attack vector when considering SEV's attacker model is `virtio-crypto` [24]. This device allows a VM to make use of hardware features to accelerate cryptographic transformations. Using the device, the VM utilizes the `virtio` interface to deliver the cryptographic keys and data to the HV. After processing the information in hardware or software, the HV returns the information to the `virtio-crypto` driver in the VM. This process allows the HV to extract cryptographic information processed by the VM.

In order for the VM to use the `virtio-crypto` engine, a malicious HV needs to (*i*) avoid that the VM uses the default `aesni_intel` engine and (*ii*) ensure that the `virtio-crypto` device is registered for the VM. By announcing the `AES-NI` CPU extension as unavailable when the VM issues a `CPUID` instruction, we are able to achieve the former. As the HV is also in charge of launching the VM, we are able to achieve the latter by registering the `virtio-crypto` device when launching the VM. This approach allows us to trick the VM into using the `virtio-crypto` engine, allowing us to extract secret cryptographic information passed to the VM's Kernel Crypto API.

4 CONTROLLING THE ENTROPY SOURCES

In this section, we show how the Linux kernel uses different sources of entropy for its probabilistic defenses. Afterwards, we demonstrate how a malicious HV is able to reduce these sources of entropy in order to have the VM use deterministic values for its presumably probabilistic defenses.

4.1 Probabilistic kernel defenses

To protect itself against attacks, the Linux kernel makes use of various mechanisms such as Kernel Address Space Layout Randomization (KASLR) and stack canaries [30].

KASLR complicates exploitation of vulnerabilities such as buffer overflows in the kernel. It achieves this by randomizing the virtual and physical offsets of the kernel image at boot stage. Due to the randomized offsets, it is difficult to utilize techniques such as Return-oriented programming (ROP) [28]. The randomization offered by KASLR has at most nine bits of entropy for the virtual offset and the limit for the physical offset is determined by the size of available physical memory.

KASLR also makes it difficult to directly read or write any physical memory address and to exploit the kernel's heap by also randomizing other memory regions such as the direct-physical map (`page_offset_base`). The *direct-physical map* determines the virtual address of the whole physical memory mapped into the kernel's address space. Additional randomizations include the *vmalloc* area (`vmalloc_base`), which determines the address of the kernel's *vmalloc heap*, and the *virtual-memory map* (`vmemmap_base`), which determines the base address of the data structure holding the meta information for all physical pages.

Another probabilistic defense are *stack canaries*, which help to detect stack buffer overflows. A stack canary is a random token that is placed at the top of the stack frame between variables and saved special-purpose registers such as the `RIP` and `RBP`. A modified stack canary indicates that special purpose registers could have been overwritten. To verify the stack canary, the compiler adds code to the function's epilogue to check if the stack canary is corrupted. Overwriting the stack canary with the same value is difficult for an attacker, as the canary offers up to 56 bits of entropy. Although guessing the randomized values for KASLR and stack canaries is possible, a single wrong guess would be detected and require a system reboot.

4.2 Early kernel entropy generation

Common probabilistic kernel defenses (Section 4.1) rely on randomizing different parameters to make them unpredictable for an attacker. To ensure that the values are unpredictable, the kernel requires a reliable source of entropy. The two most common sources

of entropy during the early boot stage under the x86 architecture are RDRAND and RDTSC, which are also used by the Linux kernel.

```
1  unsigned long kaslr_get_random_long(...) {
2      unsigned long raw, random = get_boot_seed();
3      ...
4      if (has_cpuflag(X86_FEATURE_RDRAND))
5          if (rdrand_long(&raw))
6              random ^= raw;
7      if (has_cpuflag(X86_FEATURE_TSC))
8          random ^= rdtsc();
9      ...
10     return random;
11 }
```

Figure 2: The `kaslr_get_random_long` function used to calculate KASLR offsets. While Line 2 creates an initial random state, Lines 4 and 7 make use of RDRAND and RDTSC to improve random number generation.

During the boot process, the early Linux boot code has to compute the KASLR physical and virtual offsets before decompressing the kernel image. Both offsets are computed by the function `kaslr_get_random_long` shown in Figure 2. In Line 2, the initial random value is determined by the `get_boot_seed` function which returns a hash of the kernel's build string and the boot parameters structure. The boot parameters structure is allocated, zeroed-out and populated by the OS loader earlier in the boot stage as dictated by the Linux boot protocol [1]. Starting with Line 4, if supported by the CPU, the function makes use of RDRAND to add entropy to the random value. Additionally, starting with Line 7, if supported by the CPU, RDTSC is also used to add entropy to the random value.

```
1  void __init kernel_randomize_memory(void) {
2    ...
3    prandom_seed_state(&rand_state,
4      kaslr_get_random_long());
5    ...
6    for (i=0; i < ARRAY_SIZE(kaslr_regions);i++) {
7      prandom_bytes_state(&rand_state, &rand, 8U);
8      entropy = (rand % (entropy + 1)) & P4D_MASK;
9      vaddr += entropy;
10     *kaslr_regions[i].base = vaddr;
11   }
12 }
```

Figure 3: The `kernel_randomize_memory` function used for randomizing the memory regions. To initialize the Pseudo Random Number Generator in Line 3, the function makes use of the `kaslr_get_random_long` function.

Afterwards in the initialization phase, the kernel computes random offsets for the memory regions `vmalloc_base`, `vmemmap_base` and `page_offset_base`. Figure 3 shows the respective function `kernel_randomize_memory`. In Line 3, the function initializes the Pseudo Random Number Generator (PRNG) state with the value returned from the function `kaslr_get_random_long` we just saw.

Later in the kernel initialization stage, instantiated device drivers can add entropy to the Linux entropy pool by calling the function `add_device_randomness`. Figure 4 shows the relevant code of this

```
1 void add_device_randomness(const void *buf,
2         unsigned int size) {
3     u64 time = random_get_entropy() ^ jiffies;
4     ...
5     _mix_pool_bytes(&input_pool, &time, 8U);
6     ...
7 }
```

Figure 4: The `add_device_randomness` function used for adding entropy to the Linux entropy pool. The value of `random_get_entropy()` in Line 3 corresponds to the value of RDTSC, and the value of the `jiffies` variable remains constant in the early kernel initialization phase.

function. In Line 3, the return value of `random_get_entropy` equals the return value of RDTSC, and the number of jiffies is constant at this stage since no timer interrupts have been yet received. Afterwards in Line 5, the function mixes the entropy into the entropy pool. During the Kernel's lifespan, entropy is also added to the entropy pool by other events such as interrupts, user input events, and disk usage [22]. The corresponding functions also use RDTSC as part of the random data mixed into the entropy pool.

Similarly to KASLR, stack canaries of kernel threads and userspace processes receive random bytes from the Linux RNG, which in turn receives its entropy from the entropy pool. Therefore, as long as the entropy injected into the pool is known to the HV, the entropy of any stack canary is significantly reduced.

4.3 Entropy manipulation

We have now seen that many of the probabilistic kernel defenses rely on RDRAND and RDTSC as sources of entropy. Thus, these defenses are only as good as the results returned from these instructions. Next, we show how a malicious HV is able to manipulate the return values of these instructions to eliminate the dependent probabilistic kernel defenses.

The first behavior we are using to our advantage is the fact that the HV controls the CPU capabilities offered to the VM. Having control over the HV, we can inform the VM that RDRAND is unsupported by the CPU. This prevents the VM from using its main source of entropy during the early boot and kernel initialization stage.

Instead, the VM and therefore all its probabilistic kernel defenses now only rely on RDTSC as a single source of entropy during the boot and kernel initialization stages. Additionally, we are able to intercept the RDTSC instruction, which allows us to arbitrarily adjust the return value of RDTSC. This results in full control over the VM's only source of entropy during the early stage of the kernel. Therefore, we are able to manipulate the probabilistic kernel defenses of the VM and to make them deterministic.

The only aspect we have to consider when intercepting and emulating RDTSC is that the VM also uses RDTSC for adding cycle delays and calibrating timer interrupts. Therefore, improper emulation would resolve in hanging kernel threads. Thus, we have to differ between RDTSC being executed for entropy and for a different usage.

With the first version of SEV, it is trivial to identify why the VM is calling RDTSC since we are able to read the RIP register of each of the VM's vCPUs. By statically inspecting the VM's kernel image,

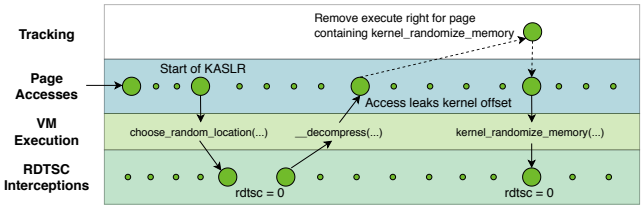


Figure 5: Monitoring the VM page access not only allows us to set the return value of the RDTSC instruction to zero at the required point in time, but also allows us to determine the KASLR physical offset.

we are able to infer the location of the kernel and to select which RDTSC executions to manipulate.

Using SEV-ES or SEV-SNP, the VM's registers are encrypted, which prevents us from determining the VM's RIP. However, we made the observation that different RDTSC instructions are typically used in functions which are located on separate pages in the VM's physical memory. Furthermore, the function calls which precede the one using RDTSC are also typically located on different pages. This allows us to identify the RDTSC usage by using the sequence of recent page accesses as context for the prediction.

We determine the most recent page accesses by manipulating the Second Level Address Translation (SLAT) table of the VM. Such manipulation cannot be performed solely with the Linux virtualization solution KVM, as it does not provide sufficient control over the SLAT table. Instead, we make use of the *SEVered framework* [19] to monitor page accesses. The SEVered framework extends KVM and allows to remove the `present` flag from the VM's physical pages. As soon as the VM attempts to access any data on such a page, the resulting page fault causes the VM to trap into the HV. This allows the HV to track the different memory accesses of an SEV(-ES)-protected VM at page granularity.

We make use of this possibility to manipulate the entropy of a VM at boot. For this, we utilize the fact that the sequence of page faults before decompressing the kernel is deterministic, as already briefly mentioned by Wilke et al. [32]. This behavior allows us to easily identify the page access after which the VM twice executes RDTSC. While the VM uses the first execution to determine the physical KASLR offset, the second execution determines the virtual offset. By intercepting both executions and specifying a fixed return value, we are able to pin the physical as well as the virtual KASLR offset.

The next page fault indicates the location in the VM's physical memory to which the VM decompresses the kernel image. This allows us to calculate the locations of all VM kernel functions in physical memory by statically inspecting the VM's kernel image.

By analyzing the VM's page accesses at boot time, we found suitable page fault sequences which uniquely identify the usages of RDTSC as a source of entropy. We modified KVM to include a state machine to easily allow us to specify what actions to perform when a page fault occurs or an RDTSC is intercepted. This allowed us to always detect when the VM uses RDTSC as a source of entropy in order to pass a known fixed value to the VM.

Figure 5 shows the sequence of the steps we perform to manipulate the virtual and physical KASLR offset. The tracking lane shows which pages we track, and the page accesses lane shows the physical pages accessed by the VM. The VM execution lane shows the execution flow in the VM, and the RDTSC interceptions lane shows which return values we set to zero. The HV tracks the naturally occurring page faults until the VM accesses the page containing the function responsible for selecting random locations for the kernel. We then set the return values of the following two RDTSC executions to zero to pin the KASLR physical and virtual image offsets. The following page access is caused by the early Linux boot code decompressing the kernel image into memory, and discloses the KASLR physical offset. This allows us to determine the location of the `kernel_randomize_memory` function within the VM's memory. We use this knowledge to track the respective page by removing its `execute` permission. Once the VM accesses the page, we set the return value of the subsequent RDTSC execution to zero. While only requiring a single interference with the SLAT table, these steps allow us to pin the KASLR code and memory regions offsets between reboots of the VM.

4.4 Stealthy manipulation of the TSC

Additional difficulties for an attacker arise under SEV-ES and SEV-SNP when attempting to intercept the RDTSC instruction. Intercepting the instruction is required in order to manipulate its return value (Section 4.3). However, when using SEV-ES or SEV-SNP, such an interception can be detected by the VM as it causes a VC exception. The VM captures the exception and then performs a VMGEXIT to request the HV to emulate the instruction. After the HV finished the emulation, the VM can sanitize the returned values.

This approach allows the VM to detect an unexpected RDTSC interception and manipulation of the return value by the HV. A VM aware of this attack vector might forbid interception of RDTSC and simply halt execution.

However, the AMD Open-Source Register Reference [3] also discusses other possibilities for the HV to manipulate the VM's TSC. These exist to ensure correct execution of the VM and smoothless relocation to a different core or to other CPUs where the TSC might have a different value. The additional possibilities include the TSC `Ratio MSR` and the TSC `Offset` field in the Virtual Machine Control Block structure. The TSC `Ratio MSR` allows the HV to provide a fixed-point multiplier for the elapsed time. Additionally, the TSC `Offset` field is an offset added to the value of the TSC when the VM reads it. Setting the value of the TSC `Ratio MSR` and the value of the TSC `Offset` to zero will effectively zero-out the TSC when read by the VM. Therefore, using the TSC `Ratio MSR` and the TSC `Offset` allows us to specify any value for the TSC without intercepting the RDTSC instruction. For the rest of this paper, we assume the TSC `Offset` to be set to zero.

This approach requires careful calibration of the attack. Unlike RDTSC interception, manipulation of the TSC `Ratio MSR` or TSC `Offset` does not allow us to determine how many times the VM calls RDTSC. Instead, we need to select a period of time during which the TSC will be zero. This period must not include calls of the VM to RDTSC for which the return value zero could cause problems for the VM. Examples for such calls are when RDTSC is used for adding

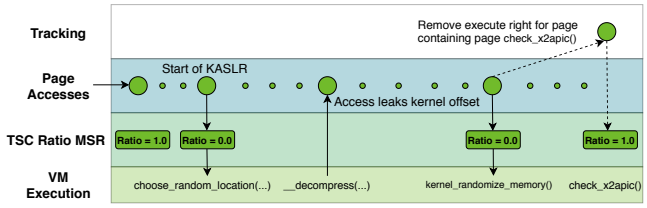


Figure 6: Diagram for TSC Ratio MSR manipulation. The `TSC Ratio MSR` is kept at 0.0 starting at the call to `choose_random_location` until after the call to `kernel_randomize_memory`.

delay or calibrating timer interrupts. Also, we need to consider that the `TSC Ratio MSR` is a register available on each core. If the VM is rescheduled to execute on another core, we must ensure that the `TSC Ratio MSR` value on the respective cores are updated.

Figure 6 shows how we use the `TSC Ratio MSR` to manipulate the VM's early entropy sources. The tracking, page accesses and VM execution lanes serve the same function as before (Section 4.3), and the `TSC Ratio MSR` lane shows the current value of the register. Once the VM starts, the `TSC Ratio MSR` has the default value of `1.0`. When the VM first accesses the page containing `choose_random_location`, we set the value of the `TSC Ratio MSR` to `0.0`. This returns the value `0*<TSC-value>` to the VM's `RDTSC` instruction. Subsequently, the VM will start to decompress its kernel image. Therefore, the following page access will leak the VM's KASLR physical offset which discloses the physical locations of all kernel functions of the VM. At this point, we further keep the `TSC Ratio MSR` value at `0.0` until `kernel_randomize_memory` returns. By statically examining the Linux image, we determined that the kernel afterwards calls the function `check_x2apic`, which is located on a different page. This allows us to use the respective page as a trigger. By removing the execute permission of the page, we are able to determine when the kernel calls the `check_x2apic` function. Once the VM attempts to fetch an instruction from this page, we set the `TSC Ratio MSR` back to its default value and resume the VM.

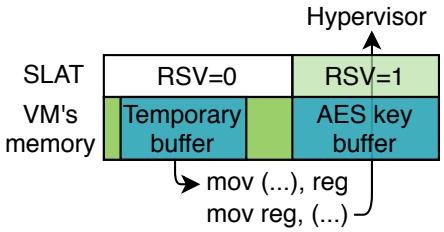
Using the provided steps, we are able to pin the KASLR code and memory regions offsets between reboots of the VM. This approach additionally complicates detection from within the VM, as this attack does not require interception of the `RDTSC` instruction.

5 DATA EXFILTRATION AND INJECTION VIA MMIO REGION FORGERY

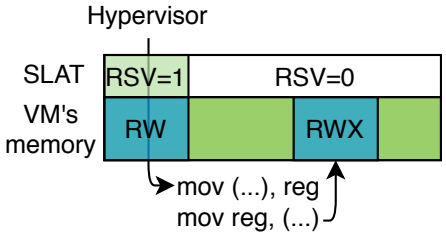
In the previous section, we showed how we are able to simplify locating secrets within an encrypted VM by manipulating its sources of entropy. In this section, we show how located secrets can afterwards be exfiltrated, or malicious data be injected, by forging MMIO regions in the VM's address space under SEV-ES.

A VM's software can program devices either by using I/O instructions, or by performing regular reads and writes to a specific MMIO region. While I/O instructions are always intercepted under SEV-ES, regular reads and writes to the MMIO region are not. Not intercepting regular reads and writes to MMIO regions raises the issue that the HV would not be able to infer which MMIO operations should be performed. To work around this limitation, SEV-ES uses the *MMIO/NPF* sequence [7], in which the HV sets a `Reserved` bit in the SLAT table entries which correspond to the MMIO region. When the VM accesses a page with a set `Reserved` bit, the CPU throws an `MMIO/NPF` exception, which invokes the VM's VC handler. The VC handler determines the type of access and the involved data values. Afterwards, it writes the Guest Physical Address (GPA) and the length of the memory access to the GHCB and executes a `VMGEXIT` in order to expose the information to the HV. Afterwards, if the VM performed a read access from an MMIO region, the HV writes the value into the GHCB *scratch buffer*, and the VM's VC handler loads the value into the corresponding register. In comparison, when writing to an MMIO region, the VM copies the value to the GHCB scratch buffer, from where the HV reads it.

At the time of writing, SEV-ES is not officially supported in Linux, but the patches have been made public in mid-2019 and are in-review since early 2020. In all iterations of the SEV-ES Linux patch set, the VC handler does not verify the address and properties of the page with the set `Reserved` bit. This allows a malicious HV to set the bit on any of the VM's pages, tricking the VM to leak secret data or to read HV-controlled values into registers. Additionally, it allows the HV to infer control-flow information as each memory access on a page with set `Reserved` bit traps into the HV. This gives a malicious HV fine-grained control on data exfiltration and injection by setting and removing the `Reserved` bit for the VM's memory pages.



(a) Data exfiltration from the VM: First, the HV sets the `Reserved` bit on a page containing the *AES key buffer*. Afterwards, writes to the page are exposed by the VM's VC handler, leaking the copied AES key.



(b) Code injection into the VM: First, the HV sets the `Reserved` bit on a page containing a Read-Write buffer with code. This allows the HV to intercept the read and to manipulate the information written into the executable buffer.

When the Linux kernel derives an AES key, it first copies the key into a temporary buffer. This temporary buffer is then provided to the cipher implementation via the Kernel Crypto API, and the cipher implementation copies the buffer's content into an internal buffer. Figure 7a shows an example for such a copy process, and how this allows a malicious HV to extract data from the VM. The HV, aware of the buffer's location within the VM's address space, can set the `Reserved` bit on the page containing the *AES key buffer*. This would cause the write into the *AES Key buffer* to throw an `MMIO/NPF` exception, causing the VM's VC handler to expose the data to be written to the HV.

Further, the HV can make use of a similar approach to gain code execution with an SEV-ES-protected VM. For the attack, the HV determines a process within the VM in which software copies data from a Read-Write buffer to another buffer, which is then marked as executable. This process occurs naturally when the VM loads a program for execution or when it compiles code *just-in-time*. Figure 7b shows an example for such a process. Before the copy process, the HV sets the `Reserved` bit for the page containing the Read-Write buffer. This allows the HV to intercept the VM's read requests from the Read-Write buffer, and to provide malicious code and data. Afterwards, the VM copies the malicious code and data to the executable buffer, where the code is eventually executed by the assigned process within the VM.

While we showed the vulnerability using the example of the VC handler of the Linux kernel, we additionally verified that the vulnerability also exists in the Open Virtual Machine Firmware (OVMF) code base. This makes the VM also vulnerable to our attack at the very beginning of the boot sequence. At this early stage, the VM's address space is not yet randomized, and most memory is marked as Read-Write-Executable. These conditions allow the HV to further simplify the attack and to gain code execution inside the VM before the early Linux boot code has even begun execution.

6 CODE EXECUTION VIA GUEST PAGE TABLE CORRUPTION

In this section, we show how another approach to execute arbitrary code within an SEV or SEV-ES-protected VM. To achieve this goal, we provide incorrect `CPUID` information when the VM sets up its page tables at boot, allowing us to modify the VM's unencrypted stack.

```
1  SYM_FUNC_START(get_sev_encryption_bit)
2      ...
3      movl $0x8000001f, eax
4      cpuid
5      bt $1, eax /* Check if SEV is available */
6      jnc .Lno_sev
7      ...
8      movl ebx, eax
9      andl $0x3f, eax /* Get C-bit location */
10     jmp .Lsev_exit
```

Figure 8: Code in Linux for checking the SEV state. On Lines 3-6, the VM checks if the CPU supports SEV by executing `CPUID`. On Lines 8-9, the VM retrieves the *C-bit* position from the returned result.

Figure 8 shows the code in the early Linux boot code which checks whether SEV is used and retrieves the location of the `C-bit` (Section 2.2) in a Guest Page Table (GPT) entry. In Lines 3 – 6, the VM executes `CPUID` and checks whether SEV is available by inspecting the first bit of the `EAX` register. If SEV is not available, the VM returns. Otherwise, the VM extracts the position of the `C-bit` on Lines 8-9. Knowing the position of the `C-bit`, the VM creates a bitmask which it applies to every entry in the GPT. Afterwards, the VM loads the GPT by writing its address to the `CR3` register.

By intercepting the `CPUID` instruction in Line 4, a malicious HV can report SEV as unavailable. The VM would then mark all of its memory, including already encrypted pages, as unencrypted, causing a corruption of the GPT. Although this corruption would mean that previously encrypted pages are not decrypted when read, the VM would not crash immediately as instruction fetches and page table walks always interpret memory as encrypted [5]. Thus, the VM will continue execution until other accesses which do not decrypt the memory, such as data fetches, would cause a fault. This gives a malicious HV sufficient time to overwrite the VM's stack, allowing to gain code execution inside the VM via Return-oriented programming (ROP).

In more detail, we target the function which setups and sets a new page table, `initialize_identity_map`. The function initializes an identity mapping of memory for which a virtual address equals the physical address. Afterwards, it sets the new GPT by writing to the `CR3`, and returns. This return will be followed by an immediate crash in case of a corrupted GPT. The reason for the crash is that the saved `RIP` on the stack was encrypted, but is now accessed without decryption due to the corrupted GPT. This will cause an invalid instruction fetch, causing the VM to crash.

However, a malicious HV can avoid the crash by overwriting the VM's stack with suitable addresses before the return gets executed. As both the HV and the VM access this memory without the `C-bit`, the VM would see the same value as the HV. This enables the HV to overwrite the VM's stack with return addresses, allowing to perform ROP.

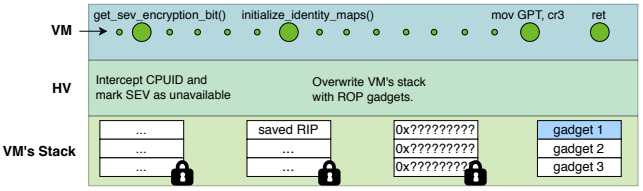


Figure 9: Diagram for performing the GPT-corruption attack. First, the HV provides invalid SEV information and waits until the VM enters the function `initialize_identity_maps()`. Next, the HV overwrites the VM's stack with a ROP-chain which gets executed after the VM sets the corrupted GPT and returns.

Figure 9 shows the detailed steps for performing the attack. The top two lanes show the operations performed by the VM and by the HV, while the bottom lane shows the state of the VM's stack. We again make use of page access tracking to track the VM's actions. When the VM enters the function `get_sev_encryption_bit()`

and executes the CPUID instruction, the HV intercepts the execution and reports that SEV is not available. Afterwards, the VM calls the function initialize_identity_maps() by which it saves the RIP on the stack. At this point, the HV overwrites the VM's stack with the addresses of suitable ROP gadgets in plaintext. If the VM would now access the data on the stack, it would attempt to decrypt the data written by the HV, which would likely result in a crash. However, as part of creating the identity mapping, the VM next sets the new, corrupted GPT by writing to the CR3 register. Due to the corrupted GPT, the VM will no longer attempt to decrypt data on the stack, causing it to read the data exactly as previously written by the HV. Therefore, after the VM executes the return instruction, it executes the ROP-chain injected by the HV. Using these steps, a malicious HV is able to execute code inside an SEV- or SEV-ES-protected VM.

```
1 mov rcx, cr3 /* Change GPT */
2 lea 0x49401385(r8), rsp /* Control RSP */
3 add 0xd0248f72, eax
4 push rax /* Control RIP */
5 ret
```

Figure 10: ROP gadget which sets the OVMF GPT, but also provides control over RSP and RIP.

Despite being now able to execute code via ROP, an attacker cannot directly steal any encrypted secrets from within the VM as all of memory is marked as unencrypted. To steal secrets or inject machine code, we need to map the corresponding pages as encrypted. However, the VM cannot modify its page table because SEV requires the page table to be written encrypted [5]. To still be able to continue exploitation, we make use of the page table created by OVMF. The OVMF's GPT contains two memory regions: an encrypted one for code and data, and an unencrypted one used for communication with the HV. However, simply performing the transition would crash the VM as the ROP-chain is written unencrypted, but the stack memory is located in the encrypted region of the OVMF's GPT. Therefore, after switching to the OVMF's GPT, the ROP-chain would be interpreted as encrypted memory and hold invalid addresses. Thus, we require a gadget which updates CR3 while allowing us to execute additional instructions to update the register state and avoid crashing the VM. To find such a gadget, we again make use of the bzImage, which is sufficiently large to contain a great selection of gadgets. Figure 10 depicts an example for a gadget that we can use to continue exploitation. With this gadget, we can first change to the OVMF's GPT in Line 1. In Line 2, we position the stack into an unencrypted region by setting the RSP. Line 4 allows us to push an arbitrary value onto the stack, which will then be used as RIP, allowing us to jump into another sequence of instructions. At this point, we have both encrypted and unencrypted memory at our disposal, allowing us to inject instructions into the unencrypted region in order to execute arbitrary code as well as to read any secrets from the encrypted region.

7 EVALUATION

In this section, we show the feasibility of our attack vectors by evaluating them in an SEV- and SEV-ES-protected environment. We performed our evaluation on a system equipped with an AMD EPYC 3251 8-Core processor with 64GB RAM running Debian 11. The host kernel was Linux 5.6.0 with the most recent patches for SEV-ES support. Additionally, we added the RDTSC and TSC Ratio MSR manipulation patches, the MMIO-region-forging patch, the GPT-corruption patch, and the patch from the SEVered framework [19].

We made use of KVM and QEMU in version 4.2.50 and two different VMs. The first VM was running Ubuntu 18.04 with kernel version 4.15.0-88 and SEV enabled, but without support for SEV-ES. The second VM was running a custom built Linux kernel with version 5.8.0-rc4 and has the latest SEV-ES patches applied. Notably, we launched the SEV-ES VM with the bzImage directly provided to QEMU via the -kernel parameter.

We verified the attack vector based on untrusted virtio devices (Section 3) on the SEV-protected VM. We first tested that the HV is able to provide arbitrary data to the /dev/hwrng device in the VM provided by virtio-rng. Further, we assured that the HV is able to extract cryptographic keys and plaintext data by using the virtio-crypto device. To ensure that the VM used the device, we modified KVM to advertise the AES-NI extension as unavailable. Lastly, we modified the default SEV launch script to launch the VM with the virtio-crypto device. In order to extract the information communicated over the virtio-crypto, we modified the implementation in QEMU to print the encryption key and the data. The selected test cases included ip-xfrm, kcapi-enc, and a custom kernel module in the VM which used the Linux Crypto API to encrypt a secret message. We used the two programs and the kernel module to encrypt secret data with AES-CBC. In all three tests, we were able to extract the encryption keys and secret data from the VM in plain text.

To evaluate our RDTSC manipulation technique through interception (Section 4.3) and TSC Ratio MSR (Section 4.4), we also used the SEV-protected VM. As part of the verification, we used a kernel module in the VM to print information about the kernel's probabilistic defenses. The module printed the KASLR virtual and physical offset, the offsets for the kernel memory regions and the kernel stack canaries of the first 15 running processes. It is important to note that we used the module solely for evaluation purposes, and did not communicate with the module in any way during the execution of the attack itself. To avoid maintenance of the TSC Ratio MSR among all cores, we pinned the QEMU process to a single core using the taskset command. Additionally, we modified KVM to always zero the TSC Offset in the Virtual Machine Control Block. For both the RDTSC and the TSC Ratio MSR manipulation, we performed 1000 evaluation rounds. In each round, we provided a value of zero for RDTSC whenever it was used as a source of entropy either through interception (Section 4.3) or by modifying the TSC Ratio MSR (Section 4.4). After having performed the attack, we connected to the VM via SSH to collect the information about the offsets. At the end of each round, we rebooted the VM to start the next round of the attack. With both approaches, we were able to pin the value of the stack canaries of the first eleven processes for 1999 out of the in total 2000 runs. The values in the one failing run and of further stack canaries have been likely influenced by interrupts and other events to the VM, which caused entropy to be added to the PRNG entropy pool [22]. Additionally, we were able to successfully pin all KASLR offsets in 999 runs, and only failed when

the VM is initially booted. The failure to pin the offsets on the first boot is due to a bug in GRUB which does not correctly initialize the boot parameters structure when SEV is used.

To prove that the failure during the initial boot is caused by incorrect initialization of the boot parameters structure, we additionally evaluated our attack on the SEV-ES-protected VM which does not use GRUB. For this, we modified the kernel image to directly print the KASLR offsets once kernel initialization has finished. We used the printed information to verify that we successfully pinned all offsets. For both the `RDTSC` and the `TSC Ratio MSR` manipulation, we performed 20 evaluation rounds. At the end of each round, we killed the QEMU process and launched a new VM instance using the same launch arguments. With both approaches, we were able to successfully pin all KASLR offsets in all 20 runs for both SEV and SEV-ES. These results confirm that our attack can also be applied at the initial boot of a VM.

We then validated a PoC based on forging an MMIO region (Section 5) over the encrypted memory of an SEV-ES-protected VM. The PoC featured a user space process which first allocated a Read-Write (RW) page and a Read-Write-Execute (RWX) page. Next, the process wrote a sequence of `NOP` instructions followed by the `RET` instruction to the RW page. Afterwards, it copied the contents of the RW page to the the RWX page and called into the RWX page. To simplify the evaluation, the process reported the GPA of the RW page by issuing a `vmmcall` instruction, after which the HV set the `Reserved` bit on the RW page. On the HV, we were able to intercept each read from the RW page and to provide our own payload to be written to the RWX page. Additionally, we verified with a similar PoC exploit that the HV can view writes to pages with set `Reserved` bit.

We also evaluated the GPT corruption attack (Section 6) on the SEV-ES-protected VM. During the evaluation, we overwrote the VM's stack with a ROP-chain which executed a sequence of gadgets to write a fixed string to a fixed location in the VM's memory. This string could be read by the HV due to the performed GPT corruption. 0.5 seconds after overwriting the VM's stack, we read the VM's memory supposed to contain the fixed string from the HV to determine if the attack was successful. Using this method, we performed 1000 evaluation rounds of the attack on the SEV-ES-protected VM. After each round, we killed the VM and launched a fresh instance to avoid the results being influenced by previous rounds. In all 1000 rounds, we successfully wrote our string to the fixed location in the VM's memory.

8 DISCUSSION

Due to unintended behavior in the GRUB bootloader, the boot parameter structure of the SEV-protected Ubuntu VM is currently partially filled with random values when it is first booted. This additional randomness prevents us from successfully pinning the KASLR offsets (Section 4) at the initial launch of the SEV-protected VM, which uses GRUB. However, as we showed in our evaluation (Section 7), our technique works on every subsequent reboot of the Ubuntu VM, which correctly initializes the boot parameters structure with zeroes. Furthermore, our attack works on all launches of a barebone VM with SEV-ES, as GRUB is not used. In this scenario, the OVMF [15] is responsible for populating the boot parameter structure before handing control to the Linux EFI boot stub [23]. While filling the boot parameters structure with random values could be interpreted as a defense mechanism against our approach, the Linux boot protocol requires the boot parameter structure to be initialized with zeros [1]. This issue therefore only poses a temporary limitation of our attack for entropy manipulation. We expect our approach to be fully compatible with all SEV features as soon as the behavior of GRUB is adapted to fulfill the requirements of the Linux boot protocol under SEV.

Using our approach for entropy manipulation (Section 4), we were able to pin the value of the first eleven stack canaries. Although it should be possible to make the sequence of stack canaries deterministic over a longer period by manipulating the injection of interrupts, we did not further verify this possibility and leave this up to future work.

The entropy manipulation attack as described in this work applies to the current implementation of KASLR. However, Kristen Accardi recently proposed a modification to KASLR to dynamically shuffle the kernel functions at boot time [13]. The corresponding code also relies on `kaslr_get_random_long` to select a new position for each function. Similar to previous examples, we would also in this scenario be able to control the source of entropy for the randomization function and thus be able to make the sequence of kernel functions deterministic.

One assumption we made for our entropy manipulation attack is that the HV is able to access the VM's unencrypted kernel. Also for the code execution attack, we make use of the `bzImage` that is being loaded, which provides a great amount of possible gadgets. However, the owner of the VM may decide to encrypt the `/boot` partition with the `bzImage`, and have the decryption be performed either by the bootloader or by OVMF. However, rendering the kernel image inaccessible to the HV would only be a temporary workaround to prevent our attack. For entropy manipulation, we could for example use a modified version of the approach presented by Werner et al. [31] to perform OS fingerprinting. Having determined the OS version of the VM, we would be able to download and inspect the respective kernel image. Additionally, the code execution attack is likely also applicable to OVMF since it also enables paging. This would allow us to perform an attack similar to the one described, but early in the boot process, circumventing the issue of not being able to access the `bzImage`.

For our GPT-corruption attack, our page tracking mechanism relies on specific conditions and would have to be adapted when the used triggers change within the VM. To simplify determining when to overwrite the VM's stack, we can make use of other events such as the `VMEXIT_CR[0-15]_WRITE_TRAP`. This trap causes a `VMEXIT` each time the VM writes to any of the control registers [5]. This allows us to detect any modification to the `CR0` and `CR3` registers, being able to precisely determine when to overwrite the VM's stack.

The presented attacks in this paper exploit missing validation of results returned by the untrusted HV. An interesting question is whether these attack vectors can also be applied to SEV-SNP, which is supported neither by any AMD CPU nor by Linux at the time of writing. The SEV-SNP specification [8] describes two features which may effectively mitigate our attacks on entropy manipulation and GPT corruption: *CPUID Reporting* and *CPUID Page*. The *CPUID Reporting* feature allows the VM to query the CPU firmware

for whether the HV has reported CPUID features which are actually unsupported by the CPU. However, this feature will not indicate if the HV does not report features which are present [8]. Thus, *CPUID Reporting* does not protect against the presented attacks since they all rely on reporting a feature being disabled. In comparison, according to our understanding, the *CPUID Page* feature is a page injected by the HV when launching the VM, and the content of the page is included into the attestation measurement. One possible usage of the *CPUID Page* feature is to have the HV communicate all possible CPUID results once, and then have the VC handler query the structure when CPUID is intercepted. This prevents the HV from providing inconsistent information during the VM's execution. However, both of these features need to be explicitly used by software in the VM.

9 DEFENSES

While the issues described in this report also apply to traditional virtualized environments, they are of much higher relevance to SEV-protected VMs due to the stronger attacker model. First, SEV must blacklist security critical `virtio` devices such as `virtio-rng` and `virtio-crypto`, as they pose security risks for software which utilizes their respective interfaces. This can be achieved by disabling `CONFIG_HW_RANDOM_VIRTIO` and `CONFIG_CRYPTO_DEV_VIRTIO` in the kernel build scripts provided by AMD [2]. We did create a pull request to apply this defense, which has been accepted and merged into the official repository [6].

To mitigate our attack for entropy manipulation (Section 4), it would be required to disallow the HV to advertise RDRAND as unsupported. Further, we suggest to make RDRAND a required feature for running SEV-protected VMs. Based on a suggestion from Joerg Roedel (SUSE), we implemented a kernel patch to verify the cached CPU capability information in the early boot stage and in the kernel initialization phase. The added validation is a sufficient countermeasure against the proposed attacks for pinning KASLR offsets and stack canaries in the VM's Linux kernel. The patch is included in the official SEV-ES patch set [26] which is currently in review.

Another mitigation would be to prevent the HV from manipulating the VM's view on the TSC, which has to be performed by the CPU firmware. For this, a new TSC field could be added into the VM Save Area (VMSA), which is the structure that contains the vCPU's encrypted register state. The firmware would be responsible for restoring the TSC register when the VM is resumed and saving it when the VM exits.

To mitigate the MMIO forgery vulnerability (Section 5), the VM's VC handler would have to check if the accessed page is unencrypted when receiving an MMIO/NPF exception. As MMIO accesses will only happen on unencrypted pages, an MMIO/NPF exception raised from accessing an encrypted page indicates an attack. This behavior has been implemented for the Linux kernel by Joerg Roedel (SUSE) directly after disclosure of the attack [27]. A similar change would be necessary in OVMF as well.

Our last mitigation addresses the attack of corrupting the GPT to gain code execution inside the VM (Section 6). After corruption of the GPT, the VM's instruction fetches remain encrypted while the VM's memory accesses now interpret memory as unencrypted. This

```
1 mov GPT, cr3
2 cmpl 0xfff63d81, -10(rip)
3 je .access_is_ok
4 /* take appropriate actions */
5 .access_is_ok: /* test passed, then return */
6 ret
```

Figure 11: Code for detecting GPT corruption. On Line 2, the instruction compares the expected four bytes of its machine code with the four bytes in memory. If the test fails, the GPT has been corrupted and the VM can take action.

behavior can be used to create the mitigation shown in Figure 11. After the new GPT is set on Line 1, the instruction on Line 2 reads the first four bytes of the current instruction and compares it with an immediate value which corresponds to the first four bytes of the machine code of the current instruction. If the read bytes do not match the immediate, the GPT has been corrupted, and the VM can take appropriate measures. Otherwise, the VM jumps to the label `.access_is_ok` and returns. We verified that this test successfully detects our attack and protects against it. However, the code running inside the VM needs to be carefully analyzed to determine all places where the test needs to be applied.

10 RELATED WORK

Checkoway et al. [10] manipulated the return values of syscalls from the Linux kernel to cause protected applications to act against their own interests. The authors showed that in order to prevent against kernel attacks, return values from the syscall interface must be sanitized. Our work builds on the idea of Checkoway et al. and applies their approach to SEV- and SEV-ES-protected environments.

One of the first attacks on SEV was presented by Hetzelt et al. [16] and made use of the unprotected SLAT table. By remapping entries in the SLAT to different pages in the physical memory, the authors were able to modify the control flow of an SSH server to allow an attacker to login without valid credentials. The remapping approach was later extended by Morbitzer et al. [21], who managed to extract the encrypted memory of a VM in plaintext by making use of a service running inside the VM. The two attacks exploit the missing integrity protection of the SLAT, and are similar to the exploitation of the MMIO region forgery vulnerability discussed in Section 5. However, our approach offers fine-grained manipulation of memory accesses, and additionally allows for tracing the VM's execution since each memory access traps in the HV.

Du et al. [12] were the first to reverse engineer the cipher mode used for encrypting the VM's memory. This allowed them to patch instruction sequences of an SSH server running in the VM in order to login without valid credentials. The knowledge of the encryption algorithm was also used by Li et al. [18], who exploited the I/O channel of the VM to create an encryption oracle. Afterwards, Wilke et al. [32] reverse engineered the cipher mode and tweakable functions on newer AMD CPUs. They made use of their discovery by moving ciphertext blocks in the VM's memory in such a way that they were able to chain short sequences of code. However, the two attacks are not applicable to recent generations of AMD CPUs where the tweakable functions are difficult to compute [32]. In

comparison, the MMIO region forgery and GPT-corruption attacks do not require knowledge of the cipher mode and tweak values. Thus, these two attacks are also applicable to recent generations of EPYC CPUs.

Werner et al. [31] monitored general purpose registers of the VM to extract confidential information such as encryption keys. However, using SEV-ES, the register state is protected, which prevents their attack. In comparison, our approach of using the `virtio-crypto` devices allows us to extract secret keys even in SEV-ES protected VMs. Furthermore, our MMIO region forgery attack can be used to leak the address and data values of a memory write with SEV-ES, which similarly can be employed to steal encryption keys. Another advantage lays in our GPT-corruption approach, which can be used to extract or inject arbitrary data and also works on SEV-ES.

Werner et al. [31] also presented a second attack, which allows to fingerprint applications running in the protected VM by using Instruction Based Sampling (IBS) even under SEV-ES. Morbitzer et al. [20] took a different approach, and located confidential information in a VM's encrypted memory by purposely removing page access rights in the SLAT table and analyzing the VM's access pattern. To extract the information, they require a method to extract the located data. Also Buhren et al. [9] analyzed access patterns of VMs. By using machine learning, they were able to detect which syscalls applications in the VM were issuing. Our work on fixing the KASLR offsets can be seen as a great addition to those contributions. For example, disabling KASLR could simplify locating secret data within the VM's encrypted memory, or to identify syscalls. Additionally, our MMIO region forgery attack would allow to extract or modify data which has been located within the VM's memory using the approach of Morbitzer et al. [20].

11 CONCLUSION

In this work, we showed the security implications of introducing a stricter attacker model into a complex code base for which some interfaces are no longer trusted. We emphasized the risk of this scenario by showing how a malicious HV can 1) extract cryptographic keys through virtual devices, 2) disable probabilistic software defenses, 3) intercept regular memory accesses and 4) gain code execution inside the encrypted VM.

We showed how a malicious HV can make use of `virtio` devices to extract secret data such as encryption keys or plaintext data from the VM's Kernel Crypto API. Further, we utilized security critical `virtio` devices to control the entropy for software running in the encrypted VM. Also, we presented an approach for a malicious HV to influence the generation of random numbers in an SEV- and SEV-ES-protected VM by intercepting CPU instructions. This enabled us to reduce the entropy of probabilistic defenses such as KASLR and stack canaries in the VM's kernel. Afterwards, we demonstrated how missing validation in the handling of `MMIO/NPF` events can lead to data exfiltration and code injection for an SEV-ES-protected VM. Finally, we presented an approach to trick an SEV- or SEV-ES VM to decrypt its stack which leads to code execution via ROP.

While these attacks can be applied to all virtualized environments, they are especially critical for environments such as SEV and SEV-ES, in which the HV is considered untrusted. We showed this at the example of the Linux kernel, which is making use of HV-controlled interfaces for critical security features. Our work reveals that software running within an SEV- or SEV-ES-protected VM must not trust any input from the HV and carefully verify all external data.

12 ACKNOWLEDGMENTS

This work has been funded by the Fraunhofer Cluster of Excellence "Cognitive Internet Technologies"[1].

We would like to thank David Kaplan and Tom Lendacky from AMD, as well as Joerg Roedel from SUSE for the quick responses and fixes.

REFERENCES

[1] 2019. The Linux/x86 Boot Protocol. https://www.kernel.org/doc/Documentation/x86/boot.txt. Accessed: 2020-28-05.
[2] Advanced Micro Devices. 2018. GitHub - AMDESE/AMDSEV: AMD Secure Encrypted Virtualization. https://github.com/AMDESE/AMDSEV.
[3] Advanced Micro Devices. 2018. Open-Source Register Reference for AMD Family 17h Processors (PUB).
[4] Advanced Micro Devices. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.
[5] Advanced Micro Devices. 2020. AMD64 Architecture Programmer's Manual (Volumes 1-5).
[6] Advanced Micro Devices. 2020. GitHub - AMDESE/AMDSEV. https://github.com/AMDESE/AMDSEV. Accessed: 2020-05-14.
[7] Advanced Micro Devices. 2020. SEV-ES Guest-Hypervisor Communication Block Standardization.
[8] Advanced Micro Devices. 2020. SEV Secure Nested Paging Firmware ABI Specification.
[9] Robert Buhren, Felicitas Hetzelt, and Niklas Pirnay. 2018. On the Detectability of Control Flow Using Memory Access Patterns. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX)*. ACM.
[10] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: why the system call API is a bad untrusted RPC interface. *SIGARCH Computer Architecture News* (2013).
[11] David Kaplan. 2017. Protecting VM Register State with SEV-ES. White Paper.
[12] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. 2017. Secure Encrypted Virtualization is Unsecure. arXiv:1712.05090 [cs.CR] https://arxiv.org/abs/1712.05090
[13] Jake Edge. 2020. Finer-grained kernel address-space layout randomization. https://lwn.net/Articles/812438/. Accessed: 2020-27-05.
[14] Edited by Michael S. Tsirkin and Cornelia Huck. 2019. *Virtual I/O Device (VIRTIO) Version 1.1.* OASIS Committee Specification 01. Latest version: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html.
[15] Laszlo Ersek. 2014. Open Virtual Machine Firmware (OVMF) Status Report. http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt. Accessed: 2020-31-05.
[16] Felicitas Hetzelt and Robert Buhren. 2017. Security Analysis of Encrypted Virtual Machines. In *International Conference on Virtual Execution Environments*.
[17] David Kaplan, Jeremy Powell, and Tom Woller. 2016. *AMD memory encryption*. Technical Report. Advanced Micro Devices.
[18] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security Symposium*.
[19] Mathias Morbitzer and Manuel Huber. 2019. Github - SEVered Framework. https://github.com/Fraunhofer-AISEC/severed-framework/.
[20] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting Secrets from Encrypted Virtual Machines. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*.
[21] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *European Workshop on Systems Security (EuroSEC)*.
[22] Stephan Müller. 2019. Documentation and Analysis of the Linux Random Number Generator.
[23] The Linux Kernel Organization. 2018. The EFI Boot Stub. https://www.kernel.org/doc/Documentation/efi-stub.txt. Accessed: 2020-28-06.
[24] QEMU. 2020. QEMU Wiki - VirtioCrypto. https://wiki.qemu.org/Features/VirtioCrypto. Accessed: 2020-05-26.

[1] https://www.cit.fraunhofer.de

[25] QEMU. 2020. QEMU Wiki - VirtIORNG. https://wiki.qemu.org/Features/VirtIORNG. Accessed: 2020-05-26.
[26] Joerg Roedel. 2020. [PATCH v4 00/75] x86: SEV-ES Guest Support. https://lkml.org/lkml/2020/7/14/581. Accessed: 2020-12-10.
[27] Joerg Roedel. 2020. x86/sev-es: Do not support MMIO to/from encrypted memory. https://git.kernel.org/pub/scm/linux/kernel/git/joro/linux.git/commit/?h=sev-es-tip-updates&id=5282faf01e085d57658a39494ea760c2b7309f3d. Accessed: 2020-12-10.
[28] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. In *ACM Transactions on Information and System Security (TISSEC)*.
[29] Brijesh Singh. 2017. x86: Secure Encrypted Virtualization (AMD). https://lwn.net/Articles/716165/. Accessed: 2020-22-05.
[30] The kernel development community. 2020. Kernel Self-Protection. https://www.kernel.org/doc/html/latest/_sources/security/self-protection.rst.txt. Accessed: 2020-23-04.
[31] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.
[32] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity - Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE Symposium on Security and Privacy (S&P)*.