undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation

Luca Wilke, Jan Wichelmann, Florian Sieck, Thomas Eisenbarth
University of Lübeck, Germany
{l.wilke,j.wichelmann,florian.sieck,thomas.eisenbarth}@uni-luebeck.de

***Abstract*—The ongoing trend of moving data and computation to the cloud is met with concerns regarding privacy and protection of intellectual property. Cloud Service Providers (CSP) must be fully trusted to not tamper with or disclose processed data, hampering adoption of cloud services for many sensitive or critical applications. As a result, CSPs and CPU manufacturers are rushing to find solutions for secure and trustworthy outsourced computation in the Cloud. While enclaves, like Intel SGX, are strongly limited in terms of throughput and size, AMD's Secure Encrypted Virtualization (SEV) offers hardware support for transparently protecting code and data of entire VMs, thus removing the performance, memory and software adaption barriers of enclaves. Through attestation of boot code integrity and means for securely transferring secrets into an encrypted VM, CSPs are effectively removed from the list of trusted entities. There have been several attacks on the security of SEV, by abusing I/O channels to encrypt and decrypt data, or by moving encrypted code blocks at runtime. Yet, none of these attacks have targeted the attestation protocol, the core of the secure computing environment created by SEV. We show that the current attestation mechanism of Zen 1 and Zen 2 architectures has a significant flaw, allowing us to manipulate the loaded code without affecting the attestation outcome. An attacker may abuse this weakness to inject arbitrary code at startup—and thus take control over the entire VM execution, without any indication to the VM's owner. Our attack primitives allow the attacker to do extensive modifications to the bootloader and the operating system, like injecting spy code or extracting secret data. We present a full end-to-end attack, from the initial exploit to leaking the key of the encrypted disk image during boot, giving the attacker unthrottled access to all of the VM's persistent data.**

I. Introduction

An increasing number of software applications, from enterprise management software to messengers used in nearly everyone's daily life, rely on storing information and performing computations in the cloud. Solutions are moved from local, trusted environments to the data centers of big cloud service providers, and are now running in untrusted environments under control of a third party—in order to save costs, reduce management effort and to improve scalability.

The loss of trust comes with significant challenges for services such as banking, private secure messaging or health services, which require strict isolation and confidentiality to ensure the safety of their assets and to comply with data privacy laws: Computing resources in the cloud are often shared, which in case of broken isolation does allow co-located users to spy on each other [23, 30, 45]. Another concern is the security of the cloud service provider's systems themselves, where internal or external attackers may leverage elevated privileges for extracting private data.

In order to deliver isolated, confidential and authenticated execution and processing of data in an otherwise untrusted setting, processor vendors added hardware features to build a root-of-trust and ensure confidential computing in a local Trusted Execution Environment (TEE). One example is AMD SEV [4, 5, 27], which allows to run VMs confidentially and isolated from their hypervisor. AMD added new features to SEV with every generation of its processor architecture. In 2017, The first generation of EPYC processors (Zen) came with the initial version of SEV. The second generation (Zen 2) added an encrypted state for context switches with SEV-ES [26] and was released in 2019. The newest addition, SEV-SNP [3], will be available on the third generation of EPYC processors (Zen 3), which are set to be released in early 2021. Intel TDX [24] aims to provide a similar solution, but is only available as a concept as of writing this work. With Intel Software Guard Extensions (SGX) [9, 17, 25], Intel offers an established TEE which enables software vendors to run smaller programs in isolated enclaves. All of these solutions provide memory encryption during execution, and attestation of the software loaded into the TEE.

Recently, cloud service providers like Microsoft and Google started to offer confidential computing environments which isolate the customer's software using Intel SGX [31] or AMD SEV [21]. Popular examples, like the secure private messenger Signal, are already using these technologies to protect the sensitive data of their customers [38]. Moreover, open source solutions enable simple development and deployment of software for TEEs [10, 11, 19].

A fundamental challenge for TEEs is having to guarantee their promises against attackers with system level privileges, resulting in a large variety of attacks [15, 16, 32, 37, 42, 44]. In this work, we extend the arsenal of attacks against TEEs and in particular against AMD SEV, with an attack targeting and circumventing its very core of trust, the remote attestation.

Remote attestation allows the owner of a software, which runs in a confidential or trusted execution environment, to verify the initial integrity and authenticity of the software loaded into the TEE, which afterwards is preserved at runtime by the properties of the TEE. Generally, remote attestation works by creating a signed measurement, usually a hash, of the initially loaded application through the trusted hardware and sending this measurement to the software owner for verification. In case of AMD SEV, the trusted hardware is an additional on-chip co-processor called Secure Processor (SP), which cannot be externally controlled.

A. Our Contribution

If the attestation process, however, is broken, the isolation and confidentiality guarantees of AMD SEV are inconsequential as the software owner cannot be sure whether their intended software was loaded or whether an attacker manipulated it during startup.
In this work, we

- show that the measurement used in AMD SEV's attestation is block permutation-agnostic, meaning that changing the order of measured memory blocks does not affect the attestation outcome, and thus allows the attacker to modify the execution flow without detection by the VM's owner;
- construct an universal attack primitive, which reorders the measured blocks of an initially loaded UEFI and sets up a Return-oriented Programming (ROP) chain to load and execute arbitrary code;
- demonstrate a full end-to-end attack which leaks the key of an encrypted disk image, and gives the attacker full control over the VM's operating system;
- propose several countermeasures and discuss why the underlying problem ultimately cannot be solved under SEV Encrypted State (SEV-ES).

B. Attack Overview

The attack described in this work targets the measurement of the initially loaded binary during startup of an SEV-ES-protected Virtual Machine (VM) (guest). When the VM is started, the hypervisor instructs the AMD SEV secure processor to load the initial binary, e.g. an Open Virtual Machine Firmware (OVMF) UEFI binary, into encrypted memory and calculate a measurement of the initial VM content using the `LAUNCH_UPDATE_DATA` and `LAUNCH_MEASURE` commands. We find that the initial binary can be split into blocks as small as 16 bytes, which we are able to load in an arbitrary order using `LAUNCH_UPDATE_DATA` , while still getting the same measurement when calling `LAUNCH_MEASURE`. This allows us to construct our own execution flow, which we use for redirecting the stack pointer to an unencrypted shared page. Consequently, we leverage this control over the stack to mount a ROP attack, allowing us to write arbitrary code and data into the encrypted VM's memory. We use the injected code to leak the protected secret values which have been provided by the guest owner. As our meddling with the block ordering does not change the launch measurement, AMD SEV's remote attestation will succeed and the guest owner will be unaware of our changes to their VM's execution flow.

C. Responsible Disclosure

We responsibly disclosed our findings to AMD via Email on January 19th, 2021. AMD requested an embargo until May 11, 2021 and provided us with the following statement: "AMD has assigned CVE-2021-26311 for this issue and provided mitigations in the SEV-SNP feature available for enablement 3rd Gen AMD EPYC™ processors. AMD appreciates the coordination efforts made by the research team.".

II. Background

A. AMD SEV

In 2016, AMD introduced their Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) technologies [27], which were implemented only in 2017 with the first generation of EPYC processors (Zen 1). SME offers hardware-based encryption of RAM content. The memory encryption key is managed by the SP, an ARM-based co-processor, and is thus never accessible by system software. The encryption/decryption takes place directly in the on-die memory controllers. Each page table entry has a special status bit, which controls whether the associated page is encrypted or not. The whitepaper [27] does not explain the mode of operation in detail, but only states that AES with an 128-bit key and a physical address-based tweak is used. In [18, 44] it is shown that early versions use the Xor-Encrypt (XE) or Xor-Encrypt-Xor (XEX) encryption mode with static, low entropy tweak values, while later versions use stronger, randomized tweak values. In addition, none of the encryption modes offer integrity protection.

While SME uses the same key for all memory pages, SEV adds the ability to encrypt the memory content of VMs with different keys, that are only known to the SP but not to the Hypervisor (HV), preventing a malicious HV from directly reading the memory content of its guests. However, VMs can also share pages with the HV. In addition, the SP offers an API to the HV to manage the SEV-protected VMs. This includes a mechanism to attest the initially loaded code of the VM and a mechanism to securely move secrets into the VM.
SEV-ES was introduced by AMD in 2017 and implemented in 2019 with the second generation of EPYC processors. It addresses one major remaining attack surface of SEV: The unencrypted Virtual Machine Control Block (VMCB), a data structure storing certain configuration bits as well the VM's register values on context switches. Certain sensitive parts of the VMCB were moved to a substructure called Virtual Machine State Save Area (VMSA) that is encrypted and integrity protected on context switches to the HV, and thus prevents an attacker from inferring or modifying a VM's state during context switches.

However, there are also several instructions that need interaction with the HV, like `cpuid`, which previously shared and received data with/from the HV via the VM's registers. To enable this with SEV-ES, AMD introduced a new communication mechanism between HV and VM, consisting of the Guest Hypervisor Communication Block (GHCB) and a new exception called VMM Communication Exception (`#VC`). The GHCB is simply a shared page, that gets setup by the VM. Instructions that require data sharing with the HV cause a `#VC`, allowing a VM exception handler to share the data via the GHCB.
SEV Secure Nested Paging (SEV-SNP) aims to address several remaining issues, like remapping attacks due to the HV's control over the nested page tables, or attacks on the missing integrity protection. It was announced by AMD in

January 2020 [3] and will only be available on the 3rd gen EPYC processors, that are set to be released after the submission deadline. The most important change is the introduction of an additional page table called Reverse Map Table (RMP), to which the HV only has mediated access. The RMP aims to ensure a one-to-one mapping between Guest Physical Addresses (GPAs) and Host Physical Addresses (HPAs), and will prevent the HV from writing to VM memory, mitigating problems arising due to the lacking integrity protection.

B. Booting physical and virtual environments

When booting a physical system, the CPU is in a well-defined state, but completely unaware of its environment. Its program counter is set to start execution at a fixed address, which points to a FLASH or EPROM. Firmware is loaded from this start position and is responsible for initializing the memory controller, creating a memory mapping for RAM, and configuring I/O peripherals. On modern computers, this firmware usually is UEFI-based, which is platform specific and performing the aforementioned tasks. The advantage of UEFI compared to legacy BIOS is its standardization of the platform initialization procedure [41]. The UEFI is configurable through variables in a non-volatile memory (e.g. NVRAM), so configuration is persisted over restarts. Additionally, it can provide *secure boot*, which allows to build up a chain of trust from the UEFI to the finally loaded kernel of the OS. In this chain, the UEFI, which contains a set of configurable certificates and keys, forms the root of trust. Every component of the chain verifies its successor before handing over the execution [41].

When the UEFI finishes the system configuration, it hands over the control to an EFI binary [41], which usually is an OS bootloader, e.g. Grub [20]. The bootloader sets up the stage for the OS kernel, loads the kernel into memory and calls its main method. However, an EFI binary does not necessarily have to be a bootloader.

In case of booting a virtual environment, e.g. with QEMU [36], the process is similar: When the hypervisor starts up the virtualization, it launches an UEFI. For virtual environments, OVMF [40] is a common choice. OVMF performs the necessary virtual system configuration and hands over control to a bootloader [12]. The bootloader, which is just a regular EFI application, can be provided in different ways. Usually it is expected to be located on a FAT-formatted disk with GUID partition table [41], thus requiring the guest owner to provide the hypervisor with a disk image. Another way is to include the bootloader, i.e. the EFI application, into the UEFI volume which also contains the UEFI firmware [13].

III. SEV(-ES) Guest Launch Process

In this section, we describe the typical workflow required to launch an SEV-secured VM, as described in [5] and implemented in AMD's patches to the the Linux kernel [1] and QEMU [2]. This includes encrypting an initial code image, proving its integrity to the guest owner, and loading secret data without leaking it to the HV.

TABLE I. Overview of the VM-specific commands provided by the SP in different states. For brevity, commands that are not relevant for our work were omitted.

State	Command	→	New State
`UNINIT`	`LAUNCH_START`	→	`LUPDATE`
`LUPDATE`	`LAUNCH_UPDATE_DATA`	→	`LUPDATE`
	`LAUNCH_UPDATE_VMSA`	→	`LUPDATE`
	`LAUNCH_MEASURE`	→	`LSECRET`
`LSECRET`	`LAUNCH_SECRET`	→	`LSECRET`
	`LAUNCH_FINISH`	→	`RUNNING`
`RUNNING`	(other commands)		

A. Prerequisites

There are three parties involved in launching a SEV VM: The guest owner, the HV and the SP. The guest owner wants to start a SEV-secured VM. The HV is typically controlled by the cloud service provider. In order to provide the SEV functionality, the HV must interact with the API provided by the SP.

The goal of the launch process is to enable the HV to prove to the guest owner that the initial content is trustworthy. Furthermore, it enables the guest owner to send secrets, like disk encryption or SSH keys, to the VM in a secure manner. The entire launch process is illustrated in Figure 1.

For each VM, the SP maintains a Guest Context (GCTX) that, among other values, contains a handle, the VM Encryption Key (VEK), the launch digest (LD), and the current state. The VEK is a VM-specific key used for memory encryption. The launch digest contains a hash value of the VM contents loaded during the launch measurement phase. The state determines which API commands are usable.

Table I shows an overview of the states along with the usable commands and the resulting state transitions. We omitted all states and commands related to migrating VMs between different hosts, as we do not use them in this paper. In addition to the VM-specific commands, there are several commands which affect the SP itself. They are used to update its firmware and to generate or export cryptographic key material.

Before any VM-specific commands are issued, the HV starts an ECDH key exchange by issuing the `PDH_CERT_EXPORT` command, upon which the SP exports a public ECDH key and some certificates. The latter are part of a public key infrastructure, that is ultimately rooted at an AMD controlled key hardcoded into the SP. The HV then sends this data to the guest owner.

B. `UNINIT` state

A new VM assumes `UNINIT` as initial state. In order to start the launch process, the guest owner verifies the authenticity of the ECDH key sent by the HV. Then they use their own ECDH key pair to derive the Transport Encryption Key (TEK), the Transport Integrity Key (TIK) and some other keys used for transport security. Afterwards, they send this data together with a configuration object called `POLICY` to the HV.

Upon receiving the data from the guest owner, the HV calls the `LAUNCH_START` command, which finalizes the ECDH handshake between guest owner and SP. The SP can now

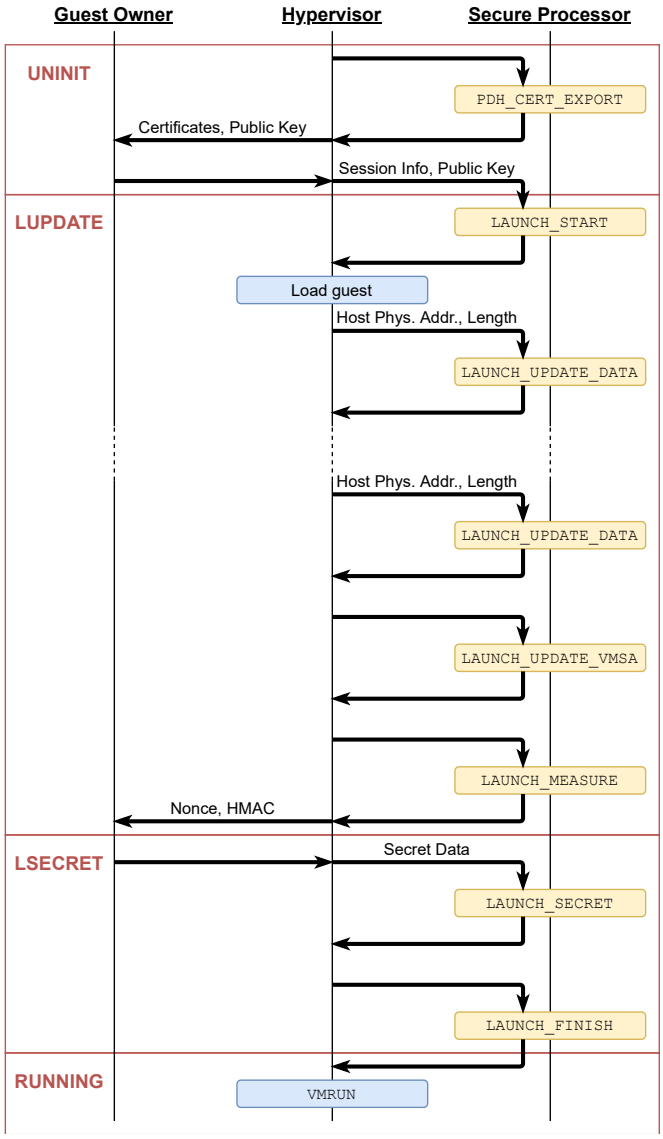

Fig. 1. SEV Guest Launch Process. This simplified illustration shows the various states during VM startup and attestation. First, keys are exchanged and a cryptographic session is established. In the LUPDATE state, the hypervisor loads the guest and then asks the SP to encrypt it via repeated LAUNCH_UPDATE_* commands. The final LAUNCH_MEASURE call retrieves a signed hash of the loaded guest data. If the guest owner approves, various secret data can be safely loaded into the VM during the LSECRET state. On completion, the hypervisor and the VM transition into the RUNNING state, and the VM is executed.

derive the shared secret and use it to unwrap/verify the received data. Next, it initializes the GCTX using the received guest policy and generates a new VEK.

C. *LUPDATE state*

In the LUPDATE state, there are three primary commands: LAUNCH_UPDATE_DATA, LAUNCH_UPDATE_VMSA and LAUNCH_MEASURE. The LAUNCH_UPDATE_DATA command allows the HV to specify a guest handle, a 16 byte aligned HPA PADDR and a multiple of 16 bytes L as a length. The SP will then in-place encrypt the next L bytes starting at PADDR with the VEK of the VM denoted by the handle. In addition, the launch digest field of the GCTX is updated with the plaintext of the encrypted data (see Section V-A). The intention of LAUNCH_UPDATE_DATA is to encrypt and measure the initial content of the VM, such that the HV can no longer modify it. Encrypting the initial content is mandatory, since the VM initially assumes that all memory accesses are encrypted, so it can only execute the initial code if it has been encrypted beforehand.

The LAUNCH_UPDATE_VMSA command is only applicable to SEV-ES VMs and works very similar to LAUNCH_UPDATE_DATA, except that it can only load 4096 bytes, as it is intended to encrypt the VMSA. In addition, it also initializes the VMCB. While not enforced, this is intended to be called only once. Again, the launch digest is updated with the loaded data.

The third and final command, LAUNCH_MEASURE, generates a launch measurement and transfers the VM to the LSECRET state. The measurement consists of a 128-bit nonce MNONCE and a 256-bit HMAC MEASURE, that is calculated as follows:

1) Replace launch digest (LD) with $\mathrm{hash}(\mathtt{LD})$
2) Calculate

$$\mathrm{HMAC}(\mathtt{0x04} \,\|\, \mathtt{API\_MAJOR} \,\|\, \mathtt{API\_MINOR} \,\|\, \mathtt{BUILD} \,\|\, \mathtt{POLICY} \,\|\, \mathtt{LD} \,\|\, \mathtt{MNONCE}, \mathtt{TIK})$$

MNONCE is generated by the SP, API_MAJOR and API_MINOR and BUILD specify the version of the firmware on the SP. POLICY is the configuration structure that was sent by the guest owner in the UNINIT state.

Next, the HV sends the launch measurement to the guest owner, in order to prove that it did not manipulate the initial content. It is assumed that the guest owner and the HV/cloud service provider negotiated the initial content of the VM, e.g., that the guest owner stated that they want a specific UEFI version to be loaded. Thus, the guest owner has all the information required to compute the HMAC themselves and compare it to the value they received.

After successfully checking the launch measurement, the guest owner can be sure that the initial memory content matches their specification. Since, on startup, the VM treats any memory as encrypted, it is unlikely that the HV can achieve any meaningful manipulation of the VM's code and data by tampering with its memory. The only possibility for the HV to encrypt data with the VM's key is by using the designated LAUNCH_UPDATE_* commands, but, as already explained, this has the side effect of updating the launch digest and thus changing the HMAC sent in the attestation report, allowing detection by the guest owner. As only the SP and the guest owner know the TIK used to key the HMAC, the HV cannot produce valid HMACs itself.

D. *LSECRET state*

After the VM has transitioned into the LSECRET state, two commands become available: LAUNCH_SECRET and

LAUNCH_FINISH. The LAUNCH_SECRET command again allows to encrypt data with the VM's VEK. However, contrary to the previous commands, the data passed to the command now is encrypted with the TEK and integrity protected by an HMAC keyed with the TIK. Both keys are only known to the SP and the guest owner. If the integrity check fails, the command aborts. The guest owner can use this mechanism to safely send confidential data (e.g., disk encryption keys) to the VM, while using the HV as a proxy. The HV could refuse to relay the data to the SP, but it can neither manipulate the data nor call the command with self-generated data, as it does not know the TIK needed to pass the HMAC check.

Finally, the LAUNCH_FINISH command transitions the VM into the RUNNING state, indicating that the VM is ready to be started. The LAUNCH_SECRET and LAUNCH_FINISH commands are disabled afterwards.

IV. Attacker Model

The attacker model is in line with SEV's security model: The attacker controls the hypervisor, and is able to modify arbitrary physical memory and run or pause the VM according to their wishes. However, they are not able to read or modify the current register state and program counter of the VM, as its state is encrypted using SEV-ES.

They know the initially launched code, since that needs to be available in plaintext in order to be loaded into the VM. We assume that the attestation is working in so far that the attacker has to actually load and attest the supplied initial code image, and cannot simply replace it with their own. We also assume that the attestation protocol is carried out correctly, such that the guest owner is assured that supposedly the correct image was loaded, and subsequently launches the virtual machine.

The attacker is not able to read or modify encrypted disk images without knowing the corresponding key. Finally, we consider the SP itself to be secure.

V. Exploiting SEV's Permutation Agnostic Launch Measurement

Given the VM attestation process laid out in Section III, we show how an attacker can deviate from the intended startup process in order to make the VM execute arbitrary code, which corresponds to a full break of confidentiality and integrity.

In a first step, we show that SEV's launch measurement can be tricked into producing the same measurement for any blockwise permutation of the initial VM content. We illustrate how an attacker can use this flaw to construct an encryption/decryption/code execution gadget, that runs within the VM, but does not change its launch measurement and thus cannot be detected by the guest owner. Finally, in Section VI, we discuss the implications of our attack for the transition from initially attested code to code residing on a virtual hard disk, and demonstrate how we can use our attack to leak secrets.

A. Breaking the Launch Measurement

First, we show how a malicious HV can abuse a flaw in the launch process to change the semantics of the loaded data without changing the launch measurement.

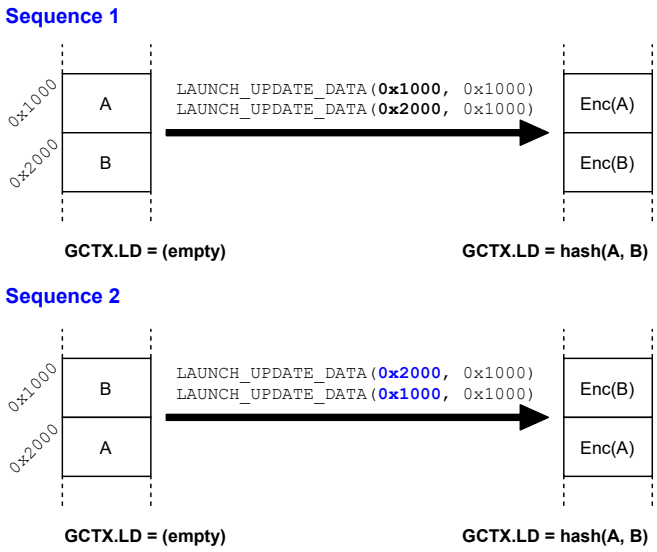

Fig. 2. Two encryption sequences, yielding the same launch measurement GCTX.LD for different orders of memory blocks. In the first sequence, the memory pages A (address 0x1000) and B (address 0x2000) are encrypted in memory order, i.e., LAUNCH_UPDATE_DATA is first called for block A, then for block B. In the second sequence, the blocks are swapped in memory: A now resides at address 0x2000, while B is at address 0x1000. By changing the order of calls to LAUNCH_UPDATE_DATA, we are able to acquire the same value for GCTX.LD in sequence 2 as for the "correct" ordering in sequence 1. The guest owner thus has no means for distinguishing which sequence has been used by the HV.

As described in Section III, the HV uses the LAUNCH_UPDATE_DATA command to load and encrypt the initial memory content of the VM. The command takes a 16-byte aligned HPA PADDR and a multiple of 16 bytes as length L, and then in-place encrypts L bytes starting at PADDR with the VM's VEK. In addition, the command updates the launch digest which is later used in the launch measurement.

In our experiments, we observed that the content of the launch digest is neither influenced by the HPAs passed to LAUNCH_UPDATE_DATA, nor by the used block size and the resulting varying number of calls to the command. Instead, the encrypted data is simply "appended" to the launch digest. While the official documentation is unclear at this point, we suspect that the launch digest internally manages a SHA-256 hash state, which is updated each time after a certain amount of data was inserted.

This implies that the HV can change the memory layout of the loaded data without any impact on the resulting hash value, as long as it makes sure that the order in which the data is passed to LAUNCH_UPDATE_DATA matches the original order. The modified ordering is illustrated in Figure 2.

B. Constructing Malicious Code Gadgets

We can now use our observations to construct malicious code gadgets, solely by moving around 16-byte blocks and triggering interaction with the HV.

The general idea is very similar to the approach presented in [44], where the authors leverage control over the first and last bytes of 16-byte blocks to stitch together a sequence of "payload" instructions and direct jumps, which they subsequently use to build an encryption oracle within the VM. However, we cannot change a block's content here, as this would be detected during attestation.

We first scan the binary of the initial VM content, which, in our case, can be split up into 230'000 16-byte blocks, for the instructions that we want to execute in our gadgets. For this, we are not bound to the originally intended decoding order: As x86 instructions have variable length and are not prefix-free, starting to decode the binary with different offsets can lead to different valid instructions. On the downside, this also means that decoding may fail because it encounters an invalid instruction encoding. To address this, we only look for "payload" instructions which reside at the end of a block or are followed by a direct jump, such that we can proceed to the next block.

Finally, we analyze the control flow of the original program to find a location where we can place our gadget, so it is executed at some point during the startup of the VM. We also make sure that our changes to the block ordering do not destroy the code needed to boot up the VM to the point where our gadget is entered.

C. Encryption/Execution Gadget

We can now use this block chaining technique to build a code gadget that enables the HV to encrypt (and decrypt) arbitrary data with the VEK and inject it into the VM.

For this, we assume that the VM is started with the default OVMF UEFI provided by AMD as initial memory content. Note that the ideas presented here are also applicable to other UEFI implementations that support SEV.

The encryption/execution gadget is constructed in two stages:

1) Permute the initial VM content, creating a gadget that maps the stack to an unencrypted shared memory page;
2) Use the control over the stack to construct a ROP gadget that copies data from the unencrypted shared page to encrypted memory and execute it as code. This code may later copy decrypted memory back to the shared page, or can be used to conduct other, more complex attacks.

Stage 1 In the first stage, we want to set the VM's stack (i.e., its `rsp` register) to an unencrypted memory page, to allow manipulation by the HV. Until reaching either long mode or legacy Physical Address Extension (PAE) mode, the VM's memory accesses are unconditionally treated as encrypted. Afterwards, the `C` bit in the page table controls whether a page is accessed in encrypted or unencrypted mode. The only exception are page table walks and instruction fetches, which are always treated as encrypted [4, Sec 15.43.4,15.34.5].

After startup, OVMF quickly progresses to long mode. While constructing its long mode page tables, OVMF also sets up a shared page for the GHCB protocol [6], which, under SEV-ES, is required to handle the emulation of instructions that need to share data with and/or receive data from the HV (c.f. Section II-A).

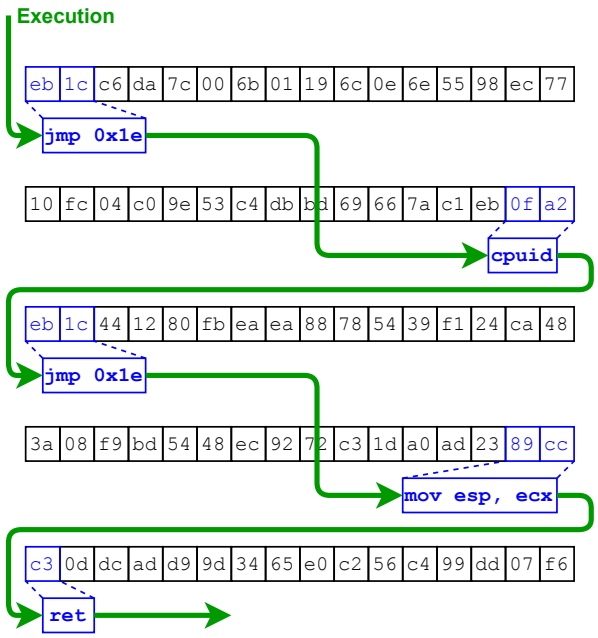

Fig. 3. Sequence of 16-byte blocks for setting the stack pointer to a HV-controlled address. The shown 16-byte blocks were taken from various places in the initial code image and moved to an address which is reached by the execution flow. The `jmp` instructions allow us to chain several blocks and skip potential junk bytes in between. After the stack pointer has been changed, the `ret` statement reads its return address from HV-controlled memory and thus triggers a ROP chain.

To load the address of the shared GHCB page into the `rsp` register, we opted for the following payload instruction sequence:

- `cpuid`
  Fills the `eax`, `ebx`, `ecx` and `edx` registers with processor feature information. As shown in previous work [44], one can abuse that this instruction is emulated by the HV and fill the `ecx` register with the virtual address of the shared GHCB page.
- `mov esp, ecx`
  Updates the stack pointer with the address of the shared page. Note the usage of 32-bit registers: This has the advantage of having a shorter instruction encoding than the 64-bit equivalent, while still being sufficient, as in OVMF the virtual address of the shared page is hardcoded to a small constant.
- `ret`
  Starts the ROP chain. As the stack pointer now points to the unencrypted shared page, the HV can place arbitrary return addresses (and other values) on the stack, which allows to conduct a classic ROP attack.

The resulting block chain is illustrated in Figure 3. To ensure the right timing for sending the manipulated `cpuid` register values, the HV simply counts the number of emulated `cpuid`

```
(1) 0x00000000fffd1ad0    pop rax
                          pop rbx
                          ret

(2) 0x00000000fffcef21    pop rdx
                          ret

(3) 0x00000000fffcfde6    mov qword ptr [rax], rdx
                          xor eax, eax
                          pop rbx
                          pop rbp
                          pop r12
                          ret
```

Fig. 4. ROP chain for writing data to VM memory. The ROP chain consists of three gadgets. Each gadget begins with a payload instruction (highlighted blue) and ends with a return statement, potentially with a few other instructions in between. Gadget (1) loads an unencrypted 8-byte address from the stack and writes it into `rax`. Gadget (2) loads an unencrypted 8-byte value from the stack and writes it into `rdx`. Finally, gadget (3) stores the value from `rdx` at the address pointed to by `rax`. The memory write in (3) triggers encryption of the data stored in `rdx`, as the address in `rax` points to encrypted memory.

instructions, which is deterministic in the executed OVMF code.

Stage 2 Coming from Stage 1, we now have full ROP capabilities.

In order to load additional data into the VM and execute it as code, we need to construct a ROP chain that copies data from the unencrypted stack to a memory location that is marked as encrypted. Then, we jump to that address using a `ret` instruction. This indirect approach of encrypting the code before execution is necessary, since, as mentioned in Stage 1, instruction fetches always assume that the underlying memory is encrypted. We abuse that, at the time of gadget injection, OVMF's page tables have all pages marked as readable/writable, without having a "no execute" bit. This simplifies our attack, since we do not have to build a ROP chain for modifying the page tables first.

Our resulting ROP chain is illustrated in Figure 4. It only needs three gadgets, and allows us to write 8 bytes to an arbitrary 64-bit address. We can then reuse the chain to write complex new code into the VM. In Section VI, we show how this code can be used to leak disk encryption keys.

In summary, we have seen that SEV's launch measurement mechanism is flawed, as it only attests that an arbitrary 16-byte granular permutation of the initial data has been loaded. Next, we have demonstrated the creation of malicious code gadgets just by swapping around 16-byte blocks. We have shown an instantiation of such a gadget, that maps the VM's stack to a shared page and employs a ROP attack to write additional data and execute it as code.

VI. Attack Case Study

With our encryption and code execution gadget from Section V-B, the attacker gains control over the initially executed code, and is able to insert their own. In the following, we highlight critical moments in the startup of a SEV-secured VM, and show how we can use our attack to take it over.

A. Experimental Setup

Our experiments are performed on a second generation AMD EPYC Processor 7232P. The firmware of the SP is version 24 build 0a [8] (most recent at time of writing). On the HV side, we use Linux Kernel 5.6 from the official AMD repository [1] (extended with our attack code) and QEMU [35] to start the VM. QEMU was extended with AMD's SEV-ES patches [2] and the proposed patches for the secret injection mechanism from [14] (see Section VI-C). Inside the SEV-ES victim VM, we run OVMF [39] and Grub [20]. Both were extended with the secret injection mechanism patches from [14]. We provide our proof-of-concept code alongside with the used software at https://github.com/UzL-ITS/undeserved-trust.

B. Trust Gap

For simplicity and scalability, only a small part of the VM's code is attested. In most cases, it suffices to attest a tiny initial code image, which takes owner-supplied secrets to load and decrypt a much larger encrypted disk image, which in turn contains the operating system and the processed data. The operating system can be considered trusted, as the attacker should not be able to modify the encrypted disk image without being in possession of the key.

The primary challenge is bridging the trust gap between the attested initial code image, which is available in plaintext, and the encrypted operating system: The guest owner needs to be able to supply secret information for decrypting the disk image, without an attacker being able to learn those secrets.

If an attacker gains access to a disk encryption key, they also gain unthrottled read/write access to all of the VM's data, even after the VM was shut down. They can abuse this access to extract secret data or manipulate the operating system.

C. Securely Injecting Secrets

To address this challenge, SEV offers the `LAUNCH_SECRET` command, which allows the guest owner to inject arbitrary secret values into the VM.

Since there is not yet a standardized toolchain, we focus on the proposed launch flow from [14], which has some of its patches already merged into the respective upstream repositories. Note that any other possible launch process will also need to bridge the aforementioned trust gap, and will thus be quite similar to the launch flow discussed here.

The proposed launch flow works as follows: The initial attested code image consists of both the OVMF UEFI binary and the Grub bootloader, which thus cannot be modified by a malicious HV. The UEFI performs the initial startup, and then transfers control to the bootloader, which in turn unlocks an encrypted disk image and boots the contained Linux kernel.

Both OVMF and Grub have been adjusted to respect the secret injection mechanism: At build time, OVMF includes a configuration table, which specifies the GPA where the HV (QEMU) should inject the secret. While preparing the VM startup, the HV scans the OVMF binary, locates the configuration table and subsequently injects the secret at the

```
5e      pop rsi     ; source address
5f      pop rdi     ; destination address
59      pop rcx     ; n
f3 a4   rep movsb   ; copy n bytes
f4      hlt         ; halt VM
```

Fig. 5. Code gadget for copying data. The register values are passed via the stack, to minimize the size of our injected code: By using the string copy instruction `rep movsb`, we can fit the entire gadget into 6 bytes, so we only need a single execution of the ROP chain from Figure 4. Since we only leak the secret, we halt the VM after copying the data; however, if we wanted to copy more data, we could replace the `hlt` instruction by a `ret` instruction and execute the gadget multiple times with different parameters.

indicated address. OVMF then passes the secret to the Grub bootloader, which uses it to unlock the disk.

D. Leaking the Disk Encryption Key

Given our attack primitives from Section V, leaking the disk encryption key is quite straightforward. We already know the length of the secret data, since the HV is responsible for receiving the encrypted secret from the guest owner and forwarding it to the SP via `LAUNCH_SECRET`, which expects a public length parameter. In addition, we know the GPA of the secret from OVMF's configuration table. Using our ROP chain, we can now inject a small code gadget which loops over the secret data and copies it into shared memory. The code gadget is shown in VI-D.

We halt the VM after extracting the secret; however, to avoid detection by the guest owner, we could also use our copy gadget to repair the code which we damaged by our block moving approach. Then, potentially after injecting further spy code into the previously encrypted disk image, we resume the boot process.

VII. Mitigations

Our attacks severely undermine the validity of AMD SEV's remote attestation. Unfortunately, it is not possible to fully mitigate our attack with the current capabilities of SEV-ES, due to the lack of page remapping protection, which will only be available with SEV-SNP on upcoming 3rd generation EPYC processors. Nonetheless, we discuss changes that could be applied to systems limited to SEV-ES, to make exploitation harder.

A. Increasing the Block Size

A simple countermeasure, which renders creating an exploit by reordering code blocks much harder, is to increase the minimal size of each measured block.

Given, e.g., our OVMF binary of 3.5 MB, a block size of 16 bytes yields around 230'000 blocks; for a block size of 4 kB, this number shrinks to merely around 900, greatly reducing a malicious HV's ability to find a block ordering that produces meaningful code, although it does not completely mitigate it. Note that `LAUNCH_UPDATE_DATA` already supports large block sizes: To improve performance, the corresponding kernel code tries to process contiguous physical memory blocks, which are chosen as large as possible.

Currently the protocol does not support specifying a certain block size or including it in the launch digest. However, it would be possible to implement a fixed block size without any changes to the protocol, by hardcoding a size of 4 kB (page) or even 2 MB (huge page) directly in the SP firmware. This way, the guest owner just needs to require the corresponding firmware version during attestation. Since the version check is already included in the protocol and the HV only has to update the SP firmware, this countermeasure is rather cheap. Only the SP firmware and client applications which verify the launch digest have to be changed.

B. Potential Changes to the Attestation Protocol

In order to further increase the power of the proposed countermeasure or even fully close the vulnerability, one may also include the physical addresses in the measurement and attestation, or increase the size of the measured blocks and add the block size to the measurement hash. This could be achieved by computing

$$h_i = \mathrm{hash}(h_{i-1} \,\|\, \mathrm{HPA}_i \,\|\, \mathrm{data}_i) \quad \text{or}$$

$$h_i = \mathrm{hash}(h_{i-1} \,\|\, \mathrm{block\_size}_i \,\|\, \mathrm{data}_i)$$

on each call i to `LAUNCH_UPDATE_DATA`, for a total of n calls, and submitting the list of addresses or block sizes along the measurement h_n. Both of these changes require changing the protocol of the remote attestation, since the address list or list of block sizes must be sent along the measurement itself.

However, both approaches are intrinsically limited by the underlying hardware assumptions and address mappings which are in place during virtualization. The HV can still legally reorder physically contiguous 4 kB pages, since it controls the mapping of host physical to guest physical memory addresses through its control over the Nested Page Table (NPT). I.e., the hypervisor is capable of performing page remapping attacks, as already exploited by Morbitzer et al. [34]. Thus, both approaches are limited to assure the correct order within and for the size of one memory page, which can already be achieved with fixing the block size to 4 kB (2 MB) as described in the previous section. The only advantage of including the physical addresses in the measurement is that we can ensure the order inside a 4 kB page, while still allowing 16 byte blocks as the smallest block size.

In conclusion, the attestation process is in need of fixing the loaded binary to addresses within the *guest's* address space: Adding the guest physical address, instead of the host physical address, to the measurement, and assuring that a remapping between guest physical address and host physical address after the initial allocation will be detected by the secure processor, would completely close the vulnerability described in this work. However, SEV-ES does not allow to detect a page remapping and thus only allows for a partial mitigation.

C. Changes in SEV-SNP

AMD SEV-SNP [3] is an upcoming extension to SEV-ES, which is only supported on the third generation of EPYC processors. As those only become available after the submission deadline, the following is solely based on the documentation [3, 4, 7].

One of the major changes in SEV-SNP is the introduction of the Reverse Map Table (RMP). The RMP is an additional page table, indexed by the HPA of a page. It adds several new attributes, mostly for distinguishing VM and HV pages, such that the HV cannot write to guest pages. In addition, the RMP contains the GPA of VM pages and can be used to ensure a one-to-one mapping between GPA and HPA to prevent remapping attacks. In contrast to traditional page tables, the HV does not have full control over the RMP, as it must use hardware- and firmware-mediated ways to access it.

While the general flow of the launch process does not appear to have changed, there are two essential changes to the `LAUNCH_UPDATE_DATA` command, preventing our attack. The first one implements the idea that we also proposed in Section VII-A: The HV must either pass a 4 kB or a 2 MB page to the command (however, 2MB pages are internally treated as multiple 4 kB pages). The second change is the calculation of the launch digest. The hash is now finalized after each call to the launch command and calculated as follows:

$$h_i = \mathrm{hash}(h_{i-1} \,\|\, \mathrm{hash}(\mathrm{data}_i) \,\|\, \mathrm{block\_size}_i \,\|\, \ldots \,\|\, \mathrm{GPA}_i),$$

where "…" represent additional fields that we omitted for brevity. Due to hash finalization after each call, the launch digest h_n now reflects the amount of `LAUNCH_UPDATE_DATA` calls used to load the data. In addition, the `GPA` field is of special interest, because it includes the memory layout, as it is observed by the guest, in the measurement. According to the documentation, the GPA value is computed by the SP and also stored in the RMP. Furthermore, the page is marked as a guest page. The GPA value is enforced, because the hardware page table walker checks that the GPA to HPA mapping in the NPT matches the one in the RMP.

VIII. Related Work

Since the initial release of AMD's memory encryption technology, first SME and later SEV, there has been a wide range of attacks against its security guarantees. Hetzelt et al. [22] exploited the unencrypted register state of the first version of SEV to construct simple encryption/decryption oracles. In addition they explored memory replay attacks.

Du et al. [18] unveiled the encryption mode, tweak values and the resulting lack of integrity protection of SME on Ryzen processors, which is closely related to SEV on EPYC processors. They used this knowledge in addition with a network service inside the VM to create an encryption oracle on a simulated version of SEV, which was not yet available.

Li et al. [28] used the lack of integrity protection combined with the knowledge of the tweak values to construct encryption as well as decryption oracles. For this, they exploited the fact that Direct Memory Access (DMA) operations issued by the VM are mediated by the HV through shared memory pages.

Wilke et al. [44] extended the analysis of the encryption mode and the tweak values to first generation EPYC and EPYC-embedded CPUs, unveiling an updated encryption mode. They showed how to abuse the missing integrity protection combined with knowledge of the tweak values, to bootstrap an encryption oracle from malicious code gadgets, solely by moving around ciphertext blocks in memory. However, the attack does not work on second generation EPYC CPUs, as these feature an enhanced randomization of tweak values. While our attack follows a similar approach of reordering memory blocks, it does not rely on the encryption mode at all, and is therefore also applicable to the currently available second generation EPYC CPUs.

Morbitzer et al. [34] leveraged the HV's control over the NPT as well as the page fault side channel to construct a decryption oracle. Similar to Du et al. [18], they require a service running in the VM. In their follow-up paper [33], they showed how to locate pages containing secrets, like OpenSSH keys, in the VM.

Werner et al. [43] showed that it is possible to use the unencrypted register values in the first SEV version to reconstruct the code executed in the VM. Furthermore, they showed how to use Instruction Based Sampling, a performance counter subsystem, to fingerprint code executed in SEV-ES VMs.

Radev et al. [37] described multiple attacks, exploiting insufficient value sanitization at the HV to VM boundary. For example, they showed how to trick the VM into treating arbitrary memory accesses as Memory Mapped I/O (MMIO), as well as into using malicious virtualized cryptographic accelerators provided by the HV. In addition, they demonstrated how faking `cpuid` results can be used to corrupt the VM page tables to mark all pages as unencrypted. They then used the unencrypted stack to launch a ROP attack, similar to our stage 2 gadget. However, in contrast to our attack, the page table manipulation used by them can be detected by a simple software countermeasure, as described in their paper.

Li et al. [29] demonstrated that the "security by crash" philosophy behind AMD's use of the Address Space Identifier (ASID) for mapping VMs to their memory encryption keys is flawed, as a malicious HV can swap the ASIDs of an attacker VM and the victim VM to leak limited amounts of data.

Buhren et al. [15] explored another attack vector by analyzing the firmware loading mechanism of the SP. They discovered a bug allowing them to load customized firmware on the SP, breaking the hardware root of trust.

IX. Conclusion

In this work, we have shown that the current attestation mechanism of SEV has a significant flaw, as it allows the HV to reorder blocks of the initially loaded image without influencing the launch measurement, leaving the guest owner unaware of our attack. We have been able to use this vulnerability to redirect execution and inject arbitrary code into the encrypted VM, giving us full control over its execution flow.

Moreover, we have shown how this vulnerability in the remote attestation allows us to extract secret data and conduct other attacks, like manipulating the booted operating system.

The attack described in this work undermines the validity of AMD SEV's remote attestation and thus its trustworthiness as a trusted execution environment. Especially, when authenticated and confidential execution in otherwise untrusted environments are required, additional means for verifying the authenticity of the loaded and executed software should be taken into consideration, until the vulnerability is fixed.

We have described possible changes to the firmware of the secure processor, allowing for a simple and reasonably secure mitigation which could be rolled out by means of a firmware update to existing EPYC processors of the first and second generation. However, as argued in Section VII, we do not think that it is possible to completely close this vulnerability with the capabilities of SEV-ES. If the information provided in the SEV-SNP white paper holds, full protection should only become available with the third generation EPYC processors.

References

[1] Advanced Micro Devices, "Github - AMDESE/linux," https://github.com/AMDESE/linux, on branch "sev-es-5.6-v3" at commit "d0d7b0f09359da7316aee077bac92ec21a1a047b".

[2] ——, "Github - AMDESE/qemu," https://github.com/AMDESE/qemu.git, branch "sev-es-v12", commit "1cfb0be5fdad55948f42f9056dfc16dc435099cf".

[3] ——, "AMD SEV-SNP: Strenghtening VM Isolation with Integrity Protection and More," https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, Advanced Micro Devices, Tech. Rep., Jan 2020.

[4] ——, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," https://www.amd.com/system/files/TechDocs/24593.pdf, October 2020, rev. 3.36.

[5] ——, "Secure Encrypted Virtualization API," https://www.amd.com/system/files/TechDocs/55766_SEV-KM_API_Specification.pdf, April 2020, rev. 3.24.

[6] ——, "SEV-ES Guest-Hypervisor Communication Block Standardization," https://developer.amd.com/wp-content/resources/56421.pdf, 2020.

[7] ——, "SEV Secure Nested Paging Firmware ABI Specification," https://www.amd.com/system/files/TechDocs/56860.pdf, August 2020, rev. 0.8.

[8] ——, "SEV firmware for Rome," https://developer.amd.com/wordpress/media/2013/12/amd_sev_fam17h_model3xh_0.24b0A.tar.gz, 2021, version 0.24b0A.

[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, 2013.

[10] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "SCONE: Secure Linux Containers with Intel SGX," in *OSDI*. USENIX Association, 2016, pp. 689–703.

[11] Asylo Project, https://github.com/google/asylo, accessed: 2021-01-26.

[12] A. Bennée, "Anatomy of a boot, a qemu perspective," https://www.qemu.org/2020/07/03/anatomy-of-a-boot/, accessed: 2021-01-29.

[13] J. Bottomley, "[edk2-devel] [patch v3 0/6] sev encrypted boot for ovmf," https://www.redhat.com/archives/edk2-devel-archive/2020-November/msg01247.html.

[14] ——, "Deploying encrypted images for confidential computing," https://blog.hansenpartnership.com/deploying-encrypted-images-for-confidential-computing, December 2020, accessed: 2021-01-22.

[15] 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.

[16] J. V. Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: hijacking transient execution through microarchitectural load value injection," in *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 54–72.

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

[18] 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.

[19] Enarx Project, https://github.com/enarx/enarx, accessed: 2021-01-26.

[20] GNU Project, "Gnu grub," https://www.gnu.org/software/grub/, accessed: 2021-01-22.

[21] Google, "Google cloud confidential computing / confidential vms," https://cloud.google.com/compute/confidential-vm/docs/about-cvm, accessed: 2021-01-26.

[22] 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. ACM, 2017, pp. 129–142. [Online]. Available: http://doi.acm.org/10.1145/3050748.3050763

[23] M. S. İnci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *Cryptographic Hardware and Embedded Systems – CHES 2016*. Berlin, Heidelberg: Springer, 2016, pp. 368–388.

[24] Intel, "Intel® trust domain extensions - white paper," https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf, 2020.

[25] Intel, "Intel®Software Guard Extensions (Intel®SGX) - Developer Guide," https://download.01.org/intel-sgx/sgx-linux/2.11/docs/Intel_SGX_Developer_Guide.pdf, August 2020, rev. 2.11.

[26] D. Kaplan, "Protecting VM Register State with SEV-ES," https://www.amd.com/system/files/TechDocs/ProtectingVMRegisterStatewithSEV-ES.pdf, Advanced Micro Devices, Tech. Rep., Feb. 2017.

[27] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," Advanced Micro Devices, Tech. Rep., 2016. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

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

[29] ——, "CROSSLINE: breaking "security-by-crash" based memory isolation in AMD SEV," *CoRR*, vol. abs/2008.00146, 2020. [Online]. Available: https://arxiv.org/abs/2008.00146

[30] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. IEEE Computer Society, 2015, pp. 605–622.

[31] Microsoft, "Azure confidential computing," https://docs.microsoft.com/en-us/azure/confidential-computing, accessed: 2021-01-26.

[32] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "Copycat: Controlled instruction-level attacks on enclaves for maximal key extraction," *arXiv preprint arXiv:2002.08437*, 2020.

[33] 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, CODASPY*. ACM, 2019, pp. 221–230.

[34] 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. ACM, 2018, pp. 1:1–1:6. [Online]. Available: http://doi.acm.org/10.1145/3193111.3193112

[35] QEMU Project, "Github - qemu," https://github.com/qemu/qemu, at commit "3dcfd4e3f285cd69d7cf581d3a688e421d28e07e".

[36] ——, "Qemu," https://www.qemu.org/, accessed: 2021-01-22.

[37] M. Radev and M. Morbitzer, "Exploiting interfaces of secure encrypted virtual machines," *CoRR*, vol. abs/2010.07094, 2020. [Online]. Available: https://arxiv.org/abs/2010.07094

[38] M. Russinovich, "Dcsv2-series vm now generally available from azure confidential computing," https://azure.microsoft.com/en-us/blog/dcsv2series-vm-now-generally-available-from-azure-confidential-computing/, April 2020.

[39] TianoCore, "Github - OVMF," https://github.com/tianocore/edk2.git, at commit "6c5801be6ef36e35f0b4ff906a4c99d68ca6f69a".

[40] TianoCore, "Ovmfpkg," https://github.com/tianocore/edk2/tree/master/OvmfPkg, accessed: 2021-01-21.

[41] Unified EFI Forum, "Unified extensible firmware interface (uefi) - specification," https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf, March 2019, version 2.8.

[42] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium. USENIX Association*, 2018.

[43] 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.

[44] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "Sevurity: No security without integrity : Breaking integrity-free memory encryption with minimal assumptions," in *2020 IEEE Symposium on Security and Privacy, SP 2020*. IEEE, 2020, pp. 1483–1496. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00080

[45] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and communications security*. ACM, 2012, pp. 305–316.