Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization

Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin, ***The Ohio State University;*** **Yan Solihin,** ***University of Central Florida***

https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization

Mengyuan Li
The Ohio State University

Yinqian Zhang
The Ohio State University

Zhiqiang Lin
The Ohio State University

Yan Solihin
University of Central Florida

Abstract

AMD's Secure Encrypted Virtualization (SEV) is an emerging technology to secure virtual machines (VM) even in the presence of malicious hypervisors. However, the lack of trust in the privileged software also introduces an assortment of new attack vectors to SEV-enabled VMs that were mostly unexplored in the literature. This paper studies the insecurity of SEV from the perspective of the unprotected I/O operations in the SEV-enabled VMs. The results are alerting: not only have we discovered attacks that breach the confidentiality and integrity of these I/O operations—which we find very difficult to mitigate by existing approaches—but more significantly we demonstrate the construction of two attack primitives against SEV's memory encryption schemes, namely a memory decryption oracle and a memory encryption oracle, which enables an adversary to decrypt and encrypt arbitrary messages using the memory encryption keys of the VMs. We evaluate the proposed attacks and discuss potential solutions to the underlying problems.

1 Introduction

Secure Encrypted Virtualization (SEV) is an emerging processor feature available in recent AMD processors that encrypts the entire memory of virtual machines (VM) transparently. Memory encryption is performed by a hardware memory encryption engine (MEE) embedded in the memory controller that encrypts memory traffic on the fly, with a key unique to each of the VMs. As the encryption keys are generated from random sources at the time of VM launches and are securely protected inside the secure processor in their lifetime, privileged software, including the hypervisor, is not able to extract the keys and use them to decrypt the VMs' memory content. Therefore, SEV enables a stronger threat model, where the hypervisor is removed from the trusted computing base (TCB). It is explicitly stated in AMD's SEV whitepaper [20] that "SEV technology is built around a threat model where an attacker is assumed to have access to not only execute user level privileged code on the target machine, but can potentially execute malware at the higher privileged hypervisor level as well." Hence, SEV provides a trusted execution environment (TEE) for (mostly) unmodified VMs to perform confidential computation that is shielded from strong adversaries that control the entire privileged software stack.

The lack of trust in the hypervisor, unfortunately, increases considerably the attack surface that a VM has to guard against. As the hypervisor controls the VMs' access to hardware resources including CPU, physical memory, I/O devices, the VM's CPU scheduling, memory management, and I/O operations must be mediated by untrusted software. As a result, new attack vectors have emerged as researchers explore the security properties of this new hardware feature. For instance, Hetzelt and Buhren [17] demonstrated attacks against SEV-enabled VMs by exploiting unencrypted virtual machine control block (VMCB) at the time of `VMExit`. They show that a malicious hypervisor may learn the machine state of the guest VM by reading register values stored in the VMCB and alter these register values before returning to the VM. The lack of integrity of the encrypted memory has been identified by several prior studies [9, 12, 17, 25], which enables a malicious hypervisor to perform a variety of attacks.

This paper studies a previously unexplored problem under SEV's trust model—the unprotected I/O operations of SEV-enabled VMs. While the entire memory of the VMs can be encrypted using keys that are not known to the hypervisor, direct memory access (DMA) from the virtualized I/O devices must operate on *unencrypted memory* or *memory shared with the hypervisor*. As a result, neither the confidentiality nor the integrity of the I/O operations can be guaranteed under SEV's trust models.

More importantly, this paper goes beyond the investigation of I/O insecurity itself. In particular, we further demonstrate that these unprotected I/O operations can be leveraged by the adversary to construct (1) an encryption oracle to encrypt arbitrary memory blocks using the guest VM's memory encryption key, and (2) a decryption oracle to decrypt any memory pages of the guest. We demonstrate in the paper that these

two powerful attack primitives can be constructed in a very stealthy manner—the oracles can be queried by the adversary repeatedly and frequently without crashing the attacked VMs.

In addition, as a by-product of the study, this paper also reveals a severe side-channel vulnerability of SEV: As the adversary is able to manipulate the nested page tables, it could alter the `present` bit or `reserve` bit of the nested page table entries to force guest VM's memory accesses to the corresponding pages to trigger page faults. While this page-fault side channel has been previously studied in the context of Intel SGX [36] and even used in previous attacks against SEV [17], it is also reported that page faults from SEV-enabled guest VMs leak the entire faulting addresses (and error code) to the hypervisor (unlike in SGX where the page offset is masked). This fine-grained page-fault attack enables fine-grained tracing of the encrypted VM's memory access patterns, and particularly in this paper is used to facilitate the construction of the memory decryption oracle.

Contribution. This paper contributes to the study of TEE security in the following aspects:

- The paper studies a previously unexplored security issue of AMD SEV—the unprotected I/O operations of SEV-enabled guest VMs. The root cause of the problem is the incompatibility between AMD-V's I/O virtualization with SEV's memory encryption scheme.
- The paper demonstrates that the unprotected I/O operations could also be exploited to construct powerful attack primitives, enabling the adversary to perform arbitrary memory encryption and decryption.
- The paper reports the lack of page-offset masking of the faulting addresses during SEV's page faults handling, which leads to fine-grained side-channel leakage. The paper also demonstrates the use of both fine-grained and coarse-grained side channels in its I/O attacks.
- The paper empirically evaluates the fidelity of the attacks and discusses both hardware and software approaches to mitigating the I/O security issues.

Responsible disclosure. We have reported our findings to AMD and disclosed the technical details with AMD researchers. While we were confirmed that the presented attacks work on current release of SEV processors, AMD researchers suggested future generations of SEV chipsets are likely to be immune from these attacks. Some of the technical feedback we obtained from AMD has been integrated into the paper.

Roadmap. Section 2 presents the overview of AMD's SEV and explains the root causes of the exploited I/O operations in this paper. Section 3 describes several attacks exploiting the unprotected I/Os. Section 4 presents an evaluation of the fidelity of the attacks. Section 5 discusses potential solutions to securing SEV's I/O operations. Section 6 summarizes related work and Section 7 concludes the paper.

2 Overview of AMD SEV

2.1 Overview

AMD Secure Encrypted Virtualization (SEV) is a security extension for AMD Virtualization (AMD-V) architecture [2]. AMD-V is designed as a virtualization substrate for cloud computing services, which allows one physical server to run multiple isolated guest virtual machines (VM) concurrently. AMD's SEV is designed atop its Secure Memory Encryption (SME) technology.

Secure Memory Encryption. SME [20] is AMD's technology for real-time memory encryption, which aims at providing strong protection against memory snooping and cold boot attacks. An Advanced Encryption Standard (AES) engine is embedded in the on-die memory controller that encrypts/decrypts memory traffic when it is transferred out of or into the processor. A single ephemeral encryption key is generated for the entire machine from a random source every time system resets. The key is managed by a 32-bit ARM Cortex-A5 Secure Processor (AMD-SP). When SME is enabled, physical address bit 47 (also called the *C-bit*) in the page table entry (PTE) is used to mark whether the memory page is encrypted, thus enabling page-granularity encryption. Transparent SME, or TSME, is a mode of operating of SME, which allows encryption of the entire memory regardless of the C-bit. Thus TSME allows unmodified operating system to use the memory encryption technology.

Secure Encrypted Virtualization. AMD's SEV combines the AMD-V architecture and the SME technology to support encrypted virtual machines [20]. SEV aims to protect the security of guest VMs even in the presence of a malicious hypervisor, by using two isolation techniques: First, the data of guest VMs inside the processor is protected by an access control mechanism using Address Space Identifier (ASID). Specifically, the data in the CPU cache is tagged with ASID of each VM; thus it is prevented from being accessed by other VMs or the hypervisor. Second, the guest VM's data outside the processor (*e.g.*, in the DRAM) is protected via memory encryption. Rather than using a single AES key for the whole machine as in the case of SME, SEV allows each VM to use a distinct ephemeral key, thus preventing the hypervisor from reading the encrypted memory of each VM. Because memory encryption keys are managed by the secure co-processor, privileged software layers are not allowed to access or manipulate these keys.

Beside confidentiality, authenticity of the platform and integrity of the guest VMs are also provided by SEV. An identification key embedded in the firmware is signed by both AMD and the owner of the machine to demonstrate that the platform is an authentic AMD platform with SEV capabilities, which is administered by the machine owner. The initial contents of memory, along with a set of metadata of the VM, can be signed by the firmware so that the users of the guest VMs

Table 1: Effects of C-bits in guest page tables (gPT) and nested page tables (nPT). M is the plaintext; $E_k()$ is the encryption function under a memory encryption key k; k_g and k_h represent the guest VM and the hypervisor's memory encryption keys, respectively.

gPT	nPT	
	C-bit=0	C-bit=1
C-bit=0	M	$E_{k_h}(M)$
C-bit=1	$E_{k_g}(M)$	$E_{k_g}(M)$

may verify the identity and the initial states of the launched VMs through remote attestation.

2.2 Memory Encryption

ASID and memory encryption. The encryption keys used for memory encryption are generated from random sources when the VMs are launched. They are securely stored inside the secure processor for their entire life-cycle. Each VM has its own unique memory encryption key K_{vek}, which is indexed by the ASID of the VM. When the VM accesses a memory page that is mapped to its address space with its C-bit set, the memory will be first decrypted using the VM's K_{vek} before loaded into the CPU caches. Data in the caches are stored in plaintext; each cache line, in addition to the regular cache tags, is also tagged by the ASID of the VM. As such, the same physical memory may have multiple copies cached in the hardware caches. AMD does not maintain the consistency of the cache copies with different ASID tags [20].

Encryption with nested paging. AMD-V utilizes nested paging structures [1] to facilitate memory isolation between guest VMs. When the virtual address used by the guest VM (gVA) is to be translated into physical address, it is first translated into a guest physical addressing (gPA) using the guest page table (gPT), and the gPA is then translated into the host physical address (hPA) using the nested page table (nPT). While gPT is located in guest VM's address space, nPT is controlled directly by the host.

With AMD's SME technology, bit 47 of a PTE is called the *C-bit*, which is used to indicate whether or not the corresponding page is encrypted. When the C-bit of a page is set (*i.e.*, 1), the page is encrypted. As both the gPT and the nPT has C-bits, the encryption state of a page is controlled by the combination of the two C-bits in its PTEs in the gPT and nPT. The effect of C-bits in the gPT and nPT is shown in Table 1. To summarize, whenever the C-bit of gPT is set to 1, the memory page is encrypted with the guest VM's encryption key k_g; when the C-bit of gPT is cleared, the C-bit of nPT determines the encryption state of the page: the page is encrypted under the hypervisor's key k_g when C-bit is 1; otherwise the page is not encrypted.

To share memory pages between a guest VM and the hypervisor while preventing physical attacks, it is required to have the memory page's C-bit set to 0 in its gPT and the C-bit set to 1 in its nPT, so that the page is encrypted under the hypervisor's encryption key.

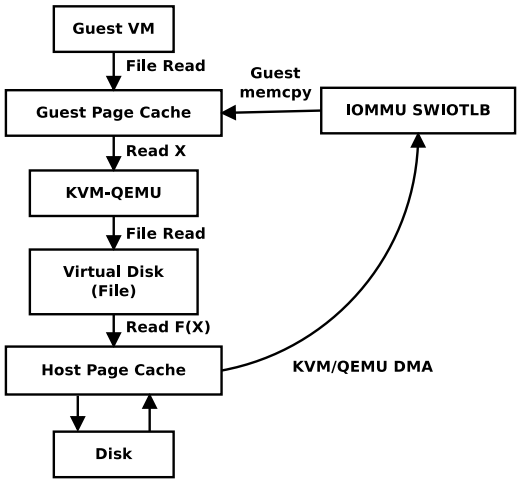


Figure 1: An example of a disk I/O operation by an SEV-enabled VM.

Encryption modes of operation. SEV uses AES as its encryption algorithm. The memory encryption engine encrypts data with a 128-bit key using the Electronic Codebook (ECB) mode of operation [12]. Therefore, each 16-byte aligned memory block is encrypted independently. A physical address-based tweak function $T()$ is utilized to make the ciphertext dependent of not only the plaintext but also its physical address [20]. Specifically, the tweak function is defined as $T(x) = \oplus_{x_i=1} t_i$, where x_i is the ith bit of host physical address x, $\oplus$ is the bitwise exclusive-or (*i.e.*, XOR) and t_i ($1 \leq i \leq 128$) is a 128-bit constant vector. The tweak function takes a physical address as an input and outputs a 128-bit value $T(x)$. Therefore, the ciphertext c of a plaintext m at address P_m is $c = E_K(m \oplus T(P_m))$. The tweak function prevents attacker from inferring plaintext by comparing the ciphertext of two 16-byte memory blocks. However, as the constant vectors t_is remain the same for all VMs (and the hypervisor) on the machine, they can be easily reverse engineered by an adversary.

Known issues with SME memory encryption. One root problem exploited in prior studies on SEV's insecurity is the lack of integrity of its encrypted memory [9, 12, 17, 25].

2.3 Virtualized I/O Operations

Similar to other virtualization technologies, SEV-enabled VMs interact with I/O devices through virtual hardware using Quick Emulator (QEMU). Common methods for VMs to perform I/O operations are programmed I/O, memory-mapped I/O, and direct memory access (DMA). Among these methods, DMA is most frequently used method for SEV-enabled VMs to do I/O accesses.

Direct Memory Access in SEV. With the assistance of DMA chips, programmable peripheral devices can transfer data to and from the main memory without involving the processor. With virtualization, a common way to support DMA is through IOMMU, which is a hardware memory management unit that maps the DMA-capable I/O buses to the main memory. However, unique to SEV is that the memory is encrypted. While the MMU supports memory encryption with multiple ASIDs, IOMMU only supports one ASID (*i.e.*, ASID=0). Therefore, in SEV-enabled VMs, DMA operations are performed on memory pages that are shared between the guest and the hypervisor (encrypted with the hypervisor's K_h). A bounce buffer, called Software I/O Translation Buffer (`SWIOTLB`), is allocated on these memory pages.

To illustrate the DMA operation from the guest, a disk I/O read is shown in Figure 1. When a guest application needs to read data from file, it first checks whether the file is already stored in its page cache. A miss in the guest page cache will trigger read from virtual disks, which is emulated by QEMU-KVM. The data is actually read from the physical disk by QEMU-KVM's DMA operation into `SWIOTLB` and then copied to the disk device driver's I/O buffer by the guest VM itself. The disk write operation is the inverse of this process, in which the data is first copied from the guest into `SWIOTLB` and then processed by QEMU.

3 Security Issues

In this section, we explore the security issues of the lack of protection for SEV's I/O operations. We start of our exploration with the most straightforward consequence of vulnerability—the insecurity of I/O operations itself—and present an attack example that breaches the integrity of I/O operations. To comprehensively study the attack surface, we also enumerate the I/O operations from a guest VM that are vulnerable to such attacks and discuss the challenges of implementing effective countermeasures. Next, we show that I/O insecurity leads to a complete compromise of the memory encryption scheme of SEV, by constructing powerful attack primitives that leverage the unprotected I/O operations to enable the adversary to encrypt or decrypt arbitrary messages with the guest VM's memory encryption key, k_{vek}.

3.1 Threat Model

We consider a scenario in which the VMs' memory are encrypted and protected by AMD SEV technology. The hypervisor run on a machine controlled by a third-party service provider. Under the threat model we consider, the third-party service is not trusted to respect the integrity or confidentiality of the computation inside the VMs. This could happen when the service provider is dishonest or when the hypervisor has been compromised.

The goal of the attacks is either to compromise the I/O operations themselves or the memory encryption of SEV. Out of scope in this paper are denial-of-service (DoS) attacks, in which the service provider simply refuses to run the VM. SEV is not designed to prevent DoS attacks.

3.2 I/O Security

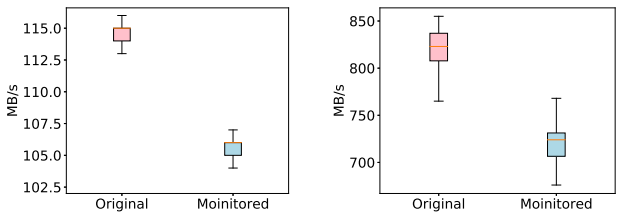
In this section, we explore the direct consequences of unprotected I/O operations from SEV-enabled guests.

3.2.1 Case Study: Integrity Breaches of Disk I/O

We first present a case study to show how SEV's guest VMs' unprotected I/O operations can be exploited to breach I/O security in practice. In this case study, we show that a malicious hypervisor is able to gain control of the guest VMs through an OpenSSH server without passwords by exploiting unprotected disk I/O. Therefore, we assume the disk is *not* encrypted with disk encryption key in this example. However, we note it is recommended by AMD to only use encrypted storage. As such, this case study only serves the purpose of proof-of-concept, rather than a practical attack. We will discuss its security implications in Section 3.2.2.

Specifically, the adversary controls the entire host and launches the SEV-enabled VM using the standard procedure [3]. During the system bootup, the binary code of `sshd` that performs user authentication is loaded into the memory. To monitor the disk I/O streams, whenever the QEMU performs a DMA operation for the guest, the adversary checks the memory buffer used for this DMA operation (*i.e.*, `SWIOTLB`) and search for the binary code of `sshd`. In our implementation, we used a 32-byte memory content (*i.e.*, 0*x*ff85 0*x*c041 0*x*89c4 0*x*8905 0*x*4e05 0*x*2900 0*x*0f85 0*x*1b01 0*x*0000 0*x*488b 0*x*3d49 0*x*0529 0*x*0089 0*x*eee8 0*x*c2bf 0*x*fdff) as the signature of the `sshd` binary and no false detection was observed. Once the DMA operation for `sshd` is identified, the adversary modifies the binary code inside `SWIOTLB`, before the QEMU commits the DMA operation. In particular, this is done by replacing the crucial code used in authentication that corresponds to `callq pam_authenticate`, which is a five-byte binary string 0*x*e8 0*x*c2 0*x*bf 0*x*fd 0*x*ff, to `mov $0 %eax` (a binary string of 0*x*b8 0*x*00 0*x*00 0*x*00 0*x*00). `pam_authenticate()` is used to perform user authentication; only when it returns 0 will the authentication succeed. Therefore, by moving 0 to the register `%eax` (the register used to store return value of a function call) directly, the adversary can successfully bypass the user authentication without knowing the password. To validate the attack, we empirically conducted the attack three times and all were successful.

Performance degradation due to I/O monitoring. We also conducted experiments to measure the performance degradation due to the hypervisor's monitoring of disk I/O streams. We used the `dd` command to write 1GB of data to the local



(a) I/O write (b) I/O read

Figure 2: Read/write performance overhead due to I/O monitoring.

disk to measure the I/O write speed. The `dsync` flag of set to make sure the data is written to the disk directly, bypassing the page caches. To measure the read speed, we cleaned the page caches in the memory by setting `vm.drop_caches=3` before reading 1GB of data from local disk. In both the read and write experiments, we measured the performance with and without I/O stream monitoring and repeated the measurements 200 times. The results show the performance degradation of I/O read and write is 11.8% and 7.9% respectively (see Figure 2).

3.2.2 Estimating The Attack Surface

As shown in the above example, I/O operations that are not encrypted by the software can be intercepted by the malicious hypervisor and manipulated to compromise the SEV-enabled guests. This vulnerability exists in all emulated I/O devices that are commonly used in cloud VMs, such as disk I/O, network I/O, and display I/O, *etc*. While a straightforward solution is to encrypt I/O streams by software, however, this simple method has many practical limitations in practice:

Network I/O. Network traffic can only be partially encrypted, as headers of IP or TCP cannot be encrypted. The adversary is still able to modify the network traffic to forge the IP addresses, port numbers, and encrypted metadata of the network packets. This is true for both TLS traffic and VPN traffic. As we will show in Section 3.3, encrypted traffic like SSH can still be exploited to construct memory decryption oracles.

Display I/O. Encrypting I/O traffic cannot be applied when the I/O devices cannot decrypt the I/O stream by themselves. Display I/O is one such example. For instance, Virtual Network Computing (VNC) is a graphical desktop sharing protocol that allows VMs to be remotely controlled. In KVM, the QEMU redirects the VGA display from the guest to the VNC protocol, which is not encrypted. Therefore, if the user of the guest VM uses VNC to control the VM, keystroke and mouse clicking will be learned and manipulated by the adversary. To protect display I/O operations, the guest VM must be modified to encrypt all display I/O traffic and the remote user interface must be modified accordingly to decrypt the traffic.

Disk I/O. For disk I/O operations, the method recommended by SEV [4] is for each SEV-enabled VMs to use encrypted disk filesystems. To use encrypted disks, however, the owner needs to first provision the disk encryption key into the protected VMs by using the `Launch_Secret` [3] command. This command first decrypts a packet sent by the VM owner (that contains the disk encryption key) encrypted using K_{tek} (Transport Encryption Key), atomically re-encrypts it using the memory encryption K_{vek}, and then injects it into the guest physical address specified by `GUEST_PADDR` (a parameter of the `Launch_Secret` command). As the address of the disk encryption key is known, if memory confidentiality is compromised (using methods to be described in Section 3.3), the disk encryption key can be learned and used to decrypt the entire image. Therefore, disk I/O is not secure, either.

3.3 Decryption Oracles

In this section, we show that the DMA operations under SEV's memory encryption technology can be exploited to construct a decryption oracle, which allows the adversary to decrypt any memory block encrypted with the guest VMs' memory encryption key K_{vek}. The oracle can be frequently and repeatedly queried and thus can be exploited as an attack primitive for more advanced attacks against SEV-enabled guests.

As mentioned in Section 2.3, the DMA operation from the SEV-enabled VM is conducted with the help of memory pages shared with the hypervisor. When DMA operates in the `DMA_TO_DEVICE` mode, data is transferred by the IOMMU hardware to the shared memory, and then copied by CPU in the SEV-enabled VM to its private memory; when DMA operates in the `DMA_BIDIRECTIONAL` mode, the SEV-enabled VM first copies the data from encrypted memory to the shared memory, and then the DMA reads or writes are performed on the shared memory.

Both these modes of operations provide the adversary an opportunity to observe the transfer of data blocks from memory pages encrypted by K_{vek} to memory pages that is not encrypted (from the hypervisor's perspective). Therefore, if the adversary alters the ciphertext of the data blocks in the encrypted memory page before they are copied by the guest VM, after the memory copy, the corresponding plaintext can be learned from the shared memory directly.

The construction of such a decryption oracle is shown in Figure 3. The decryption oracle can be constructed in three steps: *pattern matching*, *ciphertext replacement*, and *packets recovery*. We use network I/O as an example. The adversary exploits the network traffic in Secure Shell (SSH) to construct the decryption oracle. But we stress that any I/O traffic can be exploited in similar manners. In the following experiments, we configured the guest VM to use `OpenSSH_7.6p1` with `OpenSSL` 1.0.2n, which is default on Ubuntu 18.04.

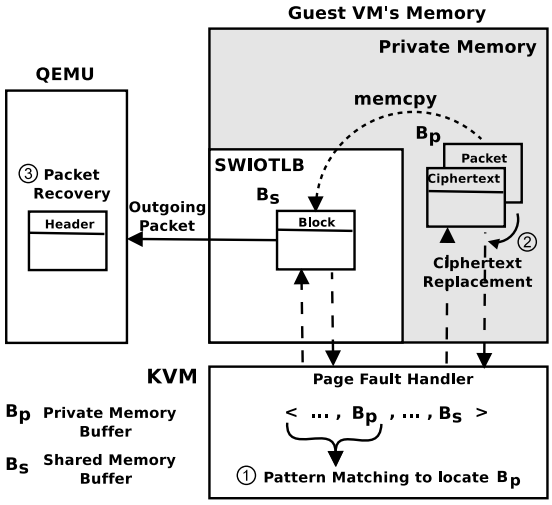


Figure 3: A decryption oracle. Step ①, the hypervisor conducts pattern matching using page-fault side channels to determine the address of B_p. Step ②, the hypervisor replaces a ciphertext block in B_p with the target memory block, which will be decrypted when copied to B_s. Step ③, QEMU recovers the network packet headers.

3.3.1 SSH and Network Stacks

To control the SEV-enabled guest remotely, the owner of the VM typically uses SSH protocol to remotely login into the VM and controls its activities. To copy data to and from the VM, protocols like SCP, which is built on top of SSH, is commonly used. Particularly, we consider the SSH traffic after the remote owner has already authenticated with `sshd` and a secure communication channel has been established. Because the SSH handshake protocol is performed in plaintext, the adversary who controls the hypervisor and QEMU can act as a man-in-the-middle attacker and recognize the established the secure channel by its IP addresses and TCP port number. Once the secure channel is established, SSH command and output data will be transferred using encrypted SSH packets that are transmitted in interactive mode [31].

In the interactive mode, each individual keystroke guest owner types will generate a packet that is sent to the SEV-enabled VM, which will be transferred by DMA to a memory buffer shared between the guest and the hypervisor. The packet is then copied by the guest to a private memory page encrypted using K_{vek}. Then the data is handled by the network stack in the guest OS kernel. The headers of the packet are then removed and the payload data is forwarded to the user-space application. Then the SSH server processes the keystroke and responds with an acknowledgement packet. The acknowledgement packet is copied back to the kernel space, wrapped by the corresponding header information, and then copied to the shared memory buffer. The last memory copying also decrypts the memory using the guest VM's K_{vek}. Therefore, our attack primitives target this process. As a result, every network packet generated by the guest VM can be exploited as a decryption oracle that helps the adversary decrypt one or multiple memory blocks.

3.3.2 Pattern Matching Using Fine-grained Page-fault Side Channels

Let us denote the private memory buffer as B_p, whose gPA is P_{priv}, and the shared memory buffer as B_s, whose gPA is P_{share}. The primary challenge in this attack is to identify the P_{priv}. As this address is never directly leaked, the adversary needs to perform a page-fault side-channel analysis.

Fine-grained page-fault side channels in SEV. The page fault side channel was first studied by Xu *et al.* in the context of Intel SGX [36]. As an SGX attacker controls the entire operating system, he or she can manipulate the page table entries (PTE) and set the `present` bit of the PTEs of pages that are mapped to the targeted enclave. By doing so, once the enclave program accesses the corresponding memory pages, the control flow will be trapped into the OS kernel through a page fault exception. On x86 processors, the faulting address will be stored in a control register, `CR2` so that the page-fault handler could learn the entire faulting address. To provide secrecy, SGX masks the page offset of the faulting address and leaves only the virtual page number in `CR2`.

Similarly, on the AMD platform, the adversary that compromises the hypervisor could also exploit the page-fault side channels to track the execution of the SEV-enabled VMs. Although the mapping between the guest VM's guest virtual address (gVA) to gPA is maintained by the guest VM's page table and is encrypted by K_{vek}, the hypervisor could manipulate the nested page tables (NPT) to trap the translation from gPAs to host physical addresss (hPA). Unlike SGX, SEV does not mask the page offset, providing more fine-grained observation to the adversary.

Moreover, the page-fault error code returned in the `EXITINTINFO` field of VMCB can also be exploited in the SEV page-fault side-channel analysis. Specifically, the page-fault error code is a 5-bit value, revealing the information of the page fault. For example, when bit 0 is cleared, the page fault is caused by non-present pages; when bit 1 is set, the page fault is caused by a memory write; when bit 2 is cleared, the page fault takes place in the kernel mode; when bit 3 is set, the fault is generated form a `reserved` bit; when bit 4 is set, the fault is generated by an instruction fetch. The error code provides detailed information regarding the reasons of the page fault, which can be leveraged in side-channel analysis.

Pattern matching. With such a fine-grained side channel, the adversary could monitor the memory access pattern of the guest when it receives an SSH packet. Particularly, after delivering an SSH packet to the SEV-enabled VM, the adversary immediately initiates the monitoring process and marks all of

the guest VM's memory pages inaccessible by clearing the `present` bit of the PTEs. Every time a memory page is accessed by the guest, a page fault takes place and the adversary is able to learn the entire faulting address P_i. Note here the faulting address in the guest VM refers to the guest physical address as the guest virtual address is not observable by the hypervisor. After the page fault, the adversary resets the `present` bit in the PTE to allow future accesses to the page. Therefore, with the fine-grained page fault side channel, one only needs to collect information regarding the first access to a memory page. The monitoring procedure stops when the acknowledgement packet is copied into B_s. At this point, the adversary has collected a sequence of faulting addresses $<P_1, P_2, \cdots, P_m>$.

Internally in the guest VM, when *sshd* is sending a packet, the encrypted data is first copied to the buffer of the transport layer, then the buffer of the network layer, and then the buffer of the data link layer. In each layer, new packet headers are added. Eventually, the entire network packet is stored in a data structure called `sk_buff`. Finally, the kernel will call `dev->hard_start_xmit` to transfer the data in `sk_buff` to the device driver, where B_p is located.

Both P_{priv} and the address of `sk_buff`, P_{sk}, should be found in the faulting addresses sequence $<P_1, P_2, \cdots, P_m>$. It is because the memory pages that store the private memory buffer B_p and `sk_buff` are not otherwise used during the process of sending network packets. The adversary could combine page offsets, page frame numbers, the page-fault error code, and the number of page faults between the two page faults of B_p and `sk_buff` to create a signature, which can be used to find P_{priv}. For example, the page-fault error code of B_p is $0b110$ and the page-fault error code of `sk_buff` is $0b100$; the page offset of P_{priv} is usually $0x0$fa or $0x8$fa and the offset of `sk_buff` usually ends with $0x$e8 or $0x00$; and the number of page faults between B_p and `sk_buff` is roughly 20. With these signatures, the adversary can identify P_{priv} from the sequence of faulting addresses. Of course, the signature may change from one OS version to another, or change with different OS kernel. However, because the adversary controls the hypervisor, such information can be re-trained offline, before performing the attacks.

It was indicated by AMD researchers (during an offline discussion) that SEV-ES should mask the page offset information when there is a VMEXIT. However, we were not able to find related public documentation. Moreover, as the KVM patch for SEV-ES support is not yet available at the time of writing, we were not able to validate the claim or estimate the remaining leakage (*e.g.*, error code, page offset) after the patch. However, regardless of the hardware changes, a coarse-grained page-fault side channel in which the page frame number of the faulting address is leaked must remain. To show that the demonstrated attack still works, we conducted experiments to perform pattern matching without page fault offsets and error code information. Specifically, we performed pattern matching using only the faulting page numbers, with the guest VM running different Ubuntu versions (*e.g.*, 18.04, 18.04.1 and 19.04) and different kernel versions (4.15.0-20-generic, 4.15.0-48-generic and 5.0.0-13-generic). The results show that after training in one virtual machine, the pattern matching rules can work well even in different virtual machines with the same Ubuntu version and kernel version—the attacker is still able to successfully identify the page frame number of P_{priv}. To determine the complete address of P_{priv}, the attacker could determine the offset by scanning the entire memory page and looking for content changes (*e.g.*, in a 90-byte buffer).

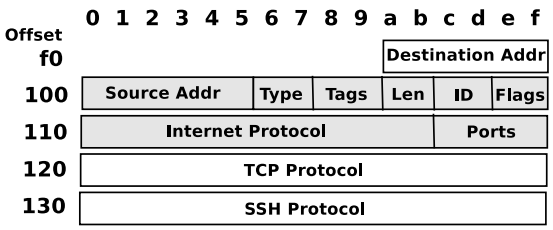


Figure 4: Format of an SSH packet.

3.3.3 Replacing Ciphertext

After determining P_{priv}, the adversary replaces aligned SSH header in B_p with the ciphertext he or she chooses to decrypt. As shown in Figure 4, the packet headers include a 6-byte destination address, a 6-byte source address, a 2-byte IP type (*e.g.*, IPv4 or IPv6), 1-byte IP version and IP header length, 1-byte of differentiated services field, 2-byte packet length, 2-byte identification, 2-byte of IP flags, 1-byte time-to-live, 1-byte protocol type, 2-byte checksum, and 4-byte source IP address and 4-byte destination address, and 20-bytes TCP headers (start with 2-byte source port and 2-byte destination port).

As shown in Figure 4, P_{priv} has the offset address ending with $0x$fa. Because SEV encrypts data in 16-byte aligned blocks, only part of the TCP/IP header (i.e., header in gray blocks in Figure 4) can be used to decrypt ciphertext. Additional constraints apply if the packet needs to be recovered later. Before replacing the packet header with the chosen ciphertext, the adversary performs a `WBINVD` instruction to flush the guest VM's cached copy of B_p back to memory. It is because cache coherence is not maintained by the hardware between cache lines with different ASIDs. To make sure the guest VM's copy does not overwrite our changes to the memory, `WBINVD` instruction needs to be called first.

The ciphertext replacement takes place before `memcpy`, after B_p is accessed and before B_s is accessed. B_s is located inside the `SWIOTLB` pool, which is the next available address within `SWIOTLB` that can be used by the guest. After replacing a few blocks in B_p, another `WBINVD` instruction is performed to ensure the guest VM reads and decrypts up-to-date cipher-

text in memory. All replacement operation is achieved by `IOremap` instead of `Kmap`, since `Kmap` decrypts data with the hypervisor's key first and `IOremap` directly operates data in the memory without decryption.

We use the following example to illustrate the attack. Let the ciphertext c be a 16-byte aligned memory block with the gPA of P_c. The function which can translate gPA to hPA is called $hPA()$. The goal of the attack is to decrypt c. The adversary replaces a 16-byte data in the SSH header that begins with address $(P_{priv}+16)/16*16$ with c. After the data in B_p is copied to B_s, the adversary could read the decrypted SSH packet and extract the plaintext of decrypted memory block, d, from the corresponding location of the packet. However, d is not the plaintext of c yet, as SEV's memory encryption involves a tweak function $T()$. That is, $c = E_{K_{vek}}(m \oplus T(hPA(P_c)))$ but $d = D_{K_{vek}}(c) \oplus T(hPA((P_{priv}+16))/16*16)$. Therefore, the plaintext message m of ciphertext c can be calculated by $m = d \oplus T(hPA((P_{priv}+16))/16*16) \oplus T(hPA(P_c))$.

3.3.4 Packets Recovery

To make the attack stealthy, the adversary needs to recover the network packet with decrypted data before those packets are passed to the physical NIC device. As shown in Figure 4, the SSH header also contains metadata of the packet. When the malicious hypervisor injects chosen ciphertext into the memory block with offset = $0x100$, the adversary only needs to be concerned about a portion of the source IP address, IP protocol type, IP tags, TCP header length, and the identification of the packet. Majority of the fields are determined. The identification of the IP packet increases by 1 every time SSH server replies a packet. So when hypervisor tries to recovery the (plaintext) packet from the QEMU side, it only need to correct the packet length, increase identification by 1 and copy the remaining portion from previous packet such as source address, header length, time to live and protocol number.

3.4 Encryption Oracle

We next show the construction of a memory encryption oracle using unprotected I/O operations. The encryption oracle stealthily encrypts a chosen plaintext message using a guest VM's memory encryption key K_{vek}. Similar to the construction of the decryption oracle, during the DMA operation of the guest that transfers data from the device to the encrypted memory, the adversary changes the message m in the shared memory buffer B_s, waits until it is copied to the private buffer B_p in the encrypted page, and then extracts the corresponding ciphertext $E_{k_g}(m)$ from B_p.

To determine the gPA address of B_p and retrieve the ciphertext of the plaintext message at address P_t, the steps shown in Figure 5 are taken. Again, we leverage the fine-grained page-fault side channel we used in the previous section. Specifically,

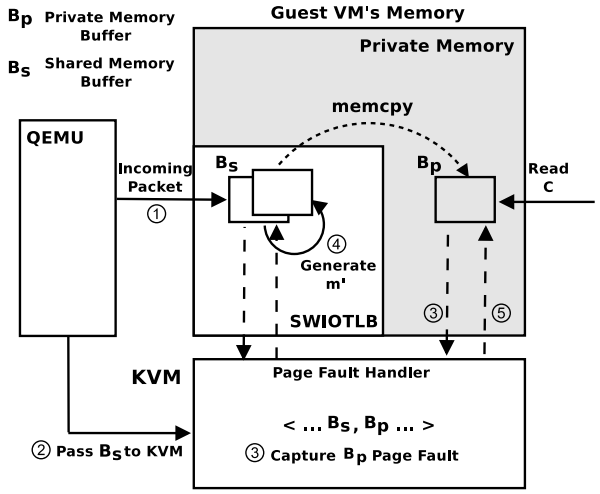


Figure 5: An encryption oracle. Step ①, QEMU forwards an incoming packet to the guest. Step ②, QEMU passes the address of B_s to the hypervisor. Step ③, a page fault immediately after the fault at B_s is captured by the page fault handler. Step ④, message m' is placed in B_p. Step ⑤, page fault handler returns the control to the guest.

we modified all memory pages' PTEs right after the QEMU finishes writing the packet into `SWIOTLB` and before the QEMU notifying guest VM about the DMA write. Then, when the guest VM performs a `memcpy` operation to copy the data, the adversary will observe a sequence of page faults: <...P_{share}, P_{priv}...>, where P_{share} is the address of B_s and P_{priv} is the address of B_p. The page fault at P_{priv} will take place right after the page fault at P_{share}. When the hypervisor handles the page fault at P_{priv}, it replaces the 16-byte aligned data block with the message m', where $m' = m \oplus T(hPA(P_{priv})) \oplus T(hPA(P_t))$, where P_t is the gPA of the target address to which the adversary wishes to copy m. The corresponding ciphertext will be $c = E_{k_g}(m \oplus T(hPA(P_t)))$, which can be used to replace the ciphertext at address P_t.

The encryption oracle can be typically exploited to inject code or data into the SEV-enabled VM's encrypted memory, or it can be used to make guesses of the memory content by providing a probable plaintext. We note that to use the encryption oracle, the adversary may simply generate meaningless packets and send them to the guest VM, which will be discarded. But the oracle can still be constructed and used. The only downside of this approach is that the guest VM will observe large volume of meaningless network traffic and may become suspicious of attacks.

4 Evaluation

We implemented our attacks on a blade server with an 8-Core AMD EPYC 7251 Processor, which has SEV enabled on the chipset. The host OS runs Ubuntu 64-bit 18.04 with

Linux kernel v4.17 (KVM hardware-assisted virtualization supported since v4.16) and the guest OS also runs Ubuntu 64-bit 18.04 with Linux kernel v4.15 (SEV supported since v4.15). The QEMU version used was QEMU 2.12. The SEV-enabled guest VMs were configured with 1 virtual CPU, 30GB disk storage, and 2GB DRAM. The OpenSSH server was installed from the default package archives.

4.1 Pattern Matching

We first evaluate the pattern matching algorithm's accuracy of determining P_{priv}. To obtain the ground truth, we modified the guest kernel to log the gPA address of `sk_buff`, the source gPA and destination gPA of `memcpy`, as well as the size of each DMA read or write. All the data was recorded in the kernel debug information, which can be retrieved using a Linux command `dmesg`.

The experiments were conducted as follows: We ran a software program *AnJian* [13] (an automated keystroke generation tool) on a remote machine, which opened a terminal that was remotely connected to the SEV-enable VM through an SSH communication channel. *AnJian* automatically typed on the SSH terminal two Linux commands `cat security.txt |grep sev` and `dmesg` at the rate of 10 keystrokes per second. This was used to simulate the remote owner controlling the SEV-enabled VM through SSH. The adversary would make use of the generated SSH packets to perform memory decryption. The `dmesg` command also retrieved the kernel debug message that recorded the ground truth.

At the same time, the pattern matching was performed by the adversary on the hypervisor side. The page-fault side-channel analysis was conducted upon receiving every incoming SSH packet to guess the address P_{priv}. There were three outcomes of the guesses: a correct guess, an incorrect guess, and unable to make a guess. Because there were 33 keystrokes generated by *AnJian*, the adversary was allowed to guess P_{priv} for 33 times in each experiment. The experiments were conducted 20 times.

Figure 6 shows the precision and recall of these 20 rounds of experiments. Precision is defined as the ratio of the number of correct guesses and the number of times that a guess can be made. Recall is defined as the ratio of the number of correct guesses and the number of total SSH packets. The average precision is 0.956, the average recall is 0.847 and the average F_1 Score is 0.897.

4.2 Persistent B_p

According to our experiments, the B_p will remain unchanged and reused for multiple network packets. This greatly helps the adversary, either by performing pattern matching once and reusing the same B_p directly in subsequent packets, or by improving the accuracy of the guesses.

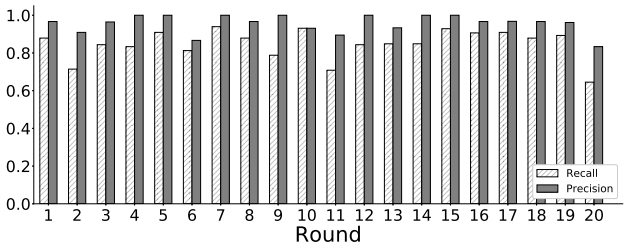


Figure 6: The precision and recall of determining P_{priv} in 20 rounds or experiments.

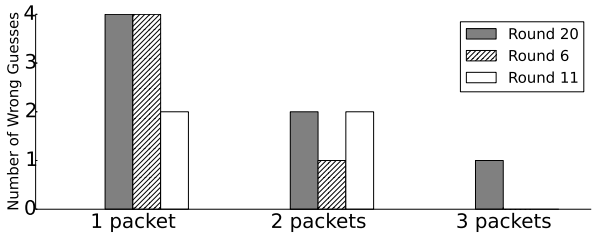


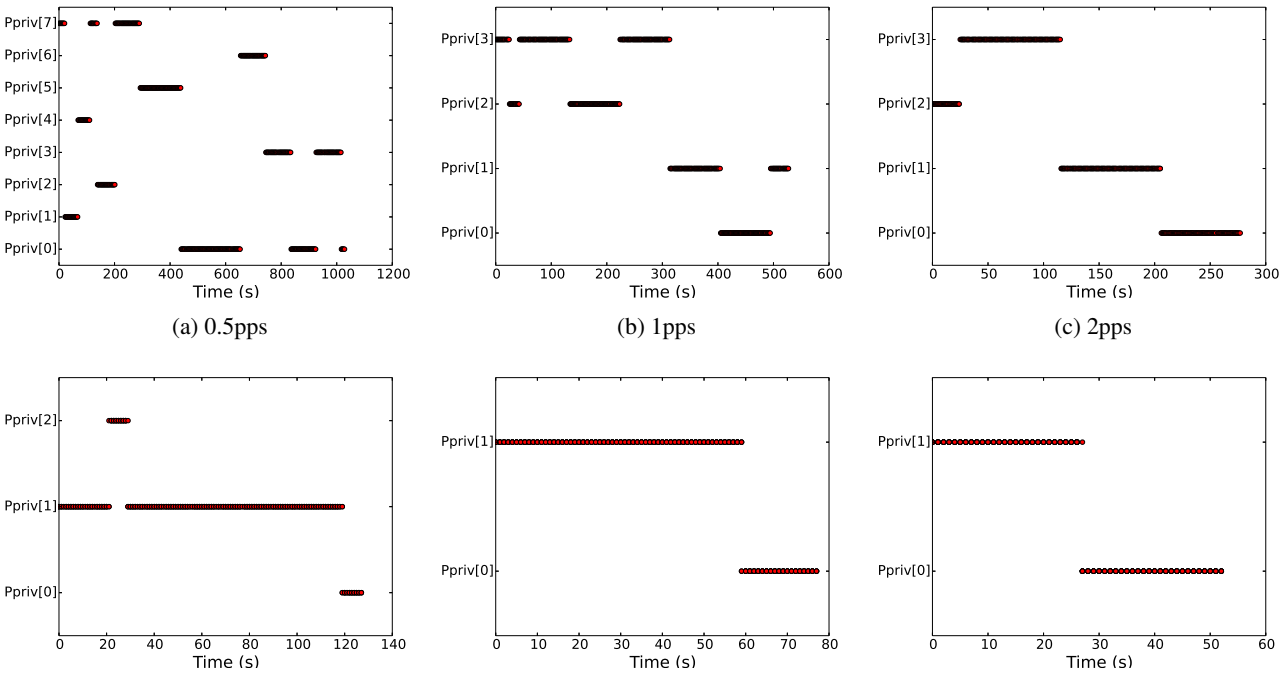
Figure 7: Reduction of incorrect guesses using the N-Streak strategy.

Improving attack fidelity using persistent B_p. The persistent B_p can be used to reduce the number of incorrect guesses. During a real-world attack, when P_{priv} is incorrectly guessed, the ciphertext replacement may crash the guest VM (although we have not experienced any crashes in our experiments). As such, a safer strategy of when to perform ciphertext replacement is only after correctly guessing P_{priv} N times in a streak, which we call the *N-streak strategy*. We then applied this strategy to Round 20, 6 and 11, which have the highest FPR (*i.e.*, 0.167, 0.133, 0.103, respectively). As shown in Figure 7, when by increasing N (*i.e.*, 1, 2, 3), the number of incorrectly performed ciphertext replacement is reduced.

Packet rate vs. B_p persistence. We further evaluated the effect of B_p persistence when the rate of SSH packets varies. Again, on the remote machine, we used *AnJian* to generate keystrokes at a fixed rate, ranging from 0.5 keystrokes per second, to 20 keystrokes per second. The rate of SSH acknowledgement packets is close to the keystroke rate. For each keystroke rate, 500 keystrokes were generated and the number of different B_ps were reported in Figure 8. We can see that as the packet rate increases, fewer number of B_ps will be used to send SSH packets. We repeated this experiment and collected over 200 different B_ps after generating 5000 keystrokes with rates ranging from 0.5 to 20 per second. The statistics of the repeated use of B_ps are shown in Figure 9.

4.3 I/O Performance Degradation

Conducting page-fault based side-channel analysis to guess P_{priv} and performing ciphertext replacement will slow down



(a) 0.5pps (b) 1pps (c) 2pps

(d) 5pps (e) 10pps (f) 20pps

Figure 8: The number of different B_ps used with various rates of packets (pps).

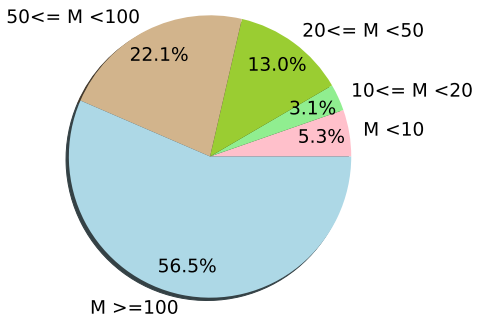


Figure 9: Statistics of repeated B_ps.

the I/O operations of the guest VM. To evaluate the degree of performance degradation, we evaluate the SSH response time on the server side during the attacks. The SSH response time measures the time interval between the QEMU receives an incoming SSH packet to the time that an SSH response packet is sent to QEMU. Note the measurements do not include network latency.

Figure 10 shows the SSH response time under three conditions: Original (not under attack), B_p Persistent (assuming B_p does not change) , and Guess Every Time (assuming B_p changes and making guesses every time). The keystroke rate used in the experiments were 10 keystrokes per second, and in total 1,000 keystrokes were generated during the tests. We can see from the figure, the average SSH response latency without attack is 2.5ms and the median is 0.99ms. The average latency for SSH connection under a B_p-persistent strategy is 6.81ms and the median is 2.4ms. The average latency for SSH connection under a guess-every-time strategy is 8.0ms and the median is 8.7ms. Because the typical network latency of cloud servers are 40-60ms within US and more than 100ms worldwide [5], it is very difficult for the VM owners to detect the latency caused by the attacks.

4.4 An End-to-End Attack

We conducted an end-to-end attack in which the adversary decrypts a 4KB memory page that is encrypted with the guest VM's K_{vek}. The attack assumes a network traffic with the rate of 10 pps, which is simulated using the same method used in the previous sections. Table 2 shows the number of packets and time used to complete the attack, when one or two 16-byte aligned blocks were exploited for the data decryption. We can see that in the four trials we conducted, roughly 300 packets are needed to decrypt the 4KB page, which takes about 40 seconds. The speed of the attack doubles if the first two blocks of the packets were used to decrypt data.

Table 2: End-to-end attack performance.

Round	1 Block		2 Blocks	
	Packets used	Time(s)	Packets used	Time(s)
1	292	43.56	148	21.29
2	329	40.78	177	20.04
3	326	39.21	154	18.99
4	299	33.58	154	16.95

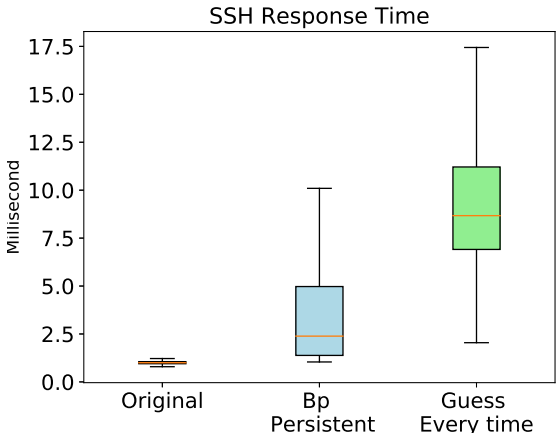


Figure 10: I/O performance degradation evaluated using SSH response time (network latency excluded).

5 A Path Towards I/O Security in SEV

The root cause of the problem is the incompatibility between AMD-V's I/O virtualization with SEV's memory encryption scheme. Specifically, the primary reason of the attacks described in Section 3 is that existing IOMMU hardware only supports memory encryption with ASID = 0 and the operated memory is encrypted with the hypervisor's memory encryption key. Therefore, every I/O operation from the guest VM must go through a shared memory page with the hypervisor. To address this limitation, IOMMU must allow DMA operations to be performed under the ASIDs of other contexts. Meanwhile, it must prevent the privileged software from abusing such IOMMU operations. This design, however, will be very challenging to implement in practice. According to our discussion with AMD researchers, future releases of SEV CPUs are *unlikely* to address this issue. Therefore, alternative solutions must be identified.

In addition to this fundamental issue, the decryption oracle is also enabled by two other vulnerabilities of SEV: (1) no integrity protection of the encrypted memory, and (2) knowledge of the tweak function $T()$. AMD researchers suggested that future SEV CPUs will disable the encryption oracle by providing memory integrity and altering the implementation of the tweak function $T()$. While authenticated memory encryption disables all known attacks against SEV, details of its implementation are yet to be disclosed. We discuss some of

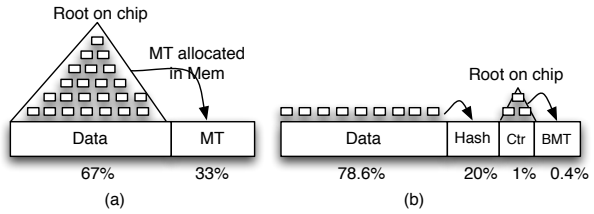


Figure 11: Merkle Tree (a) and Bonsai Merkle Tree used in conjunction with split counter mode encryption (b).

the potential considerations in Section 5.1. Future versions of the tweak function will be implemented as $T(k,a)$, where a is the physical address and k is a random input that changes after every system reboot. We leave the investigation of these vulnerabilities to future work when the technical details are published. In Section 5.2, we present a temporary software fix that works on existing AMD processors (Section 5.2).

It is worth noting that AMD researchers suggested that SEV-ES masks the page offset during page fault. However, we could not find relevant documentation or validate the claim on our testbed. Nevertheless, our analysis (see Section 3.3.2) suggests that the attack is still effective when the page offset information is unavailable. Specifically, we empirically evaluated the attack method that does not rely on page offsets by repeating the experiments in Section 4.1: the mean precision is 0.900, the mean recall is 0.730 and the mean F_1 score is 0.800, which is only slightly lower.

5.1 Authenticated Encryption

Authenticated encryption must be adopted to prevent replay attacks and replacement attacks of the encrypted ciphertext. Merkle Tree (MT) [14] has been proposed for detecting replay and replacement attacks for protecting memory integrity. MT can be built and maintained over any region of memory, and hence it can be used to protect the entire memory or only memory allocated to a VM, or any portion of it. There are two types of MT that can be used, depending on the encryption mode. For direct encryption mode, the MT covers data. For counter-mode encryption, it was shown that replay was only possible if the attacker replays data, its hash, and its counter simultaneously. Hence, protecting counter freshness is sufficient to protect against replay [26]. MT over counters is referred to as Bonsai Merkle Tree (BMT), a variant of which was chosen for implementation in Intel SGX MEE [18].

A fundamental trade off exits between the choice of encryption mode and the overheads of MT. When 128-bit hash is used for MT, MT (and hashes) over data incur memory capacity overhead of 33% (i.e. data-to-MT nodes ratio of 2:1), as illustrated in Figure 11. On the other hand, BMT incurs an overhead of 20% for hashes, plus 0.4% for BMT nodes. Hashes are needed for both encryption modes to pro-

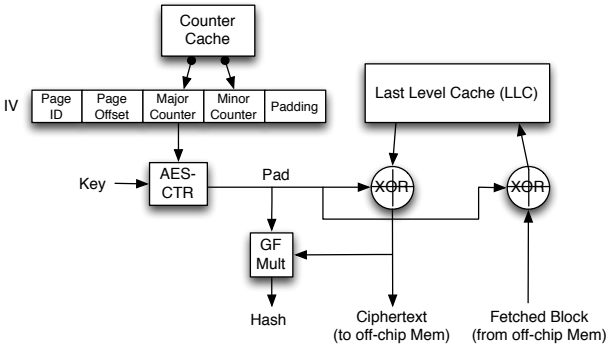


Figure 12: Counter mode encryption with split counters using Galois Counter Mode authentication.

tect against non-replay tampering of memory data. In addition, counter-mode encryption requires additional storage for counters, which depends on the type of counters. 64-bit monolithic counters take up 9% overheads, but split-counters [37] take up only 1%. Taken together, protection against replay incurs 13% memory capacity with direct encryption but only 1.4% with counter mode encryption (Figure 11). For a 1 TB memory, the difference amounts to a substantial 116 GB.

Counter mode encryption is illustrated in Figure 12. Counters are cached on chip, either in regular caches, or in a special "counter cach". With split counters, each page (4KB) of data has its own major counter, and each block (64B) in a page has its own minor counter. When there is a last level cache (LLC) miss, the counter values (major and minor) are concatenated with the page ID and block address of the page (i.e. page offset of the block) to produce a spatially and temporally unique initial vector (IV) [37]. The IV is then encrypted to produce a pad, which will be XOR-ed with ciphertext of data fetched from memory to yield data plaintext. With Galois Counter Mode, the hash of data is obtained a few clock cycles later. Counter mode encryption is more secure than direct mode encryption due to spatial and temporal uniqueness of ciphertexts even for a single plaintext value. Furthermore, as illustrated in the figure, decryption latency is largely overlapped with LLC miss latency; the only exposed latency is 1-cycle XOR of pad and data ciphertext. In contrast, direct memory mode fully exposes decryption latency in the critical path of memory fetch. Therefore, in terms of security protection against replay, memory capacity costs, and performance, AMD SEV can benefit from counter mode encryption and BMT.

A MT/BMT protects the memory from replay or tampering at all time. It is also possible to selectively protect memory region integrity only at times in which they are "expected" to be vulnerable. For example, the time window in which IOMMU buffer is vulnerable is between the time it is written by the DMA until it is read/consumed by the VM. One could take a hash of the memory region at DMA write and verify it when the VM reads the region. Any tampering or replay attempts will be detected. Selective integrity protection may obviate the need for full MT/BMT for attacks that occur within the vulnerable window, but leaves the memory integrity unprotected at other time.

5.2 A Temporary Software Solution

In this section, we present a software solution that can temporarily solve the I/O insecurity issues discussed in this paper. The key idea is to make sure the hypervisor never observe any unencrypted I/O data to/from the SEV-enabled VM. This can be achieved using SEV's platform management APIs [3] and the transport encryption key of the VM K_{tek}.

K_{tek} is a shared Diffie-Hellman (DH) key between the VM owner and the SEV firmware. Particularly, to launch an SEV-enabled VM on an SEV platform, the owner of the VM first requests the Diffie-Hellman (DH) certificate from the platform, which contains the platform's DH public key. The corresponding private key is kept inside the SEV firmware, which cannot be extracted by the system administrator or the hypervisor. The VM owner then sends her DH public key to SEV platform, so that she establishes a shared transport encryption key K_{tek} with the SEV firmware. K_{tek} is only known by the VM owner and the SEV firmware, but not known to the VM itself or hypervisor. `SEND_UPDATE_DATA` and `RECEIVE_UPDATE_DATA` are two commands (among many others) implemented by SEV to assist the hypervisor to launch and manage SEV-enabled VMs [3]. After the VM is launched, the hypervisor may use SEV's `SEND_UPDATE_DATA` command to *atomically* decrypt a piece of memory with K_{vek} and re-encrypt with K_{tek} or use `RECEIVE_UPDATE_DATA` command to decrypt the memory with K_{tek} and re-encrypt with K_{vek}.

Our proposed solution retrofits these APIs and K_{tek} to protect I/O operations. Particularly, the guest VM kernel and the QEMU can be modified so that the guest VM never copies data between the encrypted memory and the unencrypted memory. Instead, to perform any I/O operation to the SEV-enabled VM, the hypervisor issues the `SEND_UPDATE_DATA` and `RECEIVE_UPDATE_DATA` commands to atomically decrypt and re-encrypt data using the two keys K_{vek} and K_{tek}. As both keys are protected inside the SEV firmware, the hypervisor is not able to learn the plaintext during the I/O operations. The SEV firmware serves as a trusted relay of the I/O paths.

However, this solution is only a temporary fix of the issue. This is because the I/O traffic is encrypted with K_{tek}, which is only known to the owner of the VM. Therefore, all I/O operations, including network I/O, disk I/O, and display I/O must be forwarded to a trusted server that is controlled by the VM owner (as shown in Figure 13). Acting as an I/O proxy, the trusted server may limit the application scenarios of SEV and greatly reduce the I/O performance.

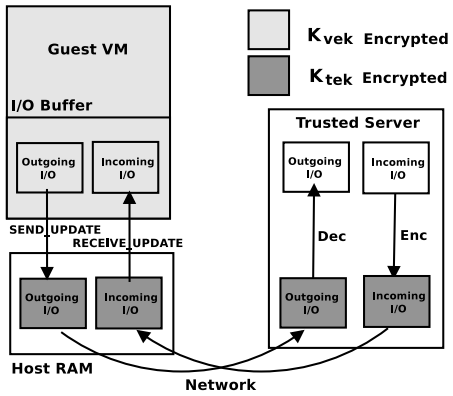


Figure 13: An illustration of the temporary software solution.

6 Related Work

6.1 Existing Security Studies on SEV

The security issues of AMD's SEV have been placed under the spotlight since its debut. Demonstrated security attacks mainly targets SEV's *unencrypted VMCB* [17] and SME's *unauthenticated memory encryption* [9, 12, 17, 25]. The former issue has been fixed using SEV-ES [19] and the latter could be addressed with integrity protection of the encrypted memory. An implementation bug in the firmware of AMD secure processors have also been reported [11]. But since the issue was not related to a design failure, we leave it as out of scope of the paper. We detail these related work as follows:

Unencrypted VMCB. Hetzelt and Buhren analyzed the security of SEV from the perspective of unencrypted virtual machine control block (VMCB) [17]. VMCB is a data structure in memory shared by the hypervisor and the guest VM, which stores the values of guest's general purpose registers and control bits for handling virtual interrupts. At the time of `VMExit`, a malicious hypervisor may learn the machine state of the guest VM by reading register values stored in the VMCB and subsequently alter their values before `VMRun` to control the registers of the guest VM. Hetzelt and Buhren [17] exploit unencrypted VMCB using code gadgets in the guest memory (similar to return-oriented programming (ROP) [29]) to arbitrarily read and write encrypted memory in the guest VM. The security issue caused by unencrypted VMCB, however, has been mitigated by SEV-ES [19], which adds another indirection layer during `VMExit` that allows the guest VM to be notified before Non-Automatic Exits (NAE)—exits requiring hypervisor emulation—and prepares a new data structure called Guest Hypervisor Communication Block—a subset of VMCB—to communicate with the hypervisor. The machine states stored in the VMCB are instead encrypted with authentication, such that they are inaccessible from the hypervisor.

Unauthenticated memory encryption. Because SME does not use authenticated encryption schemes, the integrity of the encrypted memory is not protected. As such, malicious hypervisors may alter the ciphertext of the encrypted memory without triggering errors in the decryption process of the guest VMs. Prior studies have demonstrated a variety of approaches to exploit such unauthenticated memory encryption:

- *Chosen plaintext attacks.* Du *et al.* discovered that SME uses Electronic Codebook (ECB) mode of operation in its memory encryption [12], which implies that the same plaintext always leads to the same ciphertext after encryption. As the only security measure is a physical address based tweak function *XOR*ed with the plaintext before encryption, knowledge of the tweak function will enable the adversary to deduce the relationship between the plaintext of two memory blocks (*i.e.*, of 16 bytes) if their ciphertext are the same. Du *et al.* exploit this weakness by constructing a chosen plaintext attack (via an HTTP server installed on the guest VM) and then replace the ciphertext of an `sshd` program with the ciphertext of instructions specified by the adversary (after applying the tweak functions).
- *Fault injection attacks.* Buhren *et al.* studied fault injection attacks on a simulated SME implementation [9]. Their work considers a different threat model, which assumes that the adversary is able to conduct physical DMA attacks [6] and also run an unprivileged process on the target OS. The unprivileged process performs Prime+Probe side-channel attacks to trace the execution of the SME protected application and, at the proper moment of a cryptographic operation, utilizes DMA attacks to inject memory faults to infer secret keys (or key components). We believe Buhren *et al.*'s attack against SME can be migrated to SEV as well, which is even easier to conduct as the hypervisor can be assumed to be malicious.
- *Page table manipulation.* Remapping guest pages in the nested paging structures to replay previously captured memory pages was first studied by Hetzelt and Buhren [17]. A similar idea was later demonstrated by Morbitzer *et al.* in SEVered, an attack that by manipulating the nested page table alters the virtual memory of the guest VMs to breach the confidentiality of the memory encryption [25]. More specifically, SEVered is carried out in the following steps: First, the malicious hypervisor sends network requests to the guest's network-facing application, *e.g.*, an HTTP server, which allows the attacker to download files larger than one memory page. Second, using a coarse-grained page-level side channel, the attacker determines which of the encrypted guest VM's memory pages are used to store the response data. Third, after locating these pages, the malicious hypervisor changes the page mappings in the nested page table so that these virtual pages used by the guest are mapped to different physical pages. As a result, memory content of these pages can be leaked through the responses of the network applications. The same authors further extend SEVered to perform more

realistic attacks [24], by extracting secret keys in real-world protocols and applications such as TLS, SSH, full disk encryption (FDE). Their attack makes use of the same side channels to identify the set of memory pages that are likely to contain those secrets and scans those pages (roughly 100 pages) until the secrets are found. Both these works only present decryption oracles but not encryption oracles.

While the security issues of SEV's I/O operations are orthogonal to the problems of unauthenticated memory encryption, the decryption oracle presented in this paper does rely on the lack of integrity protection for the ciphertext blocks. However, compared to previous memory decryption attacks against SEV [24, 25], our work differs primarily in three aspects. First, Morbitzer *et al.* [24, 25] manipulate unprotected nested page tables to decouple the mapping between the gVAs and the memory contents, while our decryption oracle directly replaces memory blocks used in the I/O buffer. The hardware mechanisms to defend against these two attacks may differ. Our attack highlights the necessity of mitigating both threats. Second, instead of exploiting a network-facing application executed in the guest VM to accept attacker-controlled data, our attack could make use of any I/O traffic, which is more general. Our paper suggests that application-specific defenses, such pruning secrets after use [24], may not work. Third, the attack in Morbitzer *et al.* requires the attacker to actively generate network traffic to the guest VM, which makes it easily detectable. In contrast, our decryption oracle can make use of existing I/O traffic, which can be very stealthy. Moreover, while the memory integrity issues are expected to be addressed in the next release of SEV CPUs, the fundamental I/O security problem studied in this paper will remain. The encryption oracle will not be mitigated unless the tweak function is completely secured.

Other studies. Mofrad *et al.* [23] compare Intel SGX and AMD SEV, in terms of their functionality, use scenarios, security, and performance implications. The study suggests SEV is more vulnerable than SGX as it lacks memory integrity and has a bloated trust computing base (TCB). Moreover, the performance comparison suggests AMD SEV technology performs better than Intel SGX. Wu *et al.* proposes Fidelius [35], a system that leverages a sibing-based protection mechanism to partition an untrusted hypervisor into two components, one for resource management and the other for security protection. The security of guest VMs is enhanced by the "trusted" security protection component, which, while interesting and effective, unfortunately contradicts with SEV's original intention of eliminating the hypervisor from the TCB. Fidelius mentioned a method to protect disk I/O that is similar to our temporary fix (see Section 5.2) but implies that the disk image is shipped to the SEV platform. Thus it requires using the same K_{tek} every time the disk image is used. Our proxy-style solutions in Section 5.2 is a generalization of their approach.

6.2 Security Threats of Intel TEEs

Intel TME and MKTME. Intel's counterparts of AMD's SME and SEV are Total Memory Encryption (TME) and Multi-Key Total Memory Encryption (MKTME) [18]. The concept of TME is almost the same as AMD SME: an AES-XTS encryption engine sits between a direct data path and external memory buses to encrypt data when leaving the processor and decrypt it when entering the processor. TME supports a single ephemeral encryption key for the entire processor. In contrast, MKTME supports multiple keys; it labels each page table entry with a KeyID to select one of the ephemeral AES keys generated in the encryption engine. Different from AMD SEV, guest VMs in MKTME may have more than one AES key. KeyID0 is used for guest VM to share pages with hyperviosr. KeyID*N* is assigned to guest the *N*th VM by hypervisor for guest's private page. However, the guest VM is able to obtain other KeyIDs to share memory with another guest VMs. As we were not able to purchase a machine with TME and MKTME on the market at the time of writing, we leave the analysis of these Intel's technologies to future work.

Intel SGX. Intel Software Guard eXtension (SGX) is an instruction set architecture extension that supports isolation of memory regions of userspace processes. Through a microcode-extended memory management unit, memory accesses to the protected memory regions, dubbed *enclaves*, are mediated so that only instructions belonging to the same enclave are permitted. Software attacks from all privileged software layers, including operating systems, hypervisors, system management software, are prevented by SGX. A hardware Memory Encryption Engine sits between the processor and the memory to encryption memory traffic on the fly, so that confidentiality of the enclave memory is guaranteed even with physical attackers. Remote attestation is supported in SGX to guard the *integrity* of the enclave code.

Similar to AMD's SEV, SGX constructs TEE on Intel processors. However, it differs from SEV as it only isolates portions of the user processes' memory space, whereas SEV encrypts the memory of the entire virtual machine. Developers of SEV do not need to rewrite the software when using AMD's TEE; but SGX developers have to manually partition applications into trusted and untrusted components, and recompile the source code with the SDKs provided by Intel. SGX machines have been available on market since late 2015. So far, two major types of attacks have been demonstrated to SGX applications.

- *Side-channel attacks.* Prior studies have demonstrated that enclave secrets in SGX can be exfiltrated through side channels on the CPU caches [8, 15, 16, 27], branch target buffers [22], DRAM's row buffer contention [34], page-table entries [33, 34], and page-fault exception handlers [30, 36]. More recently, side-channel attacks exploiting speculative and out-of-order execution have been

shown on SGX as well [10, 32]. Similar to SGX, SEV is not designed to thwart side-channel attacks. Therefore, we expect similar attacks can be carried out on AMD's SEV as well. Because the attacks demonstrated in this paper already completely breaks the confidentiality of SEV-protected VMs, there is no need to rely on side channels to extract secrets. However, in some of the attacks we demonstrate, side channels do facilitate the attacks. We leave the discussion on side-channel surface of SEV to future work.

- *Memory hijacking attacks.* SGX does not guard memory safety inside the enclaves. Studies [7, 21] have shown that attackers could exploit vulnerabilities in enclave programs and perform return-oriented programming (ROP) attacks [29]. Randomization-based security defenses have been proposed to mitigate ROP attacks [28]. However, as pointed out by Biondo *et al.* [7], SGX runtimes inherently contains memory regions that are hard to randomize, and thus completely eliminating the threats of memory hijacking attacks requires eradicating vulnerabilities from the enclave code. As neither SGX nor SEV is designed to provide memory safety, memory hijacking attacks are feasible on SEV as well. We will not further discuss these attacks on SEV in this paper.

AMD SEV is also vulnerable to these attacks. In this paper, we have explored a fine-grained page-fault side channel to locate the memory buffers used in the I/O operations. We leave a comprehensive study of SEV side-channel and memory-hijacking attacks to future work.

7 Conclusion

In this paper, we have reported our study of the insecurity of SEV from the perspective of the unprotected I/O operations in SEV-enabled VMs. The results of our study are two fold: First, I/O operations from SEV guests are not secure; second, I/O operations can be used by the adversary to construct memory encryption and decryption oracles. The concrete attacks have been demonstrated in the paper, along with discussion of potential solutions to the underlying problems.

Acknowledgments. We would like to thank our shepherd Dave Tan and also the anonymous reviewers for the helpful comments. The work was supported in part by the NSF grants 1750809, 1718084, 1834213, and 1834216, and research gifts from Intel and DFINITY foundation to Yinqian Zhang. Yan Solihin is supported in part by UCF.

References

[1] AMD. AMD-V nested paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf, 2008.

[2] AMD. Amd64 architecture programmer's manual volume 2: System programming, 2017.

[3] AMD. Secure encrypted virtualization api version 0.17, 2018.

[4] AMD. Solving the cloud trust problem with WinMagic and AMD EPYC hardware memory encryption. *White paper*, 2018.

[5] Amazon AWS. Optimizing latency and bandwidth for AWS traffic, 2016.

[6] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. In *CanSecWest*, 2005.

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

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*, 2017.

[9] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault attacks on encrypted general purpose compute platforms. In *7th ACM on Conference on Data and Application Security and Privacy*. ACM, 2017.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. In *4th IEEE European Symposium on Security and Privacy*. IEEE, 2019.

[11] CTS. Severe security advisory on AMD processors. https://safefirmware.com/amdflaws_whitepaper.pdf, 2017.

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

[13] Fujian Chuang YI Jia He Digital Inc. Anjian v1.1.0. www.anjian.com, 2019.

[14] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *9th International Symposium on High-Performance Computer Architecture*, 2003.

[15] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *EUROSEC*, 2017.

[16] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference*, 2017.

[17] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *ACM SIGPLAN Notices*. ACM, 2017.

[18] Intel. Intel architecture: Memory encryption technologies specification, 2017.

[19] David Kaplan. Protecting VM register state with SEV-ES. *White paper*, 2017.

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

[21] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium*, 2017.

[22] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium*, 2017.

[23] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of intel SGX and AMD memory encryption technology. In *7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2018.

[24] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting secrets from encrypted virtual machines. In *9th ACM Conference on Data and Application Security and Privacy*. ACM, 2019.

[25] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's virtual machine encryption. In *11th European Workshop on Systems Security*. ACM, 2018.

[26] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai Merkle trees to make secure processors os- and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[27] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. Springer International Publishing, 2017.

[28] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for SGX programs. In *24th Annual Network and Distributed System Security Symposium*, 2017.

[29] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*. ACM, 2007.

[30] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[31] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, 2001.

[32] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, 2018.

[33] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium*. USENIX Association, 2017.

[34] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[35] Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan. Comprehensive VM protection against untrusted hypervisor through retrofitted AMD memory encryption. In *International Symposium on High Performance Computer Architecture*, 2018.

[36] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*. IEEE, 2015.

[37] Chenyu Yan, B. Rogers, D. Englender, D. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *33rd International Symposium on Computer Architecture*, 2006.