CipherLeaks: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel

Mengyuan Li, *The Ohio State University;* Yinqian Zhang, *Southern University of Science and Technology;* Huibo Wang and Kang Li, *Baidu Security;* Yueqiang Cheng, *NIO Security Research*

https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel

Mengyuan Li[*]
The Ohio State University

Yinqian Zhang[†] ✉
Southern University of Science and Technology

Huibo Wang
Baidu Security

Kang Li
Baidu Security

Yueqiang Cheng[‡]
NIO Security Research

Abstract

AMD's Secure Encrypted Virtualization (SEV) is a hardware extension available in AMD's EPYC server processors to support confidential cloud computing. While various prior studies have demonstrated attacks against SEV by exploiting its lack of encryption in the VM control block or the lack of integrity protection of the encrypted memory and nested page tables, these issues have been addressed in the subsequent releases of SEV-Encrypted State (SEV-ES) and SEV-Secure Nested Paging (SEV-SNP).

In this paper, we study a previously unexplored vulnerability of SEV, including both SEV-ES and SEV-SNP. The vulnerability is dubbed ciphertext side channels, which allows the privileged adversary to infer the guest VM's execution states or recover certain plaintext. To demonstrate the severity of the vulnerability, we present the CIPHERLEAKS attack, which exploits the ciphertext side channel to steal private keys from the constant-time implementation of the RSA and the ECDSA in the latest OpenSSL library.

1 Introduction

AMD's Secure Encrypted Virtualization (SEV) is an extension of the AMD Virtualization (AMD-V) technology. It provides security features, such as memory encryption and isolation to virtual machines (VM), in order to support scenarios like confidential cloud computing where hypervisors are not trusted to respect the security of the VMs [2].

However, with the assumption of a malicious hypervisor, SEV faces numerous attacks. One vulnerability of the SEV is that the VM Control Block (VMCB) is not encrypted during the world switch between the guest VM and the hypervisor [15,31,35], which enables the hypervisor to inspect and/or alter the control flow of the victim VM. AMD thus released SEV Encrypted States (SEV-ES) [17], the second generation

of SEV that encrypts the sensitive portion of VMCB and stores them into the VM Save Area (VMSA) during the world switch. Therefore, these attacks can be mitigated.

However, other vulnerabilities of SEV, including unauthenticated encryption [9,11,36], unprotected nested page table (NPT) [15,26,27], unprotected I/O [23] and unauthorized Address Space identifier (ASID) [22], have been demonstrated to threaten the security of SEV-ES. To perform these attacks, the hypervisor must alter the encrypted memory or the physical address mapping of the victim VM. This is possible because SEV does not have sufficient protection for memory integrity. To tackle these issues, AMD has announced to release SEV Secure Nested Paging (SEV-SNP) in the next generation of SEV processors [4]. SEV-SNP protects the integrity of the guest VM by introducing a Reverse Map Table (RMP) to record and check the ownership of the guest VM's memory pages [2,4]. Therefore, although not yet available to be tested by security researchers, SEV-SNP is expected to be immune to all previously known attacks.

Unlike all prior work on SEV attacks, this paper presents a new side channel on SEV (including SEV-ES and SEV-SNP) processors. We call it the ciphertext side channel. It allows the privileged hypervisor to monitor the changes of the ciphertext blocks on the guest VM's memory pages and exfiltrate secrets from the guest. The root cause of the ciphertext side channel are two-fold: First, SEV's memory encryption engine uses an XOR-Encrypt-XOR (XEX) mode of operation, which encrypts each 16-byte memory block independently and preserves the one-to-one mapping between the plaintext and ciphertext pairs for each physical address. Second, the design of SEV does not prevent the hypervisor from reading the ciphertext of the encrypted guest memory, thus allowing its monitoring of the ciphertext changes during the execution of the guest VM.

To demonstrate the severity of leakage due to the ciphertext side channel, we construct the CIPHERLEAKS attack, which exploits the ciphertext side channel on the encrypted VMSA page of the guest VM. Specifically, the CIPHERLEAKS attack monitors the ciphertext of the VMSA area during VMEXITs,

---

then (1) by comparing the ciphertext blocks with the ones observed during previous VMEXITs, the adversary is able to learn that the corresponding register values have changed and thereby infer the execution state of the guest VM; and (2) by looking up a dictionary of plaintext-ciphertext pairs collected during the VM bootup period, the adversary is able to recover some selected values of the registers. With these two attack primitives, we show that the malicious hypervisor may leverage the ciphertext side channel to steal the private keys from the constant-time implementation of the RSA and ECDSA algorithms in the latest OpenSSL library, which are believed to be immune to side channels.

We discuss countermeasures of the ciphertext side channel and the specific CIPHERLEAKS attack. While there are some seemingly feasible software countermeasures, we show they become fragile when the CIPHERLEAKS attack is performed using Advanced Programmable Interrupt Controller (APIC). Therefore, we conjecture that the ciphertext side-channel vulnerability is difficult to eradicate from the software. Therefore, alternative hardware solutions must be adopted in the future SEV hardware.

Contributions. This paper contributes to the security of AMD SEV and confidential computing technology in general in the following aspects:

- It presents a novel ciphertext side channel on SEV processors. This discovery identifies a fundamental flaw in the SEV's use of XEX mode memory encryption.

- It presents a new CIPHERLEAKS attack that exploits the ciphertext side channel to infer register values from encrypted VMSA. Two primitives were constructed for inferring the execution states of the guest VM and recovering specific values of the registers.

- It presents successful attacks against the constant-time RSA and ECDSA implementation of the latest OpenSSL library, which has been considered secure against side channels.

- It discusses the applicability of the CIPHERLEAKS attack on SEV-SNP. To the best of our knowledge, the CIPHERLEAKS attack is the only working attack against SEV-SNP that breaches the memory encryption of the guest VM.

- It discusses potential software and hardware countermeasures for the ciphertext side channel and the demonstrated CIPHERLEAKS attack.

Responsible disclosure. We disclosed the vulnerability of the ciphertext side channel and the CIPHERLEAKS attack to AMD via emails in December 2020. We also distributed the first draft of this paper with AMD engineers in January 2021. AMD engineers have acknowledged the vulnerability on SEV, SEV-ES, and SEV-SNP, and filed an embargo that is effective until August 10, 2021. As of the time of writing, CVE number, CVE-2020-12966, has been reserved for the vulnerability. AMD will announce a security bulletin together with a hardware patch for SEV-SNP in August 2021.

We have also reported the vulnerable OpenSSL algorithms (see Section 4) to OpenSSL in January 2021. The OpenSSL community has acknowledged our notification, but OpenSSL will not be patched, because to properly mitigate such an attack within OpenSSL, it would require significant changes to the whole software stack. We will describe software countermeasures in Section 6.

Paper outline. The rest of the paper is outlined as follows. Section 2 describes some background knowledge of this paper. Section 3 presents an overview of the ciphertext side channel, their root causes, and two attack primitives. Section 4 sketches two end-to-end attacks against constant-time cryptography implementations in the latest OpenSSL library. Section 6 discusses the potential countermeasures. Section 7 presents the related work and Section 8 concludes the paper.

2 Background

2.1 Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is a new feature in AMD processors [19]. AMD introduces SEV for protecting virtual machines (VMs) from the untrusted hypervisor. Using the memory encryption technology, each VM will be encrypted with a unique AES encryption key, which is not accessible from the hypervisor or the VMs. The encryption is transparent to both hypervisor and VMs and happens inside dedicated hardware in the on-die memory controller. The in-use data in each VM will be encrypted by their corresponding key automatically, and thus no additional software modifications are needed to run programs containing sensitive secrets in the SEV platform. Open Virtual Machine Firmware (OVMF), the UEFI for x86 VM, and Quick Emulator (QEMU), the device simulator, are the other two critical components for the SEV-enabled VM.

Encrypted Memory. SEV hardware encrypts the VM's memory using 128-bit AES symmetric encryption. The AES engine integrated into the AMD System-on-Chip (SOC) automatically encrypts the data when it is written to the memory and automatically decrypts the data when it is read from memory. For SEV, the AES encryption uses the XOR-and-Encrypt encryption mode [12], which is later changed to an XEX mode encryption. Thus, each aligned 16-byte memory block is encrypted independently. SEV utilizes a physical address-based tweak function $T()$ to prevent the attacker from directly inferring plaintext by comparing 16-byte ciphertext [19]. It adopts a basic Xor-and-Encrypt (XE) mode on the first generation of EPYC processors (*e.g.*, EPYC 7251). The ciphertext c is calculated by XORing the plaintext m with the tweak function for system physical address P_m using $c = ENC(m \oplus T(P_m))$, where the encryption key is called VM encryption key (K_{vek}). This basic XE encryption mode can be easily reverse-engineered by the adversary as the tweak

function vectors t_is are fixed. AMD then replaces the XE mode encryption with the XOR-Encrypt-XOR (XEX) mode in EPYC 7401P processors where the ciphertext is calculated by $c = ENC(m \oplus T(P_m)) \oplus T(P_m)$. The tweak function vectors t_is are proved to have only 32-bit entropy by Wilke *et al.* [36] at first, which allows an adversary to reverse engineer the tweak function vectors. AMD adopted a 128-bit entropy tweak function vectors in their Zen 2 architecture EPYC processors from July 2019 [33] and thus fixed all existing vulnerabilities in SEV AES encryption. However, the same plaintext always has the same ciphertext in system physical address P_m during the lifetime of a guest VM.

SEV, SEV-ES, and SEV-SNP. The first version of SEV [19] was released in April, 2016. AMD later released the second generation SEV-ES [17] in February, 2017 and the whitepaper of the third generation SEV-SNP [18] in January, 2020. SEV-ES is designed to protect the register states during the world switch and introduces the VMSA to store the register states encrypted by K_{vek}. SEV-SNP is designed to protect the integrity of the VM's memory and introduces the RMP to store the ownership of each memory pages. Although SEV, SEV-ES, and SEV-SNP use the same AES encryption engine, some additional memory access restrictions are included in SEV-SNP for integrity protection. In SEV and SEV-ES, the hypervisor has read/write access to the VM's memory regions, which means the hypervisor can directly read or replace the ciphertext of the guest VM. In SEV-SNP, the RMP checks prevent the hypervisor from altering the ciphertext in the guest VM's memory by adding the ownership check before memory accesses. However, the hypervisor still has read accesses to the ciphertext of the guest VM's memory [4].

Non-Automatic VM Exits. VMEXITs in SEV-ES and SEV-SNP are classified as either Automatic VM Exits (AE) or Non-Automatic VM Exits (NAE). AE VMEXITs are events that do not need to expose any register state to the hypervisor. These events include machine check exception, physical interrupt, physical Non-Maskable-Interrupt, physical Init, virtual interrupt, pause instruction, `hlt` instruction, shutdown, write trap of CR[0-15], Nested page fault, invalid guest state, busy bit, and VMGEXIT [2]. All other VMEXITs are classified as NAE VMEXITs, which require exposing some register values to the hypervisor.

Instead of being trapped directly by the hypervisor, NAE events first result in a VC exception, which is handled by a VC handler inside the guest VM. The VC handler then inspects the NAE event's error code and decides which registers need to be exposed to the hypervisor. The VC handler copies those registers' states to a special structure called Guest-Hypervisor Communication Block (GHCB), which is a shared memory region between the guest and the hypervisor. After copying those necessary registers' states to GHCB, the VC handler executes a VMGEXIT instruction to trigger an AE VMEXIT. The hypervisor then traps the VMGEXIT VMEXIT, reads

those states from the GHCB, handles the VMEXIT, writes the return registers' states into GHCB if needed, and executes a VMRUN. After the VMRUN, the guest VM's execution will resume after the VMGEXIT instruction inside the VC handler, which copies the return values from GHCB to the corresponding registers, and then exits the VC handler. For example, to handle CPUID instructions, the VC handler stores the states of RAX and RCX and the VM EXITCODE (0x72 for CPUID) into GHCB and executes a VMGEXIT. The hypervisor then emulates the CPUID instruction and updates the values of RAX, RBX, RCX, and RDX in GHCB. After VMRUN, the VC handler checks if those return registers' states are valid and copies those states to its internal registers.

IOIO_PROT. During the Pre-Extensible Firmware Interface (PEI) initialization phase of SEV VM, IOIO port is used instead of DMA. The reason is that DMA inside SEV VM requires a shared bounce buffer between VM and the hypervisor [23]. The guest VM needs to copy DMA data from the bounce buffer to its private memory for input data and copy data from its private memory to bounce buffer for output data. Implementing bounce buffer requires allocating dynamic memory and additional memory copy operations, which is a challenge in the PEI initialization phase.

IOIO_PROT event is one of the NAE events that need to expose register states to the hypervisor. In an IOIO_PROT event, several pieces of information are returned to the hypervisor in GHCB. SW_EXITCODE contains the error code (*i.e.*, 0x7b) of IOIO_PROT events. SW_EXITINFO1 contains the intercepted I/O port (bit 31:16), address length (bit 9:7), operand size (bit 6:4), repeated port access (bit 3), and access type (*i.e.*, IN, OUT, INS, OUTS) (bit 2,0). The SW_EXITINFO2 is used to save the next RIP in non-SEV VM and SEV VM, masked to 0 in SEV-ES and SEV-SNP. For IN instructions, the hypervisor puts the RAX value into the RAX field of GHCB before VMRUN; for OUT instructions, the VC handler places the RAX register value into the RAX field of GHCB before the VMGEXIT.

2.2 Cryptographic Side-Channel Attacks

Timing attack. Timing attacks against cryptographic implementations are a subset of side-channel attacks, where the attacker exploits the time difference in the execution of a specific cryptographic function to steal the secret information. Any functions that have secret-dependent execution time variation is vulnerable to timing attacks. However, whether secrets can be stolen in practice depends on many other factors, such as the implementation of the cryptographic function, the hardware supporting the program, the accuracy of the timing measurements, *etc.*. In 1996, Paul Kocher published the first timing attack on RSA implementation [20]. Boneh and Brumley demonstrated a practical timing attack against SSL-enabled network servers in 2003, where they recovered

a server's private key based on the RSA execution time difference [8]. In fact, timing attacks are not only practical against RSA, but to other crypto algorithms, including ElGamal and the Digital Signature Algorithm [29].

Architecture side channel attack. Micro-architecture side channels are side channels that use shared CPU architecture resources to infer a victim program's behaviors. Most micro-architecture side channels exploit timing differences to infer the victim program's behaviors. Some commonly used shared resources in micro-architecture side channels include Branch Target Buffer (BTB), Cache (L1, L2, L3 cache), Translation Look-aside Buffer (TLB) and the CPU internal load-/store buffers, *etc.*. Some representative micro-architecture side-channel techniques include Flush+Reload attacks [38], Prime+Probe attack [28], utag attacks [24] and Flush+Flush attacks [14]. Those existing works show that architecture side channels can be exploited and used to break confidentiality in a local or cloud setting.

Constant-time Cryptography. Constant-time cryptography implementations [7] are widely used in mainstream cryptography library to mitigate timing attacks, the design of constant-time functions is used to reduce or eliminate data-dependent timing information. Specifically, Constant-time implementations are making the execution time independent of the secret variables, therefore, do not leak any secret information to timing analysis. To achieve constant execution time, there are three rules to follow. First, the control-flow paths cannot depend on the secret information. Second, the accessed memory addresses can not depend on the secret information. Third, the inputs to variable-time instructions such as division and modulus cannot depend on the secret information. There are a few tools developed assessing the constant-time implementations, including ImperialViolet [21], dudect [30], ct-verif [1].

2.3 Advanced Programmable Interrupt Controller

AMD processors provide an Advanced Programmable Interrupt Controller (APIC) for software to trigger interrupts [2]. Each CPU core is associated with one APIC, and several interrupt resources are supported, including APIC timer, performance monitor counter, and I/O interrupts. In the APIC timer mode, a programmable 32-bit APIC-timer counter can be used by software to generate APIC interrupts. Two modes (periodic and one-shot mode) are supported. In the one-shot mode, the counter can be set to a software-defined initial value and decrease with a clock rate. Once the counter reaches zero, an APIC interrupt is generated on this CPU core. In the period mode, the counter is automatically initialized to the initial value after reaching zero; an interrupt is generated every time the counter reaches zero.

The APIC is used in SGX-Step [34] to single-step the enclave program on Intel SGX [10]. SGX-Step builds a user-space APIC interrupt handler to intercept every APIC timer interrupt. Meanwhile, SGX-Step sets a one-shot APIC timer with a fixed value right before ERESUME. The fixed timer value is configured so that an APIC timer interrupt is generated after a single instruction is executed inside the enclave. These steps are repeated to a single step every instruction inside the enclave. SGX-Step can achieve a single-step ratio of around 98% under a machine-specific fixed counter value. However, as far as we know, no research has studied the APIC timer on the SEV platform to single-step SEV VMs.

3 The CIPHERLEAKS Attack

This section explores the side-channel leakage caused by SEV's XEX mode encryption and demonstrates its consequences when applied on the encrypted VMSA page. We particularly construct two attack primitives: execution state inference and plaintext recovery.

3.1 The Ciphertext Side Channel

We consider a scenario where the victim VM is a SEV-SNP protected VM hosted by a malicious hypervisor. We assume SEV properly protects the integrity of the encrypted VM memory as well as the VMSA pages. As such, all prior known attacks against SEV and SEV-ES (such as [15, 22, 23, 26, 27, 35]) are not applicable in our setting. The goal of the CIPHERLEAKS attack is to steal secrets from the victim VM. Denial-of-service attacks and speculative execution attacks are out-of-scope.

3.1.1 Root Cause Analysis

Because SEV's memory encryption engine uses 128-bit XEX-mode AES encryption, each 16-byte aligned memory blocks in the VMSA is independently encrypted with the same AES key. Since each 16-byte plaintext is first XORed with a physical-address-specific 16-byte value (a.k.a., the output of the tweak function) before encryption, the same plaintext may yield different ciphertext when placed in a different physical address. However, the same 16-byte plaintext is always encrypted into the same ciphertext when placed in the same physical address. Most importantly, SEV (including SEV-ES and SEV-SNP) does not prevent the hypervisor from read accessing the ciphertext of the encrypted memory (which is different from SGX).

This observation forms the foundation of our ciphertext side channel: *By monitoring the changes in the ciphertext of the victim VM, the adversary is able to infer the changes of the corresponding plaintext.* This ciphertext side channel may seem innocuous at first glance, but when applied to certain encrypted memory regions, it may be exploited to infer the execution of the victim VM.

3.1.2 CIPHERLEAKS: VMSA Inferences

The CIPHERLEAKS attack is a category of attacks that exploit the ciphertext side channel by making inferences on the ciphertext of the VMSA. We first explain in more details the VMSA structure and then outline an overview of attack.

VMSA structure. Before SEV-ES, the register states were directly saved into VMCB during the VMEXITs without hiding their states from the hypervisor, which gives the hypervisor a chance to inspect the internal states of the VM's execution or change the control flow of software inside the VM []. AMD fixes this unencrypted-register-state vulnerability by encrypting the registers during VMEXITs. In SEV-ES and SEV-SNP, the register states are encrypted and then saved into VMSA during VMEXITs. SEV-ES and SEV-SNP add additional confidentiality and integrity protection of the saved register values in VMSA.

- *Confidentiality.* The VMSA is a 4KB page-aligned memory region specified by the VMSA pointer in VMCB's offset 108h [2]. All register states saved in the VMSA are also encrypted with the VM encryption key K_{vek}.

- *Integrity.* To prevent the hypervisor from tampering VMSA, SEV-ES calculates the hash of the VMSA region before VMEXITs and stores the measurement into a protected memory region. Upon VMRUN, the hardware checks the integrity of the VMSA to prevent any modification of the VMSA data. Instead of performing such integrity checks, SEV-SNP prevents the hypervisor from writing to the guest VM's memory (including VMSA pages) via RMP permission checks.

Overview of CIPHERLEAKS. Our CIPHERLEAKS attack exploits the ciphertext side channel on the encrypted VMSA during VMEXITs. During an AE VMEXIT, all guest register values are stored in the VMSA, which is an encrypted memory page [2]. The encryption of the VMSA page also follows the same rule as other encrypted memory pages. Moreover, as the physical address of the VMSA page is chosen by the hypervisor and remains the same during the guest VM's life cycle, the hypervisor can monitor specific offsets of the VMSA to infer changes of any 16-byte plaintext. Some saved registers and their offset in the VMSA are listed in Table 1.

Some 16-byte memory blocks store two 8-byte register values. For instance, CR3 and CR0 are stored at offset 0x150. If either of the two registers changes its value, the corresponding ciphertext will change. Because CR0 does not change very frequently, in most cases, the ciphertext of this block differs because the CR3 value changes, which can infer a context switch has taken place inside the victim VM. Thus, the ciphertext pair of (CR0, CR3) can be used as identifiers of processes inside the victim VM. For other cases, like the (RBX, RDX) and (R10, R11) pairs, as both registers are subject to frequent changes, it is only possible to learn that the value of one (or both) of the two registers has changed. The adversary may learn which register has changed if she knows the executed

Table 1: Ciphertext of registers collected in the VMSA. If the content at a specific offset is 8 bytes, it means the remaining 8 bytes are reserved.

Offset	Size	Content
150h	16 bytes	CR3 & CR0
170h	16 bytes	RFLAGS & RIP
1D8h	8 bytes	RSP
1F8h	8 bytes	RAX
240h	8 bytes	CR2
308h	8 bytes	RCX
310h	16 bytes	RDX & RBX
320h	8 bytes	RBP
330h	16 bytes	RSI & RDI
340h	16 bytes	R8 & R9
350h	16 bytes	R10 & R11
360h	16 bytes	R12 & R13
370h	16 bytes	R14 & R15

binary code between the two VMEXITs.

Some 16-byte memory blocks only store values for a single 8-byte register (*e.g.*, RAX and RCX), and the remaining 8 bytes are reserved. Reserved fields are all 0s, so they never change. Therefore, from Table 1, we can see that it is possible to construct one-to-one mappings from the ciphertext to the plaintext for the values of RAX, RCX, RSP, RBP, and CR2.

3.2 Execution State Inference

We next describe two attack primitives of CIPHERLEAKS, one in Section 3.2 and the other in Section 3.3.1. First, we show the use of the ciphertext side channel to infer the execution states of processes inside the guest VM, which helps locate the physical address of targeted functions and infer the executing function of a process.

3.2.1 Attack Primitives

To infer the execution states of the encrypted VM, one could follow the steps below:

- ① At time t_0, the hypervisor clears the present bits (P bits) of all memory pages in the victim VM's NPT. The next memory access from the victim VM will trigger a VMEXIT caused by a nested page fault (NPF).

- ② During VMEXITs, the hypervisor reads and records the ciphertext blocks in the victim VM's VMSA, as well as the timestamp and VMEXIT's EXITCODE. Before VM-RUN, The hypervisor needs to reset the P bit of the faulting page so that the victim VM may continue execution. However, she may choose to clear the P bit again later to trigger more VMEXITs. This step is similar to controlled channel attacks [32, 37].

- ③ The hypervisor collects a sequence of ciphertext blocks and timestamps. By comparing the ciphertext of the CR3 and CR0 fields, the hypervisor may associate each observation to a particular process in the victim VM. Therefore, changes

Table 2: Information revealed from NPF error code.

Bit	Description
Bit 0 (P)	Cleared to 0 if the nested page was non-present.
Bit 1 (RW)	Set to 1 if it was a write access.
Bit 2 (US)	Set to 1 if it was a user access.
Bit 3 (RSV)	Set to 1 if reserved bits were set.
Bit 4 (ID)	Set to 1 if it was a code fetch.
Bit 6 (SS)	Set to 1 if it was a shadow stack access.
Bit 32	Set to 1 if it was a final physical address.
Bit 33	Set to 1 if it was a page table.
Bit 37	Set to 1 if it was a supervisor shadow stack page.

in the ciphertext blocks belonging to the same process can be collected to infer its execution states.

The NPF's error code passed to the hypervisor via VMCB's EXITINFO2 field reveals valuable information for the side-channel analysis. For example, as shown in Figure 1b, error code 0x100000014 always means the NPF is caused by an instruction fetch. The NPF error code is specified in Table 2.

The ciphertext itself is meaningless, but the fact that it changes matters. We use a vector whose size is the same as the number of registers we monitor to represent value changes in the ciphertext. A value +1 in the vector indicates that the corresponding register has changed since the last NPF. Therefore, a sequence of such vectors can be collected.

With the information described above, the hypervisor is able to profile the applications through a training process.

3.2.2 Examples

One example of such attack primitives is locating the physical address of targeted functions in the victim. Next, we illustrate such attacks using the example shown in Figure 1. We target at two `callq` instructions (❷ and ❸) in the caller function. We assume the hypervisor has some pre-knowledge of the application code running in the guest VM and the hypervisor begins to monitor the application, by clearing the P bits, before the two `call` instructions (*e.g.*, before ❶). In handling each NPFs, the hypervisor collects the ciphertext of those saved registers listed in Table 1 as well as the NPF's error code.

The hypervisor then collects a sequence of ciphertext blocks as shown in Figure 1b. The `callq` instruction at ❷ touches a new instruction page that contains the code of `sum()`. Therefore it triggers an NPF. Compared to the previous snapshot, the changes of the ciphertext of RIP, RSP, RBP, and RDI are observed; the ciphertext of CR3 and RAX remains unchanged. When `sum()` returns, the return value is stored in RAX. The ciphertext changes of the RAX register will be observed in the next NPF (at ❸), where RIP will also change. In this way, the hypervisor can locate the physical address of the functions and trace the control flow of the target application. In particular, NPF_1 reveals the physical address of function `sum()`, NPF_2 reveals the physical address of `expand()`.
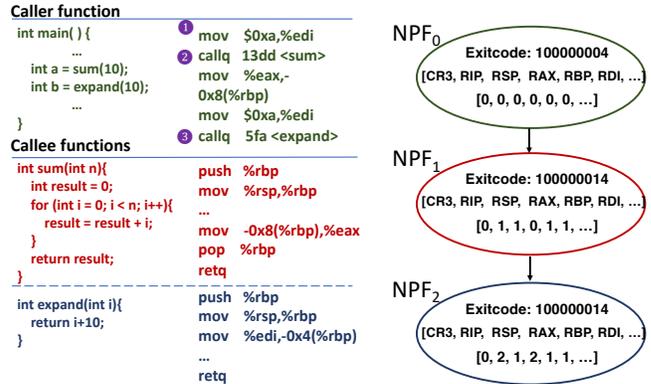
(a) C source code with assembly code.   (b) Ciphertext blocks.

Figure 1: Example about the ciphertext changes in NPFs.

3.3 Plaintext Recovery

The ciphertext side channel can also be exploited to recover the plaintext from some of the ciphertext blocks. To recover plaintext from the ciphertext, the adversary first needs to build a dictionary of plaintext-ciphertext pairs for the targeted registers, and then make use of the dictionary to recover the plaintext value of the registers of interest during the execution of a sensitive application.

3.3.1 Attack Primitive

During some NAE events, the guest kernel may exchange register states with the hypervisor through GHCB. Thus, the plaintext value of specific registers can be learned when these register states are stored in the GHCB. The hypervisor can thus collect plaintext-ciphertext pairs for those registers. Because different registers have different offset in the VMSA and different physical addresses, we need to build the dictionary of plaintext-ciphertext pairs for each register separately.

There are two ways to collect such pairs, depending on who stores the register values to GHCB. First, for those NAE events where the hypervisor returns emulated registers to the guest VM, the hypervisor may clear the P bit of the instruction page that triggers the NAE events before VMRUN. Thus, after the VC handler use `IRET` to return to the original instruction page, an NPF will occur, and the hypervisor can obtain the ciphertext of corresponding registers while handling this NPF. Figure 2a shows an example about collecting plaintext-ciphertext pairs of RAX from IOIO_PROT events (`ioread`). The hypervisor records the plaintext of RAX when emulating the VMEXIT and obtains the ciphertext of RAX when handling the NPF caused by `IRET`.

Second, for those NAE events where the VM exposes registers to the hypervisor, the hypervisor may periodically clear the P bit of the VC handler code and record the ciphertext of all registers in VMSA whenever there is an NPF triggered by the VC handler code. At the next NAE, the plaintext of
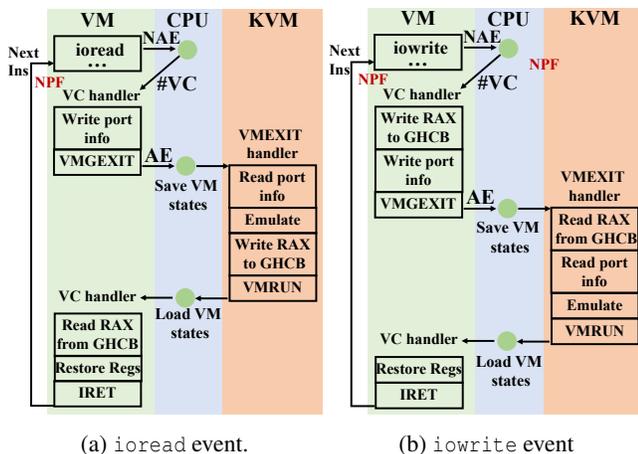
(a) `ioread` event.

(b) `iowrite` event

Figure 2: Workflow of how VC handler handles IOIO_PROT events.

some registers will be written to the GHCB, and their corresponding ciphertext can be found from the last VC handler triggered NPF. Figure 2b shows an example about collecting plaintext-ciphertext pairs of RAX from IOIO_PROT events (`iowrite`). The hypervisor obtains the ciphertext of RAX either when handling the VC-exception-triggered NPF after the NAE event or when handling the NPF caused by `IRET` and learns the plaintext of RAX when handling the VMEXIT.

3.3.2 Examples

The adversary could use the NAE VMEXITs to collect a dictionary of plaintext-ciphertext pairs for certain registers stored in VMSA. Here we present a method that leverages the IOIO_PROT (error code = 0x7b) NAE VMEXIT events to collect the ciphertext of the RAX register when its plaintext values are 0 to 127.

Building the dictionary of plaintext-ciphertext pairs. During the PEI phase, the guest VM needs to access the memory region that stores the information about the Nonvolatile BIOS settings (CMOS) and the Real-Time Clock (RTC) through IO ports 0x70 and 0x71. The OVMF code ensures the correctness of the CMOS/RTC by calling a function named `DebugDumpCmos` when loading the `PlatformPei` PEI Module (PEIM) during the initialization of the guest VM. `DebugDumpCmos` checks the CMOS/RTC by writing the offset of CMOS/RTC to port 0x70 and then reading one byte of data from port 0x71. `DebugDumpCmos` enumerates offset 0x00-0x7f (*i.e.*, 0-127) during the PEI phase to access CMOS/RTC information.

In both SEV-ES and SEV-SNP, every `iowrite` and `ioread` in IOIO_PROT are first trapped and handled by the VC handler. The VC handler and the hypervisor then cooperate to emulate `iowrite` and `ioread` as shown in Figure 2. For `iowrite`, the VC handler copies the RAX value to GHCB before calling `VMGEXIT`. For `ioread`, the VC handler copies

the RAX state from GHCB to RAX register after `VMGEXIT`. In the `iowrite` cases, the RAX state after the VC handler finishing handling an iowrite exception and before returning to the sequential instruction, should be the same as the RAX state passed to the hypervisor in the `VMGEXIT`.

In our case of `DebugDumpCmos` in `PlatformPei` PEIM, the hypervisor can observe 128 IOIO_PROT events with SW_EXITINFO1 being 0x700210 (indicating that the guest VM is accessing CMOS/RTC information) and increasing RAX values from 0x00 to 0x7f. The hypervisor can also trap the sequential instruction by clearing the P bit of the physical address of the `PlatformPei` PEIM's *EntryPoint*, which will be accessed after the guest VM exiting the VC handler. The guest physical address of *EntryPoint* is always 0x83a000 in our setting. Note that the hypervisor can also easily locate the physical address of the `PlatformPei` PEIM because the plaintext of the OVMF file is known by both the guest VM owner and the hypervisor [3] for in-place encryption during the remote attestation.

Each IOIO_PROT event in `DebugDumpCmos` helps the hypervisor record the ciphertext of a known RAX plaintext value in VMSA when handling the NPF caused by returns to the `PlatformPei` PEIM. After the `DebugDumpCmos`, the hypervisor can build a dictionary with 128 plaintext-ciphertext pairs in total, where the plaintext are from 0x00 to 0x7F. Some other IOIO_PROT events with the same SW_EXITINFO1 can also occur during the execution of `DebugDumpCmos`. The hypervisor can distinguish those events by looking at the ciphertext of RFLAG/RIP field in VMSA since all target `iowrites` inside `DebugDumpCmos` have the same RFLAG/RIP value.

3.3.3 Other Plaintext-ciphertext Pairs

In this section, we show other plaintext-ciphertext pairs the adversary may collect during the boot period of a SEV-enabled VM. We also analyze plaintext recovery under different OVMF versions and different build configurations.

All data shown in this section were collected on a workstation with 8-Core AMD EPYC 7251 Processor. The OVMF version used to boot the SEV-ES-enabled VMs may vary according to different settings that we will illustrate later. The victim VMs were configured as SEV-ES-enabled VMs with one virtual CPU, 4 GB DRAM, and 30 GB disk storage. The host and guest OS kernel were forked from branch sev-es-v3, and the QEMU version was QEMU sev-es-v12. All code is directly downloaded from AMD's Github repository [5] (commit:`96f2b75aaa9801646b410568d12b928cc9f06e0c`, Nov, 25th, 2020). We only performed the attacks on SEV-ES machines, as SEV-SNP machines were not available to us at the time of writing. But SEV-SNP is equally vulnerable (see Section 6).

Plaintext Range. To show the potential plaintext range the hypervisor can collect, we monitored all NAE events which

Table 3: Number of NAE events observed during boot period and registers state range maybe exposed. Num: the number of NAE event being observed. *: state to hypervisor. **: state from hypervisor, N/A: not observed. -: this register is not supposed to be used during this NAE event. Range R1: numbers of different exposed register states lying in [0,1], Range R2: [0,15], Range R3: [0,127], Range R4: [0,2^{64}-1].

NAE Event	Code	Num	RAX				RBX				RCX				RDX			
			R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
DR7 Read*	0x27	0	N/A	N/A	N/A	N/A	-	-	-	-	-	-	-	-	-	-	-	-
DR7 Write*	0x37	1	0	0	0	1	-	-	-	-	-	-	-	-	-	-	-	-
RDTSC*	0x6e	0	N/A	N/A	N/A	N/A	-	-	-	-	-	-	-	-	N/A	N/A	N/A	N/A
RDPMC*	0x6f	0	-	-	-	-	-	-	-	-	N/A	N/A	N/A	N/A	-	-	-	-
RDPMC**	0x6f	0	N/A	N/A	N/A	N/A	-	-	-	-	-	-	-	-	N/A	N/A	N/A	N/A
CPUID*	0x72	35328	2	6	6	276	-	-	-	-	2	11	18	1467	-	-	-	-
CPUID**	0x72	35328	1	5	6	18	2	2	3	15	1	2	3	17	2	3	4	10
IOIO_PROT*	0x7b	260648	2	16	128	8717	-	-	-	-	-	-	-	-	-	-	-	-
IOIO_PROT**	0x7b	246527	2	15	82	9033	-	-	-	-	-	-	-	-	-	-	-	-
RDMSR*	0x7c	1261	-	-	-	-	-	-	-	-	0	0	1	104	-	-	-	-
RDMSR**	0x7c	1261	2	4	4	51	-	-	-	-	-	-	-	-	1	1	2	6
WRMSR*	0x7c	12532	1	4	6	10363	-	-	-	-	0	0	1	71	1	1	2	8
RDTSCP**	0x87	0	N/A	N/A	N/A	N/A	-	-	-	-	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

have register state interactions with the hypervisor during the boot period of a SEV-ES-enabled VM. The OVMF version used was downloaded from branch sev-es-v27 with the default setting. As shown in Table 3, the collected register states are divided into 4 intervals. Range 1 (R1) is field [0,1] with only two numbers and is the most important interval since a return of true or false is very common in function implementation. Most observed NAE events can help the hypervisor to collect both two values in R1 while frequent IOIO_PROT (260648 for IO out and 246527 for IO in) events during the boot period can help the hypervisor to fill Range 2 (R2) which is [0,15] and Range 3 (R3) which is [0,127]. Range 4 (R4) contains all 2^{64} for an 8-byte register. Some NAE events are not observed during the boot period like RDPMC and RDTSC. However, these NAE events are still considered exploitable as long as some programs use these instructions during VM's lifetime. In the table, we separate RBX and RDX to present different register values the hypervisor can observe during the boot period. However, the adversary is only able to observe the ciphertext of the (RBX, RDX) pair, as these two registers are in an the same aligned 16-byte encryption block.

Different Versions. We have tested three latest (as of Nov., 25th, 2020) OVMF git branches provided by AMD [5] for SEV-ES ("sev-es-v27"[1]) and SEV-SNP ("sev-es-v21+snp"[2]) as well as the official OVMF repository used by SEV ("https://github.com/tianocore/edk2.git"[3]). All these three versions adopt the same CMOS/RTC design flow we mentioned in this section under the default configuration provided by AMD [5], and the hypervisor is able to collect all the 7-bits (plaintext from 0 to 0x7F) plaintext-ciphertext pairs in all these three versions.

Different Settings. We have also tested OVMF debug config-

---

[1]commit:834f296d3e1864b676fac9db53bc7dbb83c6eee7
[2]commit:e7bf4dfeaba60089f427af518936f29db79dd159
[3]commit:21f984cedec1c613218480bc3eb5e92349a7a812

uration options. The default debug configuration is to write debug messages to IO port 0x402. OVMF also supports original debug behavior where the debug messages are written to the emulated serial port if the DEBUG_ON_SERIAL_PORT option is set. AMD adopts the DEBUG_ON_SERIAL_PORT option according to their Github repository [5]. In both these two settings, the hypervisor is able to collect all the 7-bits plaintext-ciphertext pairs by monitoring CMOS/RTC activities in I/O PORT 0x70. The DebugDumpCmos can be disabled if the developer chooses to ignore all debug information by setting the -b RELEASE option. However, the hypervisor can still collect 19 out of the 7-bits plaintext-ciphertext pairs (with 2 numbers lying in R1, 13 numbers in R2, and 19 numbers in R3) by monitoring CMOS/RTC activities in I/O PORT 0x70. When targets at all IOIO_PROT OUT events, the hypervisor shows the potential ability to collect 115 out of the 7bits plaintext-ciphertext pairs (with 2 numbers lying in R1, 16 numbers in R2, and 115 numbers in R3), even disabling all debug activities.

4 Case Studies

In this section, we present two case studies to illustrate the CIPHERLEAKS attack. In the first attack, we show that the constant-time RSA implementation in OpenSSL can be broken with known ciphertext for the plaintext values of 0 to 31. In the second attack, we show that the constant-time ECDSA signature can be compromised with known ciphertext of the plaintext values of 0 and 1.

4.1 Breaking Constant-Time RSA

RSA is asymmetric cryptography, which is widely used in various crypto systems. In the RSA algorithm, the plaintext message m can be recovered from the ciphertext c via $m = c^d \mod n$, where d is the private key and n is the modulus

of the RSA public key system. As such, we show how the CIPHERLEAKS attack steals the private key d.

Targeted RSA implementation. Our demonstrated attack targets at the modular exponentiation used in RSA operations from the latest OpenSSL implementation (as of Nov, 4th, 2020)[4] . OpenSSL implements the modular exponentiation using a fixed-length sliding window method in function `BN_mod_exp_mont_consttime()`. We target at a *while* loop inside this function, which iteratively calculates the exponentiation in a 5-bit windows. The *while* loop is shown in Listing 1. For a 2048-bit private key, the *while* loop has about 2048/5 = 410 iterations. In each iteration, `bn_get_bits5` is called to retrieve the 5-bit of the private key `d`.

```
1  /*
2   * Scan the exponent one window at a time starting
        from the most significant bits.
3   */
4  while (bits > 0) {
5      bn_power5(tmp.d, tmp.d, powerbuf, np, n0, top,
6              bn_get_bits5(p->d, bits -= 5));
7  }
```
Listing 1: Code snippet of `BN_mod_exp_mont_consttime`.

The attacker can steal the 2048-bit private key `d` in the following steps:

① **Infer the physical address of the target function.** The attacker first uses the method introduced in Section 3.2 to obtain the physical address of the target function. We use gPA_{t0} and gPA_{t1} to denote the guest physical addresses of the target functions `bn_power5` and `bn_get_bits5`, respectively.

② **Monitor NPFs.** The attacker clears the P bit of the two targeted physical pages. Once a NPF of gPA_{t0} is intercepted, she clears the P bit of gPA_{t1}; when a NPF of gPA_{t1} is intercepted, she clears the P bit of gPA_{t0}. For a 2048-bit RSA encryption, 410 iterations can be observed, the attacker will observe 820 NPFs of gPA_{t0} and gPA_{t1} in total.

③ **Extract the private key d.** As shown in Listing 2, `bn_get_bits5` obtains 5 bits of d in each iteration, stores the value in RAX, and returns. Since the hypervisor clears the P bit of gPA_{t0}, returns to `bn_power5` will trigger a NPF of gPA_{t0}. When the hypervisor handles this NPF, it reads and records the ciphertext of RAX in the VMSA. The RAX now stores 5 bits of the private key d, and its value range is 0 to 31. The hypervisor can infer the plaintext by searching the plaintext-ciphertext pairs collected during the boot period as described in Section 3.3.2. The hypervisor can recover the whole 2048-bit private key d after a total of 410 iterations.

```
1  .globl  bn_get_bits5
2      ......
3      cmova %r11,%r10
4      cmova %eax,%ecx
5      movzw (%r10,$num,2),%eax
6      shrl  %cl,%eax
```

```
7      and \$31,%eax
8      ret
9      ......
```
Listing 2: Code segment of `bn_get_bits5()`.

4.2 Breaking Constant-time ECDSA

Elliptic Curve Digital Signature Algorithm ECDSA) is a cryptographical digital signature based on the elliptic-curve cryptography (ECC). ECDSA follows the steps below to generate a signature:

1. Randomly generate a 256-bit nonce k.
2. Calculate $r = (k \times G)_x \mod n$
3. Calculate $s = k^{-1}(h(m) + rd_a) \mod n$

where G is a base point of prime order on the curve, n is the multiplicative order of the point G, d_a is the private key, $h(m)$ is the hash of the message m, and (r, s) form the signature. With a known nonce k, the private key d_a can be calculated directly:

$$d_a = r^{-1} \times ((ks) - h(m)) \mod n$$

As such, a side-channel attack against ECDSA aims to steal the nonce k. The secret private key can be inferred thereafter.

Targeted ECDSA implementation. Our demonstrated attack targets the `secp256k1` curve, which is also used in Bitcoin wallets. In the latest OpenSSL's implementation (as of Nov, 4th, 2020) , when `ECDSA_do_sign` is called to generate a signature, `ecdsa_sign_setup` is first called to generate a random 256-bit nonce k per NIST SP 800-90A standard. To do so, `EC_POINT_mul`, `ec_wNAF_mul`, and then `ec_scalar_mul_ladder` are called to compute r, which is the x-coordinate of nonce k. `ec_scalar_mul_ladder` is used regardless of the value of the `BN_FLG_CONSTTIME` flag.

As shown in Listing 3, the core component of `ec_scalar_mul_ladder` uses conditional swaps (*a.k.a.*, `EC_POINT_CSWAP`) to compute point multiplication without branches. Specifically, in each iteration, `BN_is_bit_set(k, i)` is called to get the ith bit of the nonce `k`. The conditional swaps are determined by `kbit`, which is the XOR result of the ith bit of the nonce `k` and `pbit`.

```
1  for (i = cardinality_bits - 1; i >= 0; i--) {
2      kbit = BN_is_bit_set(k, i) ^ pbit;
3      EC_POINT_CSWAP(kbit,r,s,group_top,Z_is_one);
4  // Perform a single step of the Montgomery ladder
5      if (!ec_point_ladder_step(group, r, s, p, ctx)
         ){
6          ERR_raise(ERR_LIB_EC,
7          EC_R_LADDER_STEP_FAILURE);
8          goto err;
9      }
10 // pbit logic merges this cswap with that of the
       next iteration
11     pbit ^= kbit;
```

```
12  }
```
Listing 3: Code snippet of `ec_scalar_mul_ladder()`.

The attacker can steal the nonce k in the following steps:

① **Infer the functions' physical addresses.** The attacker first obtains the guest physical addresses of the target functions `ec_scalar_mul_ladder` gPA_{t0} and `BN_is_bit_set` gPA_{t1} using the execution inference method we introduced.

② **Monitor NPFs.** The attacker clears the P bit of the two targeted physical pages. Once a NPF of gPA_{t0} is intercepted, she clears the P bit of gPA_{t1}; when a NPF of gPA_{t1} is intercepted, she clears the P bit of gPA_{t0}. In this way, the control flow internal to the `ec_scalar_mul_ladder` function can be learned by the attacker.

③ **Learn the value of k.** In the 256-iteration *while* loop, the attacker will observes 256*5 = 1280 NPFs of gPA_{t0} and 1280 NPFs of gPA_{t1}. In each iteration of the *while* loop, the first NPFs of gPA_{t0} is triggered when `BN_is_bit_set` returns. As shown in Listing 4, the ith bit of the nonce k is returned in RAX. Thus, the ith bit of the nonce k is stored in the RAX field of the VMSA for the first NPFs of gPA_{t0} in each iteration. The attacker then compares the ciphertext of the RAX field to recover the nonce k.

```
1  000f8e20 <BN_is_bit_set>:
2            ......
3  f8e38:  48 8b 04 d0    mov    (%rax,%rdx,8),%rax
4  f8e3c:  48 d3 e8       shr    %cl,%rax
5  f8e3f:  83 e0 01       and    $0x1,%eax
6  f8e42:  f3 c3          repz retq
7            ......
```
Listing 4: Assembly code snippet of `BN_is_bit_set()`.

4.3 Evaluation

All end-to-end attacks shown in this section were evaluated on a workstation with 8-Core AMD EPYC 7251 Processor. The victim VM was configured as SEV-ES-enabled VMs with one virtual CPU, 4 GB DRAM, and 30 GB disk storage. The versions of the guest and host OS, QEMU, and OVMF are the same as described in Section 3.3.3. The latest OpenSSL from Github was used in the evaluation (commit:8016faf156287d9ef69cb7b6a0012ae0af631ce6, Nov, 4th, 2020). These attacks can also be applied to VMs with multiple vCPUs as well, but the adversary needs to collect ciphertext-plaintext dictionaries for each vCPU independently, since each vCPU has its own VMSA.

To locate the physical address of the target function, the attacker must train the pattern of ciphertext changes in a training VM (a different VM from the victim VM). In the training VM, the attacker first repeats the RSA encryption and the ECDSA signing several times by calling APIs from the OpenSSL library (with the same version as the targeted OpenSSL library in the victim VM). The attacker also collects the NPF sequence, the corresponding VMSA ciphertext changes (see

Section 3.2), as well as the ground truth (guest physical address) for the target functions. In our experiments, the pattern of ciphertext changes is very stable, especially for a function call without many branches (*e.g.*, ECDSA_do_sign() for ECDSA). As such, simple string comparison is sufficient for pattern matching and no sophisticated machine learning techniques are required.

In the attack phase, the victim VM performs an RSA encryption or an ECDSA signature using the OpenSSL library, which can be triggered by the attacker remotely but it is not a necessary condition for a successful attack. As the attacker does not know the start time of the targeted program, she must consider every newly observed CR3 ciphertext as the beginning of the targeted crypto code. It clears all P bits and starts monitoring the pattern of ciphertext changes. If the expected ciphertext change pattern is observed, the attacker can continue to steal the secret from the victim VM.

In both of the two cases we presented, we repeated the experiment 10 times and each time the attacker was able to identify the trained ciphertext pattern and recover the private key *d* and the secret nonce k with 100% accuracy. We measured the time needed to steal the 2048-bit private key d and the secret nonce k 10 times after the ciphertext change pattern is identified. The average time needed to obtain the private key d is 0.40490 seconds with a standard deviation of 0.08920 seconds. The average time needed to steal the secret nonce k is 0.10226 seconds with a standard deviation of 0.00330 seconds.

5 Countermeasures

In this section, we first discuss several potential software-level countermeasures for the CIPHERLEAKS attack. We then show the CIPHERLEAKS attack can still work by exploiting the Advanced Programmable Interrupt Controller (APIC) to collect the function's internal state. Thus, none of that software may work properly. We also discuss hardware-level countermeasures in Section 5.3.

5.1 Software Mitigation

Solutions to the ciphertext side channel can be categorized into two kinds: preventing the collection of the plaintext-ciphertext dictionary and preventing exploitation by modifying targeted functions.

Preventing dictionary collection. One potential solution is to remove unnecessary IOIO_PROT events. However, other NAE event may still serve the same purposes as IOIO_PROT. More importantly, as we have shown in Section 4.2, the hypervisor can steal the nonce k with only two plaintext-ciphertext pairs. Complete removal of all such leak sources is required to make the solution effective, almost impossible in SEV's current design.

Preventing exploitation. To fix the target functions, changes to the whole software stack may be necessary. We list three potential solutions below, but unfortunately, these approaches can be bypassed using the method we outline in Section 5.2.

- **Masking the return value in RAX.** If the return value only needs a few bits to represent, compilers can introduce randomness into the higher bits of the return value. For example, if the return is 1, then a random number can be added to mask the RAX, *e.g.*, by returning RAX = 0x183af6b800000001, where the higher 4-byte are generated randomly. The caller of the function can ignore the higher bits. In this way, the ciphertext of RAX will be new and thus unknown to the adversary.

- **Passing return values through memory or other registers.** The return value can be passed to the caller via stack. As the physical address of the stack frame is hard to predict and collect beforehand, attacks can be prevented. Similarly, the software can also write the return value to other registers (*e.g.*, R10), which can avoid using the RAX register.

- **Using inline functions or keep the callee code on the same page.** If the code of the caller and the callee are on the same page, for instance, by using inline functions, no NPFs will be triggered during function return.

These three potential solutions require significant rewriting of sensitive functions, which may require compiler-assisted tools to perform. However, the success of all these solutions relies on the assumption that the hypervisor cannot infer the internal states of a function call, which, as we will show in Section 5.2 shortly, is not true.

5.2 Function's Internal States Intercept

We present an APIC-based method to allow the hypervisor to single-step the functions in order to intercept the function's internal states. Therefore, the adversary can learn the internal states of a targeted function. Our method, though conceptually similar to SGX-Step [34], requires integrating the APIC handling code into the VMEXIT handler of KVM. Moreover, unlike SGX-Step that uses a static APIC interval to interrupt the controller, we need to select APIC intervals as the execution time of VMRUN is not constant. More specifically, the following steps are taken to interrupt VMRUN:

① **Infer the functions' physical addresses.** The attacker first obtains the guest's physical addresses of the target function, namely gPA_t, using the execution state inference method we introduced.

② **Dynamically determine APIC timer intervals.** The attacker follows a "0 steps is better than several steps" principles to single step or intercept a small advancement of the execution of the target function. Because the time used for VMRUN instruction is not fixed, the hypervisor always starts with a small APIC interval to single step into the guest VM

as much as possible. The hypervisor then checks the VMSA field to see if the ciphertext in VMSA has changed; if so, it means that one or several registers' value have changed and the guest VM executes one or several instructions before interrupted by APIC. The algorithm to choose the proper APIC time interval is specified in Algorithm 1.

**Algorithm 1:** Dynamic Timer Interval Prediction

int apic_time_interval; //APIC interrupts the VM after the interval
int roll_back ; //roll back to a small interval after any movement
apic_time_interval = 20 ;
roll_back = 10; // initialize the setting, may vary in different CPU
while true **do**
    apic_timer_oneshot(apic_time_interval);
    __svm_sev_es_vcpu_run(svm->vmcb_pa);
    svm_handle_exit(vcpu, physical interrupt VMEXIT) ;
    if not observe VMSA changes **then**
        apic_time_interval ++;
    else
        apic_time_interval = apic_time_interval - roll_back ;
    end
end

---

③ **Collect the target function's internal states.** The hypervisor can collect the internal states of the target function after a WBINVD instruction which is used to flush VMSA's cache back to the memory. With a known binary, the hypervisor may also determine the number of the instructions that have been executed by comparing the ciphertext blocks changes with the assembly code.

Evaluation. To evaluate the effectiveness of single-stepping the guest VM's execution, we perform experiments on a workstation with 8-Core AMD EPYC 7251 Processor. The victim VM was configured as SEV-ES-enabled VMs with two virtual CPUs, 4 GB DRAM, and 30 GB disk storage. The versions of the guest and host OS, QEMU, and OVMF are the same as described in Section 3.3.3. Unlike the previous settings, we enable SEV-ES's debug option in the guest policy, which allows the hypervisor to use SEV_CMD_DBG_DECRYPT command to decrypt the guest VM's VMSA. This configuration is only to collect ground truth of the experiments, which will not influence the guest VM's execution and is not a required step in practical attacks.

To make the experiments representative, we randomly select the starting point during the VM's execution to initiate our tests. In each test, we follow Algorithm 1 to collect 100 trials. Each trial is collected only when the hypervisor observes changes in the register's ciphertext in the VMSA. Meanwhile, we collected ground truth by using the SEV_CMD_DBG_DECRYPT command from the hypervisor to decrypt the RIP filed in VMSA. We use Δ to represent the number of bytes that the RIP has advanced between two consecutive VMEXITs. Note that the SEV_CMD_DBG_DECRYPT command will not affect the execution of the guest VM. We repeat the test 60 times. In total, 6000 trials are collected.
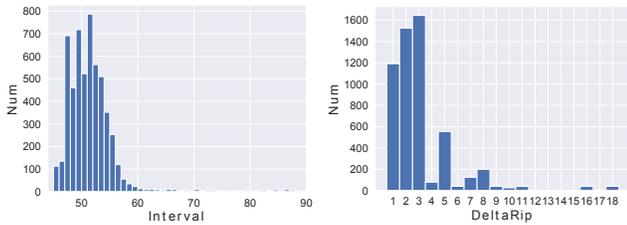
(a) Interval when VMSA changes.   (b) Δ when VMSA changes.

Figure 3: Performance of stepping VM execution using APIC.

Among the 6000 trials, 454 lead to Δ greater than 20 because of a `jmp` instruction (thus can be filtered out). For the remaining 5546 trials, the APIC-timer intervals used to trigger APIC interrupts range from 40 to 90 (with a divide value of 2, this translates from 80 to 180 CPU cycles). The distribute is shown in Figure 3a. These results suggest that the runtime of the VMRUN instruction is not constant (on SEV-ES VM), which may be caused by the presence of VMCB cache states and the non-constant time VMSA integrity checks. Even though VMRUN is not constant-time, as shown in Figure 3b, 78.7% trials lead to Δ smaller than 3 bytes. 90.1% trials lead to Δ smaller than 5 bytes. Note that a typical x86 instruction has 2 to 4 bytes [16]. These results show that the APIC-based method can successfully interrupt the execution of the guest VM with very small steps.

5.3 Hardware Countermeasures

The root cause of the ciphertext side channel is the mode of encryption adopted in the memory encryption. AMD uses the XEX encryption mode in all SEV versions (*e.g.*, SEV, SEV-ES, and SEV-SNP) and all CPU generation (*e.g.*, Zen, Zen 2, and Zen 3). This results from a well-known dilemma in the design of memory encryption: On one hand, if the ciphertext of each 16 blocks is chained together (like in the CBC mode encryption), the static mapping between ciphertext and plaintext can be broken. However, changing one bit in the plaintext will lead to changes in a large number of ciphertext blocks. On the other hand, if freshness is introduced to each block (like the CTR mode encryption used in Intel SGX), a large amount of memory needs to be reserved for storing the counter values. However, this idea may be applied to only selected memory regions, such as VMSA. In this way, the CIPHERLEAKS attack against VMSA can be prevented. To our knowledge, the hardware patch that will be integrated in SEV-SNP takes a similar idea for protecting VMSA. However, the ciphertext side channel still exists in other memory regions.

Alternatively, a plausible hardware solution is to prevent the hypervisor's read accesses to the guest VM's memory. This idea could be implemented with the RMP table (see Section 6), by restricting the read access from the hypervisor on guest pages. However, this feature is not yet available in SEV-SNP.

6 Applicability to SEV-SNP

To mitigate memory integrity attacks against SEV and SEV-ES [23, 27, 35, 36], AMD introduced another extension of SEV, named SEV Secure Nested Paging (SEV-SNP) [18]. AMD released the whitepaper describing in January, 2020 [4] and a hardware API document in August, 2020 [6]. Nevertheless, commercial processors supporting SEV-SNP have not been released yet. According to the technical details revealed in SEV-SNP's whitepaper, all prior attacks listed in Section 7 can be mitigated by SEV-SNP.

In this section, we discuss some of the new features introduced by SEV-SNP and discuss CIPHERLEAKS's applicability on SEV-SNP.

6.1 Overview of SEV-SNP

SEV-SNP protects guest VM's memory integrity by introducing a new structure called Reverse Map Table (RMP). Each RMP entry is indexed by the system page frame numbers; it contains the page states (*e.g.*, page's ownership, guest-valid, guest-invalid, and guest physical address) of this system page frame. The SEV-SNP VM must interact with the hypervisor to validate each RMP entry. Specifically, the guest VM needs to issue a new instruction PVALIDATE, a new instruction for guest VMs, to validate a guest physical address before the first access to that guest physical address. Any memory access to an invalid guest physical address will result in an NPF. More importantly, once a guest page is validated, the hypervisor cannot modify the RMP entry. Therefore, the guest VM itself can guarantee that its memory page is only validated once, and a one-to-one mapping between the guest physical address and system physical address mapping can be maintained.

As shown in Figure 4, RMP limits the hypervisor's capabilities of managing NPT. The RMP check is performed before the NPT walk is finished. Without RMP check, the hypervisor can easily remap guest physical address (*gPA*) to an arbitrary memory page by manipulating the page table entry in the NPT. With RMP check, if the hypervisor remaps the guest physical address to a memory page not belonging to the current guest VM or a memory page mapped to the current guest VM's other guest physical address, an invalid NPF or a mismatch NPF will be triggered, which can prevent attacks that require modification of the NPT [15, 26, 27].

Another protection enabled by RMP is that the ownership included in the RMP entry restricts the hypervisor's write permission towards the guest VM's private memory, which can prevent attacks that require directly modifying the ciphertext [11, 23, 36]. More details about existing attacks and how RMP can mitigate these attacks are introduced in Section 7.
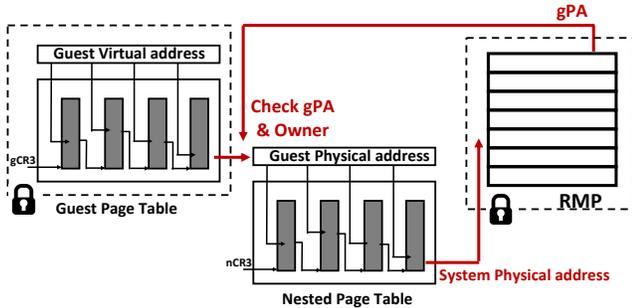
Figure 4: The RMP Check in AMD-SNP.

6.2 The CIPHERLEAKS attack on SEV-SNP

There are two key requirements of the CIPHERLEAKS attack:

- **Mapping of plaintext-ciphertext pairs of the same address does not change.** When applying the CIPHERLEAKS attack on SEV-SNP, the memory encryption mode in SEV-SNP needs to preserve the mapping between the plaintext and the ciphertext throughout the lifetime of the VM. According to [2], SEV-SNP still adopts the XEX mode of encryption, which satisfies this requirement.

- **The hypervisor must have read access to the ciphertext.** When applying the CIPHERLEAKS attack on SEV-SNP, the adversary needs to have read access to the ciphertext of guest VM's memory. According to [4], even though RMP limits the hypervisor's write access towards VM's private memory, the hypervisor still has read access to the guest VM's memory, including the VMSA area.

AMD has confirmed that SEV-SNP is also vulnerable to the CIPHERLEAKS attack. A CVE number will be assigned the discovered vulnerability for SEV-SNP and a hardware patch will be available to protect the VMSA during VMEXITs.

7 Related Work

7.1 Known Attacks against SEV

With the assumption of an untrustworthy hypervisor, SEV has faced numerous attacks caused by unencrypted VMCB [15, 31, 35], unauthenticated encryption [9, 11, 36], unprotected NPT [15, 26, 27], unprotected I/O [23] and unauthorized key use [22]. These attacks successfully break the confidentiality and/or the integrity of SEV design. AMD patched SEV with additional features SEV-ES.

Unencrypted VMCB. The VMCB is not encrypted during VMEXIT in SEV mode, which exposes SEV VM's registers state to the hypervisor. Hetzelt and Buhren [15] first showed that the untrusted hypervisor could manipulate guest VM's register during VMEXIT to perform return-oriented programming (ROP) attacks [31]. Werner *et al.* also showed by continuously monitoring unencrypted VMCB, the adversary is able to fingerprint applications inside guest VM and partially

extract guest VM's memory [35]. However, SEV-ES and SEV-SNP fix the unencrypted VMCB problem by encrypting most registers in the VMSA page during VMEXIT.

Unauthenticated encryption. The hypervisor can read and write the SEV/SEV-ES VM's memory because there is no authentication in these two modes. Previous research [9, 11, 36] showed by reverse-engineering the *physical address-based tweak function*, the adversary is able to generate useful ciphertext when there are enough known plaintext-ciphertext pairs. However, EPYC processor after the EPYC 3xx1 series fixed this problem by increasing the entropy of the tweak functions, which makes it impossible to reverse engineer the *physical address-based tweak function*. SEV-SNP further fixed this problem by removing hypervisor's write permission in guest VM's memory.

Unprotected NPT. Hetzelt and Buhren [15] first demonstrated address translation redirection attacks in SEV and discussed changing guest VM's control flow by remapping guest pages in the nPT. This method is later explored by other research works [26, 27]. In the SEVered attack [27], the adversary extracts guest VM's memory by changing the memory mapping in some network-facing applications. The adversary first triggers some network requests and then changes the mapping of the guest physical address, which is supposed to contain network data before guest VM responding to the request. Thus, some wrong memory pages will be sent back, which leaks secrets to the adversary. SEV-SNP fixed this problem by restricting unauthorized NPT remapping.

Unprotected I/O. Li *et al.* [23] exploited unprotected I/O in SEV and SEV-ES. More specifically, they showed that SEV and SEV-ES rely on a shared region within a guest VM called Software I/O Translation Lookaside Buffer (SWIOTLB) to perform I/O behaviors. This design allows the hypervisor to alter parts of I/O traffic, which helps to construct encryption and decryption oracles that can encrypt and decrypt arbitrary memory with the victim's VEK. Even SEV-SNP did not fix the unprotected I/O problem, the restriction of the hypervisor's write permission in SEV-SNP mitigates this attack.

ASID abuses. Li *et al.* [22] studied SEV's "Security-by-Crash" principle and Address Space Identify (ASID) management problem. They presented a series of attacks named CROSSLINE attacks by exploiting these problems. ASID is used as an index of encryption keys in AMD firmware as well as TLB tags and cache tags. While the hypervisor is not considered trusted, SEV still leaves the ASID management to the hypervisor and relies on a "Security-by-Crash" principle where incorrect ASIDs always cause VM crashes to protect guest VM's integrity and confidentiality. In CROSSLINE attacks, the authors showed that the adversary is able to extract the guest VM's memory blocks, which conforms to the PTE format in a stealthy way. The CROSSLINE attack can work as long as the target VM's memory encryption key is not deactivated by the hypervisor, even if the victim VM is terminated.

SEV-SNP did not change its ASID management design, but the ownership check restricts other software components from accessing the target VM's memory pages. Thus, CROSSLINE attacks cannot work in SEV-SNP.

Side-channel attacks. Architectural side channels like cache side channels [25, 38–41], performance counter tracking or TLB side channels [13] are common attacks in cloud. SEV's design increases the difficulty of performing some kinds of architectural side channels. For example, it is rather hard to perform a Flush+Reload attack when SEV is enabled [38]. This is because cache lines are tagged with the VM's ASID, indicating to which VM this data belongs, thus preventing the data from being misused by entities other than its owner. Since the cache is now tagged with ASID, cache coherence of the same physical address is not maintained if the two virtual memory pages do not have the same ASID and C-bit. So although the malicious hypervisor can access the guest VM's arbitrary physical address, she cannot directly tell whether the guest VM has accessed particular memory by measuring the time using the Flush+Reload method.

While resistant to some architectural side channels, SEV is still vulnerable to page-fault side-channel attacks, in which the adversary monitors the page faults of the SEV-enabled VM to track its execution. In SEV mode, although the mapping between the guest VM's guest virtual address (gVA) to gPA is maintained by the guest VM's page table and encrypted by the VM Encryption Key, the hypervisor could still manipulate the NPT by clearing the P bit to trap the translation from gPAs to system physical address (sPAs). Hetzelt *et al.* [15] relies on this NPF side channel to identify memory pages containing web data. Li *et al.* use the page fault side channels to locate network buffer pages [23].

8 Conclusion

This paper describes the ciphertext side channel on SEV (including SEV-ES and SEV-SNP) processors. The root causes of the side channel are two-fold: First, SEV uses XEX mode of encryption with a tweak function of the physical addresses, so that the one-to-one mapping between the ciphertext and plaintext of the same address is preserved. Second, the VM memory is readable by the hypervisor, allowing it to monitor the changes of the ciphertext blocks. The paper demonstrates the CIPHERLEAKS attack that exploits the ciphertext side-channel vulnerability to completely break the constant-time cryptography of OpenSSL when executed in SEV-ES VMs.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In 25th USENIX Security Symposium, pages 53–70, 2016.

[2] AMD. AMD64 architecture programmer's manual volume 2: System programming, 2019.

[3] AMD. SEV API version 0.22, 2019.

[4] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White paper, 2020.

[5] AMD. AMDSEV/SEV-ES branch. https://github.com/AMDESE/AMDSEV/tree/sev-es, 2020.

[6] AMD. SEV secure nested paging firmware API specification. API Document, 2020.

[7] BearSSL. Why constant-time crypto? https://www.bearssl.org/constanttime.html, 2021.

[8] David Brumley and Dan Boneh. Remote timing attacks are practical. Computer Networks, 48(5):701–716, 2005.

[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] Victor Costan and Srinivas Devadas. Intel SGX explained. IACR Cryptol. ePrint Arch., 2016(86):1–118, 2016.

[11] 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.

[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] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In 27th USENIX Security Symposium, pages 955–972, 2018.

[14] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ Flush: a fast and stealthy cache attack. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 279–299. Springer, 2016.

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

[16] Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. An analysis of x86-64 instruction set for optimization of system softwares. Planning perspectives, page 152, 2011.

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

[18] David Kaplan. Upcoming x86 technologies for malicious hypervisor protection. https://static.sched.com/hosted_files/lsseu2019/65/SEV-SNP%20Slides%20Nov%201%202019.pdf, 2020.

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

[20] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Annual International Cryptology Conference, pages 104–113. Springer, 1996.

[21] Adam Langley. Checking that functions are constant time with valgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html, 2010.

[22] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CROSSLINE: Breaking"security-by-crash"based memory isolation in amd sev. arXiv preprint arXiv:2008.00146, 2020.

[23] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In 28th USENIX Security Symposium, pages 1257–1272, 2019.

[24] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In 15th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2020), 2020.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In 2015 IEEE symposium on security and privacy, pages 605–622. IEEE, 2015.

[26] 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.

[27] 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.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In Cryptographers' track at the RSA conference, pages 1–20. Springer, 2006.

[29] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure DSA signing exponentiations really are constant-time. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1639–1650, 2016.

[30] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pages 1697–1702. IEEE, 2017.

[31] 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.

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

[33] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" processor. IEEE Micro, 40(2):45–52, 2020.

[34] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In Proceedings of the 2nd Workshop on System Software for Trusted Execution, pages 1–6, 2017.

[35] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt of them all: Inference attacks against secure virtual enclaves. In ACM Asia Conference on Computer and Communications Security, pages 73–85. ACM, 2019.

[36] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1483–1496. IEEE, 2020.

[37] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In 2015 IEEE Symposium on Security and Privacy, pages 640–656. IEEE, 2015.

[38] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: a high resolution, low noise, l3 cache side-channel attack. In 23rd USENIX Security Symposium, pages 719–732, 2014.

[39] Yinqian Zhang. Cache side channels: State of the art and research opportunities. In Proceedings of the 2017 ACM SIGSAC conference on Computer and Communications Security, pages 2617–2619, 2017.

[40] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In Proceedings of the 2012 ACM SIGSAC conference on Computer and Communications Security, pages 305–316, 2012.

[41] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in Paas clouds. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pages 990–1003, 2014.