A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP

Mengyuan Li*
The Ohio State University
li.7533@osu.edu

Luca Wilke*
University of Lübeck
l.wilke@uni-luebeck.de

Jan Wichelmann
University of Lübeck
j.wichelmann@uni-luebeck.de

Thomas Eisenbarth
University of Lübeck
thomas.eisenbarth@uni-luebeck.de

Radu Teodorescu
The Ohio State University
teodores@cse.ohio-state.edu

Yinqian Zhang
Southern University of Science and Technology
yinqianz@acm.org

***Abstract*—Hardware-assisted memory encryption offers strong confidentiality guarantees for trusted execution environments like Intel SGX and AMD SEV. However, a recent study by Li et al. presented at USENIX Security 2021 has demonstrated the CipherLeaks attack, which monitors ciphertext changes in the special VMSA page. By leaking register values saved by the VM during context switches, they broke state-of-the-art constant-time cryptographic implementations, including RSA and ECDSA in the OpenSSL.**

In this paper, we perform a comprehensive study on the ciphertext side channels. Our work suggests that while the CipherLeaks attack targets only the VMSA page, a generic ciphertext side-channel attack may exploit the ciphertext leakage from any memory pages, including those for kernel data structures, stacks and heaps. As such, AMD's existing countermeasures to the CipherLeaks attack, a firmware patch that introduces randomness into the ciphertext of the VMSA page, is clearly insufficient. The root cause of the leakage in AMD SEV's memory encryption—the use of a stateless yet unauthenticated encryption mode and the unrestricted read accesses to the ciphertext of the encrypted memory—remains unfixed. Given the challenges faced by AMD to eradicate the vulnerability from the hardware design, we propose a set of software countermeasures to the ciphertext side channels, including patches to the OS kernel and cryptographic libraries. We are working closely with AMD to merge these changes into affected open-source projects.

I. Introduction

For years, the main obstacle to cloud adoption has been a lack of trust in Cloud Service Providers (CSP). The concept of confidential Virtual Machine (VM) has been enabled by an emerging security feature in modern CPUs, dubbed Trusted Execution Environment (TEE), which removes the need to trust the CSP [15]. Aiming at providing data-in-use protection, confidential VM uses hardware-based memory encryption to protect the integrity and the confidentiality of VMs against both physical access attacks and privileged software-level attacks. Another key benefit of confidential VM is that any VM can be deployed as confidential VM on systems that support them, without costly adaption and rewriting that is necessary to turn applications into secure enclaves [12]. Due to the enormous market potential, all main processor vendors have released or are working on releasing confidential VM features in their server CPU lines, including AMD Secure Encrypted Virtualization (SEV) [23], Intel Trust Domain Extension (TDX) [19], and ARM Confidential Compute Architecture (CCA) [8].

Currently, only AMD's confidential VM solution—AMD SEV—is available and has been deployed in public clouds [15], [29]. Since its first deployment, SEV has been exhaustively analyzed by the security community. Due to the powerful adversarial scenario of a malicious hypervisor, several weaknesses have been found, including unauthenticated encryption [10], [14], [36], Nested Page Table (NPT) remapping [17], [30], [31], unprotected I/O [26], and unauthorized Address Space Identifiers (ASID) [25]. With the newest version of SEV—the recently released SEV-SNP (Secure Nested Paging [4])—most of the attacks are now mitigated.

The only software-based attack that still applied to SEV-SNP is CipherLeaks [27], a novel side-channel attack where a malicious hypervisor can steal the secret keys of RSA and ECDSA algorithms in the OpenSSL implementation by monitoring the guest VM Save Area (VMSA). Specifically, SEV's memory encryption engine adopts a deterministic XOR-Encrypt-XOR (XEX) mode of operation. For each physical address, the same 128-bit plaintext block is always encrypted to the same ciphertext block during the life cycle of the VM. Meanwhile, whenever there is a guest-host world switch, register values are encrypted and then stored in the VMSA. With the power of read access to the guest VM's VMSA area, the malicious hypervisor can continuously monitor and record the ciphertext of encrypted registers. The authors show that the ciphertext of certain registers (*e.g.*, RAX) can be used to inspect inner execution states of cryptographic algorithms and eventually reveal the private key or secrets.

Due to its severity, AMD recently released a microcode patch (MilanPI-SP3_1.0.0.5) [6] to mitigate the CipherLeaks attacks. The microcode patch enables the 3rd generation AMD EPYC processors (Milan series) to include a nonce into the encryption of the VMSA area, such that the link between the plaintext and the ciphertext is broken. As such, CipherLeaks attacks against register values in the VMSA

*The two authors contributed equally to this paper.

are no longer feasible. Note that the patch only changes the encryption of the VMSA, while the remaining memory space of the VM is still protected with the same deterministic XEX encryption as before.

In this paper, we perform a comprehensive study on the exploitability of leakage caused by ciphertext in encrypted VM memory and try to answer the question:

> *Are current cryptography implementations still safe when an attacker has access to the ciphertext of the encrypted memory?*

We broadly group ciphertext side channel attacks into two categories: *the dictionary attack* and *the collision attack*. We show that these two classes of attacks can be applied to general memory regions during cryptographic activities, including kernel data structures, stacks, and heaps, which all lead to key leakage. Most main cryptography libraries (including OpenSSL, WolfSSL, GnuTLS, OpenSSH, and libgcrypt) are shown to be vulnerable against the ciphertext side channel.

Contribution. The contributions of this paper can be summarized as follows:

- Systematically studies the ciphertext side channel in the entire memory of SEV-protected VMs. It shows that the ciphertext side channel can be exploited in *all* memory regions, including kernel structures, stacks, and heaps.
- Presents end-to-end ciphertext side-channel attacks against the ECDSA implementation of the OpenSSL library. Other main cryptography libraries (including OpenSSL, WolfSSL, GnuTLS, OpenSSH, and libgcrypt) are also shown to be vulnerable to the ciphertext side channel.
- Discusses both hardware and software countermeasures. Presents a kernel patch to mitigate ciphertext side channels caused by kernel structures. The ciphertext side channel can be mitigated when adopting the kernel patch together with software fixes for cryptographic libraries.

Responsible disclosure. We disclosed the generic ciphertext side-channel attacks on kernel data structures, heaps, and stacks to the AMD SEV team in August 2021. Henceforth, we provided more supplementary materials via email communications. AMD has acknowledged the vulnerability and had several discussions with us about potential countermeasures and stated interest in a kernel level fix. While hardware countermeasures might not be feasible in the near future for both performance and design concerns, AMD assisted us with the development of the software countermeasures, including both kernel patches (Section VI) and helping us get connected to other projects like OpenSSL.

We also disclosed the vulnerability on the code level to the communities of cryptography libraries (including OpenSSL, WolfSSl, GnuTLS, OpenSSH and libgcrypt). At the time of writing, we had received feedback from both OpenSSL and WolfSSL. They both acknowledged the concerns and recognized the necessity of addressing this vulnerability from software. WolfSSL has already provided a draft version of software fixes.

Paper outline. The rest of the paper is organized as follows: Section II introduces necessary background of this paper; Section III illustrates the root causes of ciphertext side channels in general; Section IV shows how an attacker can break current cryptography implementations by monitoring ciphertext changes in the operating system's process control block; Section V shows that the secret leakage can also be caused by stack variables and heap buffers in user space; Section VI discusses the potential countermeasures, including a kernel patch and application fixes; Section VII discusses the threat of ciphertext side channels to other confidential VM implementations; Section VIII presents state-of-the-art related work and Section IX concludes the paper.

II. Background

A. Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (SEV) is a trusted execution environment (TEE) supported by AMD server-level EPYC processors with "Zen" Architecture. SEV aims at providing confidential virtual machines for cloud customers. In SEV's threat model, other virtual machines, as well as the cloud host itself, are considered untrusted. The attacker may execute arbitrary code at the privileged hypervisor level and may also have physical access to the machine (*e.g.*, DRAM chips) [23]. To achieve this ambitious goal, a dedicated security subsystem consisting of the AMD Secure Processor (AMD-SP) and an AES memory encryption engine is introduced by SEV to protect data in use.

Hardware Memory Encryption. When SEV is enabled, the cryptographic isolation provided by Hardware Memory Encryption protects the confidentiality of the VM. Specifically, the VM's memory pages are always stored in encrypted form, and the VM encryption keys are guarded by the AMD-SP. SEV adopts a 128-bit AES encryption with the XOR-Encrypt-XOR (XEX) encryption mode, which incorporates a physical address-specific *tweak* such that the same plaintext yields different ciphertexts for each memory location. However, for a fixed address, an identical plaintext always yields the same ciphertext.

Nested Page Tables (NPT) and the page fault controlled channel. When SEV is enabled, the address translation between the VM's guest physical addresses and the host physical addresses is managed by the hypervisor with the help of a NPT, which is a two-layer page table consisting of a Guest Page Table (GPT) and a Nested Page Table (NPT). The GPT is managed inside the guest VM and thus protected by the VM encryption key. The NPT is solely managed by the hypervisor.

As shown in prior work [25], [31], [36], the hypervisor can leverage the control over the NPT to intercept the execution of the guest with page granularity. To achieve this, the hypervisor can unset the Present bit (P bit) in the NPT. The next time the VM tries to access the corresponding guest physical page, a nested page fault (NPF) will be generated, revealing the addresses of the access and the causes.

SEV extensions. Two extensions of SEV have been introduced by AMD to add additional security protections since SEV's first release in 2016.

The second generation of SEV is called SEV-ES (Encrypted State) [22], which was first introduced in 2017. SEV-ES adds additional protection for CPU registers. Prior to SEV-ES, CPU registers were stored unencrypted in the Virtual Machine Control Block (VMCB) during world switches from the VM to the hypervisor (VMEXIT). In SEV-ES, the hardware automatically encrypts the registers in a designated Virtual Machine Save Area (VMSA) along with additional integrity protection. In addition, a guest-host communication protocol was introduced for instructions that need to expose registers to the hypervisor (*e.g.*, `CPUID`, `RDMSR`, *etc.*). A VMM Communication handler (#VC handler) inside the guest VM assists the instruction emulation. Specifically, the #VC handler intercepts those instructions with the help of hardware, passes necessary register values to a shared area called Guest-Host Communication Block (GHCB), triggers a special VMEXIT by the `VMGEXIT` instruction, and reads the resulting register values from the GHCB afterwards.

The third generation of SEV is called SEV-SNP (Secure Nested Paging) [4], which was released in 2020. As a response to attacks which used remapping or modification of guest memory in order to inject code into the VM [36], a structure called Reverse Map Table (RMP) was introduced. It maintains a second translation of host physical addresses to guest physical addresses as well as keeps track of the ownership of memory pages, and thus, prevents the hypervisor from modifying or remapping the guest VM's private memory. Most of the existing attacks against SEV and SEV-ES can be mitigated by SEV-SNP (Section VIII).

B. Ciphertext Attacks against SEV-SNP

Ciphertext attacks against SEV-SNP were first introduced by Li *et al.* in CIPHERLEAKS [27]. The work exploited leakage caused by the ciphertext of the registers inside the VMSA. Specifically, by inspecting the ciphertext stored in the VMSA during VMEXITs, an attacker could (1) infer the execution state of a known binary inside the guest VM, and (2) build a ciphertext-plaintext mapping for certain registers. For example, the ciphertext of the `RAX` register could reveal the return value of function calls. Since the ciphertext was deterministic, functions that returned the same value produced an identical ciphertext for the `RAX` register inside the VMSA, which is sufficient for the attacker to distinguish secret-related data content and steal secrets from an application using the OpenSSL library.

In response to that attack, AMD added additional randomization when encrypting and saving register values into the VMSA during VMEXITs [6]. Thus, the ciphertext of the register state is now completely different even if the register values inside CPU did not change between two VMEXITs, which fully mitigates the CIPHERLEAKS attacks.

C. Off-chip Attacks

Off-chip attacks are usually classified into *stolen DIMM attacks* and *bus snooping attacks*. Stolen DIMM attacks directly grab data from the Non-Volatile Memory (NVM) or perform cold boot attacks on volatile memory [33]. Bus snooping attacks target the data transmission between two components of the computer (*e.g.*, CPU and DRAM). These attacks involve both data eavesdropping and even data altering [12].

Off-chip attacks are also considered as one of the potential attacks in a TEE's threat model [4]. While the plaintext is protected inside the chip and can hardly be inspected, all data outside the CPU might be inspected, either on the external memory buses or on the NVM. TEEs like Intel SGX and AMD SEV protect data outside the CPU by an in-chip memory encryption engine. While it is widely accepted that attacks by monitoring the data bus flow can be thwarted by memory encryption [34], researchers move their attention to the unencrypted address bus [12]. Recent results [24], [32] showed that an attacker could recover some data by monitoring memory address patterns. For those attacks, an interposer is needed to be installed on the DIMM socket. The interposer can duplicate signals on the memory bus and pass the data to a signal analyzer on the fly with CPU cycle granularity.

D. Operating System Context Switch

Under x86_64, there are four different privilege levels that can be used to implement a hierarchy in the software [3, Sec. 4.9.1]. Under Linux, ring 0 is used to run the kernel, while ring 3 is used to run user space applications. When a privilege level change occurs, *e.g.* due to an interrupt or exception, the CPU automatically switches to a separate stack and fills it with some information about the previous software. The stacks are configured in the Task State Segment (TSS). The register values, however, remain unchanged and are not stored by a hardware mechanism [3, Sec. 12.2.5]. Under Linux, one TSS per CPU is used, meaning that each CPU has its own set of stacks. Most Interruptd/Exception handlers use TSS managed as an entry point to intialy store the register values, before eventually copying them to the so-called thread stack. The thread stack is part of the Process Control Block (PCB, also called `task_struct` in Linux), a data structure that bundles all information related to a process/thread. The saved registers are referred to as the `pt_regs` structure, which simply consists of the register values stored next to each other.

Note that in other scenarios a context switch is also used to describe a switch between different processes and threads. In this work, we always refer to the aforementioned privilege level change if not stated otherwise.

III. A GENERIC CIPHERTEXT SIDE CHANNEL

In this section, we are going to show that the ciphertext-based attack demonstrated in the CIPHERLEAKS paper is not limited to the VMSA register storage mechanism of SEV-SNP, but applies to any deterministically encrypted memory. We define a generic attacker model and show two primitives that allow the attacker to infer memory contents and runtime

behavior of any application which relies on deterministically encrypted memory for protecting the confidentiality.

A. Attacker Model

We consider the standard threat model of confidential VM: The attacker has both software and physical access to the system, *i.e.*, they have unrestricted administrator capabilities and can physically access the machine. The confidential VM shields the VM's secrets from the attacker by encrypting the memory consumed by the user's application, using a deterministic memory encryption scheme with an address-based tweak, such that the ciphertext depends on the encryption key, the plaintext and the current physical address. Specifically, we target SEV-SNP, which also prevents the attacker from remapping memory containing ciphertext to other physical addresses, denies them write access to any encrypted memory, but leaves the attacker the ability to read ciphertext by software.

B. Attack Primitives

We suggest two general methods for exploiting deterministic memory encryption: A *dictionary attack* and a *collision attack*.

Dictionary attack. A dictionary attack is applicable when a secret-dependent variable features a small, predictable value range with a fixed memory address. In this case, the attacker can build a dictionary of ciphertext-plaintext mappings for this variable and selectively recover the plaintext. This is a generalization of the approach taken in the CIPHERLEAKS attack, where the authors learned ciphertext mappings for the registers stored in the VMSA.

Contrary to CIPHERLEAKS, the dictionary attack targets arbitrary memory locations and variable types. Two examples about recovering ECDSA key using stack variables (Section V-A), or registers stored during a context switch (Section IV) are presented. While this attack is quite powerful, it is restricted by the number of possible plaintexts for a given encryption block, since the attacker cannot tell which part of the plaintext has changed when observing a new ciphertext. If the targeted variable shares an encryption block with other variables which get new values frequently (*e.g.*, a loop counter), the number of possible plaintexts becomes too large to efficiently build a mapping, as is illustrated in Figure 1. We use this fact in Section VI-B to propose a countermeasure which appends random nonces to small variables.

Collision attack. A collision attack transfers the concept of secret dependent code execution to memory writes. In secret-dependent branching, the attacker exploits that the targeted algorithm executes a certain code region depending on specific values of a secret value (*e.g.*, an *if* statement checking key bits). By observing the access pattern to the respective code chunks, the attacker can learn the secret. A common countermeasure is so-called *constant-time* code, *i.e.* code that always exhibits the same control flow and memory accesses, independent of the secret. This is usually achieved by converting secret-dependent branch decisions into fixed expressions, which compute all possible results of a given operation and then use a mask to pick the desired one. One such primitive is the constant time swap CSWAP (Algorithm 1), which is used for example by the Montgomery ladder: CSWAP takes two variables a and b and a (secret) decision bit c. If the bit is set, the values of a and b are swapped; if the bit is cleared, a and b remain unchanged. The depicted code gadget always executes the same amount of instructions in the same order, and always accesses the same memory addresses, making it resistant against microarchitectural side-channel attacks.

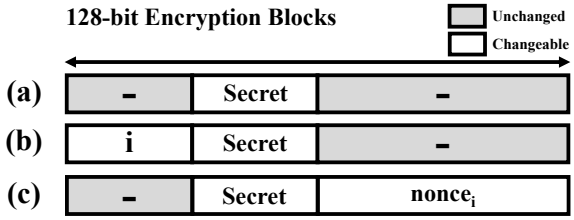


Figure 1: Encryption block configurations with different exploitability by the dictionary attack. In the first scenario (a), most of the block's plaintext is constant, with the secret being the only variable. Thus, the attacker can build a one-to-one mapping of ciphertexts to secrets. In (b), the block also contains a loop counter i, so there are many different ciphertexts mapping to the same secret. If the attacker can always observe the secret for a specific fixed value of i, they may still be able to build a dictionary, as this is equivalent to scenario (a). In the last scenario (c), the secret is followed by a random nonce which is regenerated before spilling secret to the memory. This prevents the attacker from creating a dictionary, as he never observes the same ciphertext twice.

But, if the attacker is able to observe whether the values of a or b *change*, they can immediately learn the decision bit c_i. The collision attack again exploits the fact that ciphertext blocks are deterministic. However, contrary to the dictionary attack, the attacker does not aim to learn the direct mapping of ciphertexts to actual plaintext values, but they only check whether certain ciphertexts *repeat* or *change*. Going even further, if the attacker knows that a memory write was executed (*e.g.*, through a control flow side-channel), but they do not see any ciphertext change, they learn that the instruction wrote the same value as was present in memory before. Given knowledge of the executed program, they may use this to infer more information other than the traditional control flow.

IV. LEAKAGE DUE TO CONTEXT SWITCH

We now take the dictionary attack primitive from Section III and show how it can be used for extracting register values from a VM running with SEV-SNP. After CIPHERLEAKS, AMD published a firmware patch which added protection to the VMSA area [6]. However, the VM-hypervisor world switch is not the only occasion where the entire register state is written to memory. When moving from user space to kernel space (*e.g.*, after an interrupt or an exception), the Linux kernel pushes all register values of the user program onto the stack, and then copies those into the PCB of the current

Algorithm 1 Constant time swap

Require: Byte arrays a, b of same length and decision bit c

1: **procedure** CSWAP(a, b, c)
2: $mask \leftarrow 0 - c$ ▷ $0 - 1$ underflows to 0xff
3: **for** $i = 0$ to $i = \text{length}(a)$ **do**
4: $x \leftarrow a[i] \oplus b[i]$
5: $x \leftarrow x \ \& \ mask$
6: $a[i] \leftarrow a[i] \oplus x$
7: $b[i] \leftarrow b[i] \oplus x$
8: **end for**
9: **end procedure**

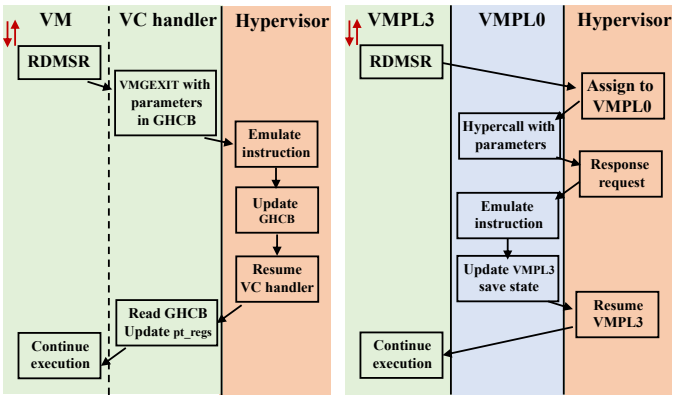
thread, such that the exception handler can access the register values through the `pt_regs` structure. The PCB address is fixed per-thread, allowing an attacker to build a dictionary of register values by causing repeated interrupts within the VM and observing the resulting ciphertexts. We show how an attacker can use nested page faults to indirectly trigger internal user-kernel context switches and use the learned register values to attack the constant-time ECDSA implementation of OpenSSL. Given their source code, similar attacks should also be applicable in WolfSSL, GnuTLS, OpenSSH, and libgcrypt.

A. Leaking Register Values via Context Switches

Forcing context switches in the VM. SEV-SNP restricts the hypervisor's ability to inject interrupts and exceptions into the VM, so we will show how a malicious hypervisor can work around this limitation by forcing the VM to pause at a certain execution point until a "natural" internal context switch is triggered, which should also be detectable by the hypervisor.

First, the hypervisor interrupts the targeted application at certain execution points by using the well-known page fault controlled channel, that allows the attacker to force a NPF when the VM tries to access or execute a given page. However, the NPF itself does not lead to a context switch inside the VM, as it is immediately intercepted by the hypervisor. To do so, the hypervisor now simply waits for a short amount of time and then resumes the VM without handling the NPF. As a result, the attacker can trap the execution of the targeted program and the victim application cannot resume its execution. After a short amount of waiting time, a time-driven internal context switch will be performed by the guest OS, which updates the victim application's register values in main memory (`pt_regs`).

Even though the internal context switch is out of the hypervisor's control, we show that the VM-host interaction mechanism adopted by SEV can work as an indicator of a finished context switch. Specifically, we observed that the guest VM has frequent interaction with the hypervisor through reading and writing hypervisor-managed registers of the Advanced Programmable Interrupt Controller (APIC), like `IA32_X2APIC_TMR1`, which are used for scheduling and timekeeping. These `RDMSR` and `WRMSR` accesses result in a special exception called #VC exception inside the VM, as they



(a) #VC handler (b) VMPL0 emulation

Figure 2: Workflow of how #VC exceptions are handled. Red arrows represent a context switch between processes.

require the VM to share registers with the hypervisor. The #VC exception handler inside the VM then calls `VMGEXIT` after putting the necessary register values into the GHCB (shown in Figure 2a). As the #VC exception is handled in VM's kernel space, a `VMGEXIT` also indicates a user-kernel context switch. Thus, the hypervisor simply waits for a `VMGEXIT` with the appropriate exit code, as an indicator of updated registers' ciphertext in `pt_regs`. We analyze the necessary pause time for triggering a `VMGEXIT` in Section IV-D.

Other than the traditional #VC handler mechanism, SEV-SNP has another option to adopt a more secure VM-host communication mechanism that moves the APIC emulation into the trust domain of the guest VM. As shown in Figure 2b, the VM is divided into multiple Virtual Machine Privilege Levels (VMPLs) that provide additional hardware isolated abstraction layers. However, the hypervisor can still sense a finished context switch due to the interaction triggered by the hypercall from VMPL0.

Locating `pt_regs` after VMEXIT. Besides using the `VMGEXIT` to detect a context switch, the attacker can also use it to locate the `pt_regs` struct. For that, after reaching a `VMGEXIT`, the attacker clears the P bit for all guest pages and resumes the VM. This will hand back control to the #VC handler in the VM, which will subsequently try to copy the results of the emulated instruction from the GHCB to `pt_regs`. Since all guest pages were marked as not present, this causes a nested page fault. In our experiments, the second NPF caused by data page read access after resuming the VM is the memory page containing `pt_regs`. We did not encounter any false positives during our experiments.

B. Attacking Constant-time ECDSA

In this section, we demonstrate how to use the context switch primitive from the previous section to attack the constant-time ECDSA implementation in OpenSSL. More precisely, we show that the adversary can infer the nonce k in the constant-time ECDSA algorithm by inspecting the ciphertext changes in the `pt_regs` structure of the targeted process. This can then be used to recover the secret key.

```
int i, cardinality_bits, group_top, kbit, pbit,
    Z_is_one;
...
for (i = cardinality_bits - 1; i >= 0; i--) {
    kbit = BN_is_bit_set(k, i) ^ pbit;
// kbit is used to determine the conditional swap
    EC_POINT_CSWAP(kbit,r,s,group_top,Z_is_one);
// single step of the Montgomery ladder
    if (!ec_point_ladder_step(group, r, s, p, ctx)){
        ERR_raise(ERR_LIB_EC,
        EC_R_LADDER_STEP_FAILURE);
        goto err;
    }
// pbit helps to merge CSWAP with that of the next
    iteration
    pbit ^= kbit;
}
```

Listing 1: Part of the elliptic curve scalar multiplication `ec_scalar_mul_ladder()` from OpenSSL. The function uses the Montgomery ladder algorithm and constant-time primitives to protect the secret scalar k against side channels.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely used signature algorithm that works as follows:

1) Prepare the curve parameters (CURVE, G, n), where G is the elliptic curve base point of prime order n.
2) Prepare a key pair by choosing uniform $d_A \in \mathbb{Z}_n^*$. d_A is the private key. The public key is $Q_A = d_A G$.
3) Generate a cryptographically secure random integer $k \in \mathbb{Z}_n^*$ (also known as the nonce k).
4) Calculate a non-zero r by $r = (kG)_x \mod n$ (only the x-coordinate of the resulting point is used).
5) Calculate $s = k^{-1}(h(m)+rd_A) \mod n$, where m is the message and $h(m)$ is a hash of m. (r, s) then forms the ECDSA signature pair.

A predictable or leaked nonce k allows to immediately recover the private key d_A by:

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

Targeted ECDSA implementation. Our attack targets the ECDSA implementation of the OpenSSL library[1] for the curve `secp384r1` that is commonly used for TLS/SSL connections. The goal of our attack is to steal the nonce k and thus infer the private key d_A. In OpenSSL, ECDSA signing is handled by the `ECDSA_do_sign` function, which in turn calls `ec_scalar_mul_ladder` to calculate `r`. Note that the implementation of the function is specifically designed to protect k against side channel attacks (Listing 1).

Identify instruction pages. Besides monitoring context switches and locating `pt_regs` via the methods shown in the previous part, we also need to identify the appropriate code locations in order to intercept the guest VM at proper execution points, which gives the attacker the opportunity to extract valuable ciphertext. In our work, we combine the widely-used page fault controlled side channel [26], [31], [35], [36] with performance counters to build a fine-grained tool to identify instruction pages' physical addresses. Specifically, we make use of the *Retired Instructions* counter [2, Event `PMCx0C0`], which can be configured to only count the amount of retired instructions inside the VM and thus reveal the number of instructions executed between two pages faults. The attacker can simply build a template of the retired instruction counts for code paths in a known binary. In our experiments, we were able to locate the target pages on the fly, without relying on repeated access patterns.

[1]Commit: `c4b2c53fadb158bee34aef90d5a7d500aead1f70`.

C. End-to-end attack against Nginx

We now show the steps needed to steal the nonce k generated by an Nginx webserver. The nonce, together with the corresponding signature, allows the attacker to recover the secret key of the server.

① **Send HTTPS request.** The attacker sends a HTTPS request to the Nginx server in order to trigger the targeted code paths.

② **Locate target function in physical memory.** Right after sending the HTTPS request, the attacker clears the P bit of all VM pages. The attacker then locates the guest physical addresses of the functions `ec_scalar_mul_ladder()` (gPA_0) and `BN_is_bit_set` (gPA_1) using the page fault channel combined with the retired instruction counter.

③ **Locate `pt_regs`.** The attacker pauses the VM for a while (*e.g.*, by trapping the VM in the NPF handler for a few milliseconds) when they intercept a NPF of gPA_0. They then use the method from Section IV-A to find the physical address gPA_3 of the current thread's `pt_regs` structure.

④ **Single-step loop iterations.** The attacker iteratively clears the P bit of gPA_1 to pause the VM when it enters `BN_is_bit_set`. After intercepting the corresponding NPF for gPA_1, the attacker clears the P bit for gPA_0, causing an NPF when the `ret` instruction inside `BN_is_bit_set` is executed, *i.e.* the function tries to return to `ec_scalar_mul_ladder()`. The attacker then pauses the VM in the gPA_0 NPF for a while (several milliseconds) and resumes the VM without handling the NPF. The attacker might observe several consecutive NPFs for gPA_0, but keeps the P bit cleared until a `VMGEXIT` is encountered.

⑤ **Record the ciphertext and recover the nonce k.** The attacker records the ciphertext of the RAX field in `pt_regs` after the `VMGEXIT`, which contains the return value of `BN_is_bit_set` at this execution point. The conjunct register stored near RAX in `pt_regs` is R8, which remains unchanged during the `for` loop. The attacker then sets the P bit of gPA_0, clear the P bit of gPA_1 in order to intercept `BN_is_bit_set` for the next iteration and repeat step ④. After 384 iterations, the attacker has collected a sequence of ciphertexts. Since RAX can only take two distinct values, they can recover the nonce `k` with only 1 bit of entropy.

D. Evaluation

All experiments throughout this paper were conducted on an AMD EPYC 7763 64-Core Processor. The host

kernel (branch `sev-snp-part2-rfc4`), QEMU (branch `sev-snp-devel`), and OVMF (branch `sev-snp-rfc-5`) were directly forked from AMD SEV's GitHub repository [5]. The victim VMs were protected by SEV-SNP and used the unmodified guest kernel provided by AMD (branch `sev-snp-part2-rfc4`). The victim VMs were configured with 2GB DRAM, 30GB disk, and one virtual CPU (vCPU). However, the capacity of the victim VMs (including vCPU, DRAM, and disk) is not relevant for the attack procedure.

For the attack on Nginx, an unmodified Nginx server and an OpenSSL library were installed inside the victim VMs. The Nginx version is 1.21.3, which was released on 07 Sep. 2021. The Nginx server supports HTTPS requests with a self-signed ECC certificate with 384-bit key. The curve used is `secp384r1`. The OpenSSL was forked from OpenSSL's Github repository (Commit: `c4b2c53fadb158bee34aef90d5a7d500aead1f70`) and was modified to log the ground truth after the signing procedure, so we could verify the extracted secret.

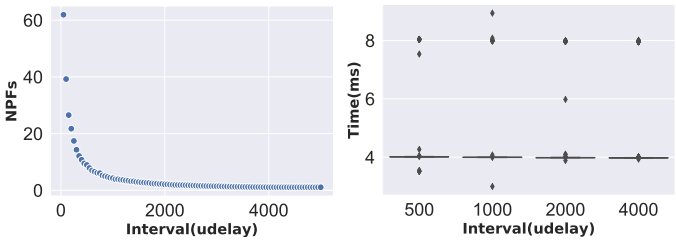
Proof of concept code is available at https://github.com/UzL-ITS/sev-ciphertext-side-channels/.

Identifying target functions. To estimate the attacker's ability to locate target functions on the fly, we sent 500 consecutive HTTPS requests. For each request, we monitored the page access pattern along with the number of retired instructions and tried to locate the target functions in real-time. The reference page access pattern and the corresponding performance counter values were collected in a different VM with the same Nginx and OpenSSL version, but without SEV-SNP's protection and with a different kernel version, to show the pattern's independence of the exact kernel version.

In 496 out of those 500 requests, the target function's physical addresses were successfully located, while a miss was reported for the remaining four requests. The average time needed to locate the target functions was 59.28 milliseconds with a standard deviation of 2.12 milliseconds. No false positive was reported.

Context-switch latency. To collect the ciphertext of the updated `pt_regs`, the attacker needs to wait until an internal context switch, which is the most time-consuming part of the end-to-end attack. In our implementation, the attacker pauses the VM by calling `udelay(<interval>)`, which takes a delay in microseconds. We evaluated both the proper interval for a direct context switch and the average waiting time. Since the attacker doesn't set the P bit at the execution point unless observing the `VMGEXIT`, the attacker might get several repeated NPFs in a row. Figure 3a shows the number of NPFs we observed under different intervals. We usually directly detected a context switch when `interval` was larger than 2000 (two milliseconds). Figure 3b shows the average waiting time. It usually took four milliseconds until an internal context switch occurred, thus we paused the victim VM by using `udelay(4000)` in our attack.

Performance. We repeated the attack 50 times and measured the overall time for an end-to-end attack. The average time



(a) #NPFs for `udelay` intervals. (b) Avg. time for context switch.

Figure 3: Relationship between `udelay` interval and internal context switch.

was 8.53 seconds with a standard deviation of 0.33 seconds. The main latency is caused by waiting for an interval context switch. For a 384-bit nonce k, the attacker can intercept 384 * 5 = 1920 NPFs for gPA_0 in total. In our setting, we chose to wait for a context switch every time when intercepting an NPF of gPA_0. However, for each iteration, only one out of five NPFs is caused by the `ret` instruction inside `BN_is_bit_set`. Thus, the attacker could also choose to only wait and grab ciphertext at that NPF. By doing that, approximately 6 seconds (384 * 4 * 4ms) waiting time can be avoided. However, one side effect is that some internal events (*e.g.*, an unexpected context switch) might cause a repeated NPF of gPA_0, which will confuse the attacker and reduce the accuracy. In our implementation, the average accuracy for the recovered nonce `k` is 89.1%.

V. Exploiting Memory Accesses in User Space

In the previous section, we have seen how an attacker can exploit the context switch mechanism of the Linux OS inside the VM to leak register values of running processes. We now turn our attention to leakages directly caused by the victim application's memory access behavior. We demonstrate that the OpenSSL ECDSA code from the previous section is also vulnerable to the dictionary attack targeting stack variables, and show an example of the collision attack against the EdDSA implementation in OpenSSH.

A. Breaking Constant-time ECDSA via Dictionary Attack

As shown in Listing 1, `ec_scalar_mul_ladder` uses several local integer variables: `kbit` controls the conditional swaps by `EC_POINT_CSWAP` in the `for` loop. Assuming that k_i refers to the i-th bit of k, at the beginning of a loop iteration, `pbit` stores k_{i-1}. After calling `BN_is_bit_set(k, i)` to retrieve k_i, `kbit` stores $k_{i-1} \oplus k_{i-2}$ (XOR). `pbit` is later updated to k_i at the end of the iteration.

Stack layout. We target the 16-byte memory block where `pbit` is stored. By our observation, the memory block containing `pbit` also contains additional variables, which is not surprising given the small size of `pbit`. In our case, `pbit`, `kbit` and `cardinality_bits` all share the same 16-byte memory block. The `cardinality_bits` variable does not change during the runtime of the `for` loop from Listing 1. Thus, the value range of the ciphertext is only dependent on the secret, *i.e.* `pbit` and `kbit`.

Table I: Possible pbit and kbit pairs when intercepting BN_is_bit_set() in ec_scalar_mul_ladder(). The letters A to D represent the 16-byte ciphertexts the attacker may observe, which depend on the values of kbit and pbit. The value of kbit and pbit in the $i+1$-th iteration is updated depending on k_i.

i-th iteration				$i+1$-th iteration		
pbit	**kbit**	**Pair**	k_i	**pbit**	**kbit**	**Pair**
0	0	A	0	0	0	A
0	0	A	1	1	1	D
0	1	B	0	0	0	A
0	1	B	1	1	1	D
1	0	C	0	0	1	B
1	0	C	1	1	0	C
1	1	D	0	0	1	B
1	1	D	1	1	0	C

Recovering k from ciphertext pairs. Recall that, at the end of each loop iteration, pbit stores the i-th bit of the nonce k. The attacker thus can recover k if they can infer the value of pbit in each iteration. We use gPA_0 to denote the guest physical address of the stack page where pbit is stored, and gPA_1 for the address of BN_is_bit_set(). Similar to the attack in Section IV-A, the attacker uses the page fault controlled channel in combination with the retired instructions performance counter for locating the pages.

The attacker records the ciphertext of gPA_0 when he intercepts the NPF of BN_is_bit_set() (gPA_1), which corresponds to the state after the previous loop iteration (*i.e.*, pbit still has its old value). As shown in Table I, in the i^{th} iteration, the attacker can observe one of four possible pbit and kbit pairs. We use the letters A to D to denote the four possible ciphertexts. At the end of the i-th iteration, pbit and kbit are updated according to k_i (0 or 1). Thus, when the attacker intercepts the NPF of gPA_1 in the $i+1$-th iteration, there are 8 possible observation cases.

They then analyze the ciphertext of gPA_0 to (1) locate the offset of the 16-byte block where pbit is in and to (2) infer the value of pbit for this iteration. For (1), the attacker can easily identify the offset because they should observe the four different ciphertext randomly but repeatedly at a certain offset, which reveals the ciphertext changes of the pair (pbit, kbit). For (2), the attacker can infer the value of pbit by analyzing two subsequent ciphertext of (pbit, kbit) as shown in Table I. The attacker applies the following algorithm to recover the pbit sequence: In the first iteration, both kbit and pbit are initialized to 1, thus producing ciphertext D. The attacker then finds an n-th iteration that has the same ciphertext as the following $n+1$-th iteration. Then (pbit, kbit) for the n-th and $n+1$-th iterations must either be A or C. If the next $n+x$-th iteration with a different ciphertext produces a ciphertext other than D, then the ciphertext for n^{th} and $n+1^{th}$ iterations must be C. Otherwise, the ciphertext represents A. After identifying A, C, and D, the remaining ciphertext represents B.

1) Attack Steps:

① **Locate the two target physical addresses.** The attacker first needs to locate the guest physical addresses of the target stack page gPA_0 and the target function page gPA_1 . We use the same methods as in Section IV-A to locate the pages.

② **Intercept the for loop.** The attacker iteratively clears the P bit in the NPT to interrupt the execution of the for loop. Specifically, the attacker clears the P bit of gPA_0 when a NPF of gPA_1 is intercepted and clears the P bit of gPA_1 when a NPF of gPA_0 is intercepted later. The attacker thus tracks the internal execution states of the for loop.

③ **Record the ciphertext of** gPA_0**.** Given the structure of the loop, there are 5 NPFs for both gPA_0 and gPA_1 for one iteration. Thus, for a 256-bit nonce k, the attacker needs to intercept 256 * 5 = 1280 NPFs for both gPA_0 and gPA_1. In each iteration, the first NPF for gPA_0 is triggered when BN_is_bit_set finishes execution and the program tries to touch the stack page where (pbit and kbit) is in. At this execution point, both kbit and the pbit are not yet updated. The attacker records the ciphertext of the whole stack page since the offset of pbit and kbit change slightly between different runs of the algorithms.

④ **Infer the value of** k**.** After all 256 iterations of the for loop, the attacker determines the offset and recovers the nonce k using the strategy we introduced in Section V-A.

2) Evaluation: The test platform was the same as described in Section IV-D. Instead of targeting the secp384r1 curve, we picked a different curve secp256k1, which is widely used in Bitcoin, to show that the attack works for different curves. The victim VM computes an ECDSA signature by calling ECDSA_do_sign in the OpenSSL library. We repeated the attack 50 times. In 92% of the attempts, we could recover the nonce k with 100% accuracy. After identifying the target functions, which we only needed to do once, the average time used to conduct the attack is 1.23 seconds with a standard deviation of 1.01 seconds.

B. Breaking Constant-time EdDSA via collision attack

In the previous attack case studies we have used the dictionary attack primitive by guessing and recording plaintext-ciphertext mappings. We now show how the attacker can break constant-time EdDSA by monitoring the collision of the secret dependent value's ciphertext. While the attack would also be applicable to the constant time swaps used by the ECDSA variant described above, we show how the collision attack can work on the constant time EdDSA implementation of OpenSSH with the ed25519 curve. As this implementation processes the secret in a batched manner, it is less susceptible to the dictionary attack previously applied to the ECDSA implementations.

The EdDSA signature algorithm [9] works similar to ECDSA, with the most noticeable difference being the deterministic nonce generation to prevent attacks based on flawed random number generators. The algorithm works as follows:

1) Provide a valid EdDSA parameter set (CURVE, G, n, c, l, H) with $2^c \cdot l = |\mathrm{CURVE}|$, where G is the elliptic curve base point of prime order l and thus $l \cdot G = 0$. H is a cryptographic hash function with $2b$ output bits.

```
1  void ge25519_scalarmult_base(ge25519_p3 *r, const
       sc25519 *k) {
2    signed char b[85];
3    int i;
4    ge25519_aff t;
5    sc25519_window3(b,k);
6    choose_t((ge25519_aff *)r, 0, b[0]);
7    fe25519_setone(&r->z);
8    fe25519_mul(&r->t, &r->x, &r->y);
9    for(i=1;i<85;i++) {
10     choose_t(&t, (unsigned long long) i, b[i]);
11     ge25519_mixadd2(r, &t);
12   }
```

Listing 2: Function performing the multiplication of the secret scalar with the curve base point. In the original code, the variable k is named s.

2) Prepare a key pair. Choose a secure random b-bit string d_A as the secret key. Calculate the public key $Q_A = d_s G$, where d_s is derived from the hash of d_A.
3) Deterministically compute a nonce for the signature as $k = H(H_{b,\ldots,2b-1}(d_A) \parallel m)$, where m is the message.
4) Calculate $R = kG$.
5) Calculate $s = k + H(R \parallel Q_A \parallel m) \cdot d_s \mod l$. The final EdDSA signature is defined as the tuple (R, s).

Targeted EdDSA implementation. We target the EdDSA implementation of OpenSSH 8.2p1, which is the version shipped with the latest Ubuntu LTS 20.04. The targeted implementation uses the `ed25519` curve. More precisely, we attack the multiplication $R = kG$ to learn k which then allows us to recover d_s from s, by computing

$$d_s = (s - k) \cdot H(R \parallel Q_A \parallel m)^{-1} \mod l.$$

While d_s is not the actual private key d_A, it is sufficient to create valid signatures.

Listing 2 shows the function performing the calculation $k \cdot G$. The arithmetic is implemented using a windowing technique with pre-computed partial sums in a lookup table. First, in line 6, the secret scalar is broken down into 3-bit chunks. In addition, a transformation is applied converting the chunks to signed values. However, this is reversible. Lines 12 and 13 in the for loop contain the main multiplication work. In `choose_t` the partial sum is loaded from the precomputation table in a cache attack resistant manner by accessing multiple values and choosing the correct one using a constant time swap operation. Line 13 performs the actual multiplication.

For our attack, we focus on the constant time swap operation `cmov_aff` that is used in `choose_t`. Both functions are shown in Listing 3. The idea of the attack is to use the collision attack to leak the value of `b`, which corresponds to d_s in our EdDSA description, in the calls to `cmov_aff`. We compare the values of `t` before and after the function call. While the constant-time swap will write to the memory locations regardless of the value of `b`, to be secure against cache and timing side channels, the actual value that is written still depends on `b`. Although the written data has a large value range, making a dictionary attack infeasible, it suffices to compare the ciphertext of `t` before and after the call to

```
1  static void cmov_aff(ge25519_aff *r, const
       ge25519_aff *p, unsigned char b) {
2    fe25519_cmov(&r->x, &p->x, b);
3    fe25519_cmov(&r->y, &p->y, b);
4  }
5
6  static void choose_t(ge25519_aff *t, unsigned long
       long pos, signed char b) {
7    fe25519 v;
8    int i = 0;
9    *t = ge25519_base_multiples_affine[5*pos+0];
10   cmov_aff(t, &ge25519_base_multiples_affine[5*pos
       +1],equal(b,1) | equal(b,-1));
11   cmov_aff(t, &ge25519_base_multiples_affine[5*pos
       +2],equal(b,2) | equal(b,-2));
12   cmov_aff(t, &ge25519_base_multiples_affine[5*pos
       +3],equal(b,3) | equal(b,-3));
13   cmov_aff(t, &ge25519_base_multiples_affine[5*pos
       +4],equal(b,-4));
14   fe25519_neg(&v, &t->x);
15   fe25519_cmov(&t->x, &v, negative(b));
16 }
```

Listing 3: Swap and lookup table access functions.

`cmov_aff` without knowing the plaintext for the ciphertext. The information whether the ciphertext value has changed or not allows us to directly infer `b`.

After leaking the value of `b`, the attacker can invert the operations applied in `sc25519_window3` (Listing 2) to recover the secret scalar k. Knowing k and the corresponding signature (R, s) allows to recover d_s, which is sufficient to create arbitrary valid signatures. Knowing d_s is not equal to knowing the secret key d_A, as the latter is still required to compute the nonce k according to step 3. However, only a party knowing the *private* key d_A can detect this subtle difference.

1) Attack Steps:

① **Trigger the OpenSSH server.** The attacker opens an SSH connection with the server, and explicitly requests the usage of the EdDSA key. EdDSA is enabled in the default configuration under Ubuntu.

② **Locate the target physical addresses.** The attacker uses the page fault controlled channel and the performance counter technique from Section Section IV-A) to infer the physical addresses of the `choose_t` and `fe25519_cmov` functions.

③ **Intercept execution before and after the constant time swap operation.** The attacker then uses the page fault controlled channel to intercept the execution of the VM by unsetting the P bit of the targeted pages in the NPT.

④ **Take snapshots of the buffer `t`.** The attacker obtains the physical address of the buffer `t` by tracking the write access pattern during the execution of the constant time swap operation using the NPF side channel. The attacker then steps the loop using the page fault controlled channel and takes snapshots of the buffer `t` in each iteration.

⑤ **Recover the secret scalar `t`.** Using the snapshots of the buffer `t` before and after each call to `fe25519_cmov` in `choose_t` (note that `cmov_aff` wraps this function), the attacker can immediately deduce the value of

b. After knowing the value of b, the attacker inverts the windowing and sign transformation operations applied in sc25519_window3(b,s) to obtain the secret scalar k. The attacker uses the first parameter R of the signature that the server sends in step ① to validate the value of k, and extracts the signing secret d_s from the second parameter S of the signature using k.

2) Evaluation: We ran the end-to-end attack 500 times. In 86% of the attacks, we could fully recover the signing secret with 100% accuracy. Of the failed attack runs, only 7 where due to errors in detecting the correct code pages. The remaining errors are most likely misdetections of the memory location of the buffer t. The average runtime of the attack was 7.9 seconds with 2.2 seconds standard deviation.

VI. Countermeasures

There are two categories of countermeasures against the attacks presented in this paper: First, the underlying issue may be addressed at the architectural level, which would likely be the most reliable approach. Otherwise, the identified problems can be also tackled at the software level, with a certain performance overhead. We discuss both hardware/architecture-based and software-based countermeasures, and point out methods for hardening existing software against the attacks presented in this paper.

A. Architectural Countermeasures

There are two possible hardware approaches for closing the ciphertext side channels. However, both approaches introduce high overhead.

First, one may change the encryption mode of SEV to use *probabilistic* encryption: a random nonce or incremental counter is included in the encryption and is updated on each memory write, effectively randomizing the resulting ciphertexts on each write. However, probabilistic memory encryption requires additional memory for storing the nonces. For example, Intel SGX combines AES-based probabilistic encryption with MACs to achieve confidentiality, integrity and replay protection. In SGX, data is encrypted in a tweaked counter mode, where the nonce depends on both the physical address of the encrypted memory block and a 56 bit counter value, to ensure replay protection [16]. The counter values are kept in the integrity tree, together with the MAC tags that ensure integrity protection. Only the head nodes of the tree are stored on-chip, while the remaining integrity tree remains in memory and needs to be checked on each memory access, resulting in a significant memory and latency overhead.

A second approach is preventing the attacker from reading the VM's physical memory: On a software/firmware layer, this could be achieved by using a similar RMP mechanics as in SEV-SNP (Section II-A), which already prevents write accesses through an additional RMP check. However, this would introduce a certain overhead when applied to all read operations due to the more frequent read access and the extra RMP lookup. For example, for a single read access inside the VM, a series of RMP checks are needed, including four checks for the 4-level GPT and one check for the data page. For each GPT level, four additional RMP checks are needed for the 4-level NPT. In addition, on-chip access control may still be susceptible to the off-chip attacks described in Section II-C.

B. Software-based Countermeasures

While hardware-based countermeasures would be preferable due to stronger security guarantees, their feasibility and practicality demand further validation. Thus, in the following sections, we describe general methods for mitigating the vulnerabilities on a software level. There is no single software-based method that is perfectly suited for all scenarios, as kernel structures, stack, and heap are all vulnerable. Thus, we present how applications can mitigate ciphertext side channels in three different ways, building on the assumption, that register values are immune to the ciphertext side channel. However, as shown in Section IV, this is not the case, as the kernel stores the registers' content in memory upon context switches. Thus, we also present how the ciphertext side channel caused by register states stored inside kernel structures can be mitigated with a kernel patch, to achieve the invariant of secure registers (Section VI-C), and measure the kernel patch performance (Section VI-D).

Secret-aware register allocation. If secret-related variables would fit into a register, but are kept in memory due to register pressure, changing the register allocation strategy may be worth pursuing. The secret-related variables can be protected by staying inside the register during their lifecycle and never being spilled to memory.

In order to do that, compiler-level modifications are needed. Even though developers can suggest the compiler to keep some variables into registers by applying a register hint (*e.g.*, register int var;), the variables are not guaranteed to be placed inside registers. Thus, a compiler can be modified to prioritize variables marked as 'secret' when allocating registers. An example of a similar scheme is GINSENG [38], which employs a custom register allocation strategy and a secure storage in a TEE to shield sensitive variables from a malicious operating system. In case a register containing a secret must be spilled to the stack anyway (*e.g.*, it is frequently used in function calls or large variables), it can be protected using a random mask as described in the later software-based probabilistic encryption part.

Limiting reuse of memory locations. Both the dictionary attack and the collision attack rely on repeated writes to a fixed physical memory address. Thus, limiting reuse of a fixed memory address leads to fresh ciphertext and can prevent the attacker from inferring secrets via the ciphertext.

To achieve this, the application developer has to identify and rewrite vulnerable code sections. For example, in our collision attack (Section V-B), the conditional swap operation should not be written to be performed in-place, but should store the result in a newly allocated memory area. In this way, an attacker always observes a fresh ciphertext in a new location, independent from the value of the decision byte c_i.

Software-based probabilistic encryption. If the aforementioned methods are not applicable, one can mimic probabilistic encryption in software and add a random nonce to the secret data each time when the data is written to the memory.

This can be approached in two ways: First, one can modify the memory layout of the affected data structures to include random nonces in between, such that each memory block gets a sufficient amount of random bits. Second, the memory layout is left as-is, but a second buffer of the same size is allocated for storing masks, which are then XOR-ed onto the plaintext.

The first approach can be implemented by reserving the high 8 bytes of each 16-byte encryption block for a random nonce, while the low 8 bytes are used for payload. When storing a value in this block, the nonce is incremented to ensure that the ciphertext changes. In addition, the old plaintext must be overwritten with a random value before storing the new plaintext, to keep the attacker from detecting consecutive writes of the same value. In the second approach, the nonces and the data are stored in separate locations, and the nonces are XOR-ed onto the data as a mask. On each memory write, the corresponding location in the mask buffer is resolved, the mask value is updated and then XOR-ed to the new plaintext. Finally, the masked plaintext is written to the desired memory address. As the nonces are high entropy values and updated independently of the written data, they are not susceptible to the dictionary attack or collision attack. Due to its high locality, the first approach is better suited for small variables (*e.g.*, variables on the stack), while the second approach has better support for pointer arithmetic and should thus be used for buffers and complex data structures. Both countermeasures could be implemented as a compiler extension, that automatically applies them to variables marked as secret.

C. Software-based Countermeasures: Kernel Context Switch

While the generic software-based countermeasures are sufficient to protect applications in user mode, they make the critical assumption that registers are immune to ciphertext side channels. However, our attack in Section IV shows that the attacker can inspect the ciphertext in the kernel's `pt_regs` structure to infer register values. To mitigate the ciphertext leakage on register-level, we developed a kernel patch that protects registers during context switches. We focus on the Linux kernel, but similar methods can also be applied to other operating systems.

Specifically, the kernel patch protects the `pt_regs` structure, which stores x86-64 user space registers as described in Section II-D. We present two methods for securing this structure. One is to insert a random nonce alongside each register. The other is to randomize the stack location on each context switch.

Storing a nonce alongside registers. A random 64 bits nonce can be stored next to each register (64-bit) to add enough randomization. In this way, on a context switch, the kernel doesn't simply push all registers to the stack, but interleaves them with pushes of a random value, which is incremented on every context switch. This method gives us 64 bits of security, which makes it impossible for the attacker to infer the plaintext even for long running VMs. However, this strategy comes with a major caveat: It requires significant changes to existing highly-optimized code paths, as a lot of exception/signal handling functions rely on the exact offset of the registers in `pt_regs` and would thus may not be adapted by the upstream kernel committee.

Context switch stack randomization. As an alternative strategy, we adapt the memory address randomization idea to the kernel entry point stack. Instead of inserting nonces between the saved registers, we randomize the address of the stack where the exception/interrupt handlers store the register values of the interrupted user space application.

This method is much less intrusive than the nonce approach and easy to hide behind a feature flag, as we only need to keep track of stack pages and replace the stack pointer on each exit from kernel space to user space. However, it also comes with a high memory overhead, as we have to reserve a lot physical memory only for the kernel entry point stacks. Also, at some point we will run out of physical memory, giving us a hard limit on the reachable entropy.

For example, if we assume that we have 8 GB of physical memory which can be freely used for our stack countermeasure, with a stack size of 4 KB (one page) we get 2^{21} possible stack locations (21 bits of entropy). This is significantly less than the 64 bits obtained with the nonce approach, but still considerably reduces the attack bandwidth, as the attacker would have to wait until a stack page repeats. To assess the practicality and the resulting overhead, we implemented the stack randomization countermeasure in the Linux kernel.

D. Case Study: Randomizing pt_regs Location

For our case study, we focused on the common exception and interrupt path described by `idtentry_body` which is defined in `arch/x86/entry/entry_64.S`. The `idtentry_body` path is *e.g.* used for the high frequency page fault exception as well as for the local APIC timer interrupt. The latter is especially interesting, as it is the main driver in determining if a task has used up its time slice, leading to a reschedule to a different task. While interrupts and exceptions can also occur when the CPU is already in kernel mode, we restrict our countermeasure to events that interrupt a user space application, as they contain the register values that we want to protect.

Since the thread stack is empty upon entering the kernel from user space, we can simply replace it with a newly allocated stack. For the entry stack, randomizing the stack upon entry to the kernel is more difficult, as all general purpose registers hold user data and thus cannot be used to perform the change. To circumvent this, we randomize the stack on the exit path before returning back to user space. Thus upon the next entry, we have a fresh entry stack.

Using the regular memory allocation mechanisms of the Linux kernel for the stack allocation proves difficult, as they were not build with guarantees regarding not returning a recently freed page upon a new allocation. In addition, they

share a common memory pool with the rest of the system, which increases the collision probability under high memory load, if taking random pages from the pool. Instead we allocate a large chunk of memory at boot time and manage the stacks in a first-in-first-out queue, maximizing the time between reuses.

To evaluate the performance of our prototype implementation, we call the `cpuid` instruction 10 million times in a tight loop from a user space application. Under SEV, this is an emulated instruction that will directly trigger the modified code paths in `idtentry_body` without doing further expensive computations, allowing us to efficiently measure the performance impact of the modifications to the context switch. Using this strategy, we measured a total average overhead of 1063 nanoseconds per context switch with standard derivation 4.93. We also ran a modified benchmark, where the application also loops over a large memory buffer each iteration, to measure the additional cache pressure created by randomizing the kernel stack. We ran the experiment 1000000 times resulting in a total average overhead of 2232 nanoseconds with standard derivation 297.

VII. Discussion

Secure encryption of large memory. Memory encryption is a basic building block used in TEEs to establish the confidentiality of data that leaves the CPU. Ideally, a probabilistic authenticated encryption scheme needs to be used, as was implemented for the first generation of Intel SGX [16]. However, managing and updating authentication tags and counter values consumes additional storage, costs latency and decreases the memory bandwidth for payload data. Thus, we do not believe that integrity trees can scale to protect large amounts of memory, as it is required for the confidential VM usage model.

To cope with these conflicting properties, many confidential VM designs use a mixture of cryptography and additional, architectural permission checks to achieve their security guarantees. Since random memory access latency is a critical performance property for the entire system, ECB would be the best candidate from a performance point of view. However, the independent encryption of all memory blocks with the same key leaks repetition patterns, as there is only one ciphertext for each plaintext. Thus, current confidential VM designs (AMD SEV [23]), but also designs to be commercially available in the near feature (Intel TDX [19] and ARM CCA [7], [8]) all adopt a tweaked block cipher, like AES XTS/XEX. Table II shows a more comprehensive overview. These modes offer a middle ground between performance and security, as the tweak mechanism offers a cheap way to ensure that the same plaintext encrypts to different ciphertexts when stored in two different addresses. However, for a given memory block, there is still only one ciphertext for each plaintext. As we have seen throughout this paper, this is the root cause of the ciphertext side channels.

To prevent attacks on the missing integrity protection, systems like SEV-SNP or Intel TDX and Intel SGX prevent untrusted parties from writing to protected memory [4], [13]. Intel TDX and SGX also prevent read accesses to the ciphertext [13], [19]. However, as discussed in Section II-C, these checks do not prevent physical attacks like bus snooping.

Finally, the implementation of access right checks also comes with technical hurdles. On the one hand, they need to be fast, as they influence the memory access latency. On the other hand, static approaches that simply block access to a fixed range, like in Intel SGX, hinder efficient memory use and scaling. These hurdles remain open research questions to be answered in the future works.

Side-channel resistant cryptosystems. With decades of studies on micro-architectural side channels, including cache or TLB side channels, building side-channel resistant cryptographic implementations has become a common practice. Most practically used cryptographic libraries adopt some levels of side-channel defenses, to prevent exploitation from a remote attacker [1] or another user on shared machines [39], [40]. The known best practice for defeating side channels is data-oblivious constant-time implementation, which dictates the execution time of the cryptographic operations (or an arbitrary portion of it) is constant regardless of the secret values used in the computation and that branch decisions or memory accesses may not depend on secret values. Data oblivious Constant-time implementation has been shown to defeat all known micro-architectural side-channel attacks, except the ciphertext side-channel attacks discussed in this work.

The ciphertext side channel opens up a new way of exploiting cryptographic code, which the data oblivious constant-time implementation is no longer sufficient to guard against. Given the difficulties of securing accesses to the ciphertext through memory access or bus snooping (Section II-C), we envision cryptographic code to be used in TEEs with large memory needs to adopt a new paradigm that achieves indistinguishability not only on execution time and access patterns, but on the ciphertext values. We hope our work will inspire a new research direction on secure implementation of cryptography, such as tools to automate the discovery of such vulnerabilities, compilers to transform a vulnerable code to a secure one, or formal provers to assert the absence of such vulnerabilities.

VIII. Related work

To protect SEV-protected VMs against an untrusted cloud service provider, SEV adopts some additional designs atop traditional Virtualization. Some of those adjustments are challenged, including *AES memory encryption*, the *I/O bounce buffer* and *ASID-based key management*. Meanwhile, some designs inherited from AMD's traditional hardware-based virtualization are also proven to be insecure under the assumption of the untrusted host, including the *VM control block*, *Nested Page Tables*, and *ASID-tagged TLB entries*. Besides the Ciphertext leakage caused by VMSA, this section summarizes other attacks against SEV.

Intercept plaintext in VMCB (SEV). The original SEV allows the adversary to intercept and manipulate register values inside the unencrypted VMCB. Several existing works exploit

Table II: Comparison of hardware memory encryption-based TEEs. Drop-In replacement means that applications do not need to be adjusted to work with the TEE. * denotes the release time of the whitepapers while the commercial machine is not available yet. † to our understanding only a recommendation for a possible instantiation.

Project	Vendor	Release	TCB type	TCB size	Drop-In replacement	Encryption mode	Block size
SEV [23]	AMD	2016	VM	No Limit	✓	XE or XEX	128-bit
SEV-ES [22]	AMD	2017	VM	No Limit	✓	XE or XEX	128-bit
SEV-SNP [4]	AMD	2020	VM	No Limit	✓	XEX	128-bit
SGX [13]	Intel	2015	Enclave	256 MB [18]	✗	AES-CTR + integrity + freshness	128-bit
SGX on Ice Lake SP [20], [21]	Intel	2021	Enclave	up to 1 TB	✗	XTS	128-bit
TDX [19]	Intel	*2020	VM	No limit	✓	XTS	128-bit
CCA [7]	ARM	*2021	VM	No limit	✓	AES XTS or QARMA†	128-bit†

the unencrypted VMCB vulnerability. Hetzelt *et al.* showed that the attacker could control the VM's execution and perform ROP attacks [17]. Werner *et al.* showed that the attacker can infer VM's instructions, fingerprint applications, and steal secret data [35]. From SEV-ES, registers are encrypted and stored in VMSA. For SEV-ES, an additional integrity check is performed on every VMRUN. For SEV-SNP, the RMP table restricts software's write access towards the VMSA area.

Manipulate Nested Page Table (SEV-ES). By changing the mapping between the guest physical address and the system physical address in the nested page table, the attacker can disturb the VM's execution and turn the VM's benign activities into malicious activities. In the SEVered attack [31], Morbitzer *et al.* showed that programs with a network interface (*e.g.*, web server) could be used to decrypt the VM's memory. Specifically, the attacker sends some file query requests to the webserver inside a SEV-enabled VM and then remaps the guest physical address belonging to those data files to some host physical addresses of private data. The private data will then be sent back to the attacker. The latest SEV-SNP mitigates this vulnerability by prohibiting the hypervisor from unauthorized NPT remapping.

Note that the hypervisor-controlled nested page table also results in a page-level controlled channel. The page fault controlled channel is widely used in numerous attacks against AMD SEV ([25], [27], [35], [36], *etc.*), and is used to infer the VM's activities and step its execution. SEV-SNP also suffers from this controlled channel. According to SEV-SNP's whitepaper [4], the page-level controlled channel is not in the scope of SEV-SNP's designed features.

Modify encrypted memory (SEV-ES). Before SEV-SNP, the hypervisor had write access to the VM's memory, which led to some delicate attacks ([10], [14], [36], *etc.*) that broke the integrity of SEV-enabled VMs by carefully overwriting their encrypted memory. Wilke *et al.* [36] improved the analysis of the encryption modes on Zen 1 Embedded CPUs, discovering the updated XEX encryption mode and extending the reverse engineering of the tweak function. Using the tweak values in combination with a known plaintext-ciphertext dictionary, they built malicious code gadgets by copying ciphertext blocks in memory. Based on that, they bootstraped an encryption oracle. From Zen 2 onwards these attacks are no longer possible due to an improved tweak function.

Tamper with the I/O bounce buffer (SEV-ES). Because of the encrypted memory, DMA is not directly supported in SEV. A shared bounce buffer (SWIOTLB) is then introduced for I/O traffic. For incoming I/O traffic, the guest VM copies the data from the bounce buffer to its private memory. For outgoing I/O traffic, the guest VM copies the data from the private memory to the bounce buffer. The memory copy activities give the attacker a chance to construct encryption and decryption oracles. Li *et al.* [26] showed that the attacker could overwrite I/O traffic to encrypt/decrypt the VM's memory stealthily. SEV-SNP or processors with XEX mode memory encryption can mitigate this attack.

ASID-based momentary execution (SEV-ES). In SEV, including SEV-ES and SEV-SNP, the Address Space Identify (ASID) is managed by the untrusted hypervisor. While ASIDs play some rather important roles in SEV-enabled VMs, including cache tagging, TLB tagging, and identifying the VM encryption keys, the hypervisor has the ability to modify a VM's ASID during the VM's lifecycle. SEV relies on a "Security-by-Crash" principle that an improper ASID always causes a meaningless VM crash, assuming good behavior of the hypervisor. Li *et al.* [25] exploited this improper principle and introduced the CROSSLINE attacks. The authors showed that the attacker could extract the victim VM's encrypted memory blocks by setting an adversary-controlled attacker VM and changing the attacker VM's ASID to the victim VM's ASID. Because of the lack of ASID checks, the hardware always tried to execute the VM directly, which enabled momentary execution and a time window for leaking secrets. Even though SEV-SNP still gives the hypervisor the permission of ASID management, the additional ownership check mitigates the CROSSLINE attacks by restricting read access from the attacker VM to the victim VM.

ASID-tagged TLB (SEV-ES). Li *et al.* studied the hypervisor controlled TLB flush problem in SEV and SEV-ES [28] and presented TLB poisoning attacks. A TLB control field inside the VMCB controls the TLB flush during VMRUN. The authors exploited the fact that the hypervisor can skip TLB flushes by intentionally clearing the TLB control field. By doing so, the attacker could breach the TLB isolation between vCPUs from the same VM. The authors showed that an SSH connection controlled by the attacker could reuse other SSH connections' TLB entries and bypassed the login

authentication. SEV-SNP adds a hardware-controlled TLB flush mechanism to mitigate this vulnerability.

Permutation agnostic attestation (SEV-ES). Wilke *et al.* [37] exploited that the attestation mechanism of SEV and SEV-ES was not able to detect permutations of the attested data in memory on a 16-byte granularity. They further showed how an attacker can use the ability to reorder code blocks to construct malicious code gadgets allowing to encrypt/decrypt arbitrary data. This attack is mitigated with SEV-SNP.

Voltage glitching attack (SEV-SNP). Buhren *et al.* studied a fault injection attack against AMD-SP, named voltage glitching attack [11]. Different from other works in this section, voltage glitching attack needs additional equipment (including a μController and a flash programmer) and real-physical access to SEV's machine. By inducing errors in AMD-SP's bootloader and implanting a malicious SEV firmware, voltage glitching attack are shown to be able to extract secrets used in SEV's remote attestation.

IX. Conclusion

In this paper, we have performed a comprehensive study on the ciphertext side channels. Our work extends ciphertext side-channel attack to exploit the ciphertext leakage from *all* memory pages, including those for kernel data structures, stacks and heaps. We have also proposed a set of software countermeasures, including patches to the OS kernel and cryptographic libraries, as a workaround to the identified ciphertext leakage.

As a general design lesson, deterministic encryption modes like XEX must be combined with both read and write protection to prevent software-based attacks. To also prevent physical memory attacks, freshness and integrity protection are required.

References

[1] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

[2] AMD. Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh. *Manual*, July 2018. Rev 3.03.

[3] AMD. AMD64 architecture programmer's manual volume 2: System programming. *Manual*, 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. AMD Secure Encryption Virtualization (SEV) Information Disclosure (Bulletin ID: AMD-SB-1013). https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1013, 2021.

[7] ARM. Arm CCA Security Model, August 2021. Rev 1.0, Document Number DEN0096.

[8] ARM. Arm Confidential Compute Architecture software stack. https://developer.arm.com/documentation/den0127/latest, 2021.

[9] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang. High-speed high-security signatures. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[10] R. Buhren, S. Gueron, J. Nordholz, J. Seifert, and J. Vetter. Fault attacks on encrypted general purpose compute platforms. In G. Ahn, A. Pretschner, and G. Ghinita, editors, *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 197–204. ACM, 2017.

[11] R. Buhren, H. N. Jacob, T. Krachenfels, and J. Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2875–2889. ACM, 2021.

[12] J. V. Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017.

[13] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

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

[15] Google. Introducing google cloud confidential computing with confidential VMs. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms, 2020.

[16] S. Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.*, page 204, 2016.

[17] F. Hetzelt and R. Buhren. Security analysis of encrypted virtual machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*, pages 129–142. ACM, 2017.

[18] Intel. 10th Generation Intel Core Processor Families. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf, July 2020.

[19] Intel. Intel Trust Domain Extensions. *Whitepaper*, 2020.

[20] Intel. Product brief, 3rd gen intel xeon scaleable processor for iot. https://www.intel.com/content/www/us/en/products/docs/processors/embedded/3rd-gen-xeon-scalable-iot-product-brief.html, 2021.

[21] S. Johnson, R. Makaram, A. Santoni, and V. Scarlata. Supporting intel sgx on multi-socket platforms. *Intel Corporation.[Online]. Available: https://www. intel. com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms. html*, 2021.

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

[23] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. *White paper*, 2016.

[24] D. Lee, D. Jung, I. T. Fang, C. Tsai, and R. A. Popa. An off-chip attack on hardware enclaves via the memory bus. In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 487–504. USENIX Association, 2020.

[25] M. Li, Y. Zhang, and Z. Lin. CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021.

[26] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected I/O operations in amd's secure encrypted virtualization. In N. Heninger and P. Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1257–1272. USENIX Association, 2019.

[27] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. CIPHERLEAKS: breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In M. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 717–732. USENIX Association, 2021.

[28] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference*, 2021.

[29] Microsoft. Azure and AMD announce landmark in confidential computing evolution. https://azure.microsoft.com/en-us/blog/azure-and-amd-enable-lift-and-shift-\\confidential-computing/, 2021.

[30] M. Morbitzer, M. Huber, and J. Horsch. Extracting secrets from encrypted virtual machines. In G. Ahn, B. M. Thuraisingham, M. Kantarcioglu, and R. Krishnan, editors, *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, pages 221–230. ACM, 2019.

[31] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. Severed: Subverting amd's virtual machine encryption. In A. Stavrou and K. Rieck, editors, *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*, pages 1:1–1:6. ACM, 2018.

[32] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 565–581. USENIX Association, 2016.

[33] P. Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In R. H. Zakon, J. P. McDermott, and M. E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 73–82. ACM, 2011.

[34] S. Swami and K. Mohanram. COVERT: counter overflow reduction for efficient encryption of non-volatlle memories. In D. Atienza and G. D. Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 906–909. IEEE, 2017.

[35] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In S. D. Galbraith, G. Russello, W. Susilo, D. Gollmann, E. Kirda, and Z. Liang, editors, *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, pages 73–85. ACM, 2019.

[36] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. Sevurity: No security without integrity : Breaking integrity-free memory encryption with minimal assumptions. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1483–1496. IEEE, 2020.

[37] L. Wilke, J. Wichelmann, F. Sieck, and T. Eisenbarth. undeserved trust: Exploiting permutation-agnostic remote attestation. In *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*, pages 456–466. IEEE, 2021.

[38] M. H. Yun and L. Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

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

[40] Y. Zhang, A. Juels, M. K. Reiter, and T. 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.