CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP

Stefan Gast*, Hannes Weissteiner*, Robin Leander Schröder†‡ and Daniel Gruss*
*Graz University of Technology, Austria
†Fraunhofer SIT, Darmstadt, Germany
‡Fraunhofer Austria, Vienna, Austria

***Abstract*—Confidential virtual machines (VMs) promise higher security by running the VM inside a trusted execution environment (TEE). Recent AMD server processors support confidential VMs with the SEV-SNP processor extension. SEV-SNP provides guarantees for integrity and confidentiality for confidential VMs despite running them in a shared hosting environment.**

In this paper, we introduce CounterSEVeillance, a new side-channel attack leaking secret-dependent control flow and operand properties from performance counter data. Our attack is the first to exploit performance counter side-channel leakage with single-instruction resolution from SEV-SNP VMs and works on fully patched systems. We systematically analyze performance counter events in SEV-SNP VMs and find that 228 are exposed to a potentially malicious hypervisor. CounterSEVeillance builds on this analysis and records performance counter traces with an instruction-level resolution by single-stepping the victim VM using `APIC` interrupts in combination with page faults. We match CounterSEVeillance traces against binaries, precisely recovering the outcome of any secret-dependent conditional branch and inferring operand properties. We present four attack case studies, in which we exemplarily showcase concrete exploitable leakage with 6 of the exposed performance counters. First, we use CounterSEVeillance to extract a full RSA-4096 key from a single Mbed TLS signature process in less than 8 minutes. Second, we present the first side-channel attack on TOTP verification running in an AMD SEV-SNP VM, recovering a 6-digit TOTP with only 31.1 guesses on average. Third, we show that CounterSEVeillance can leak the secret key from which the TOTPs are derived from the underlying base32 decoder. Fourth and finally, we show that CounterSEVeillance can also be used to construct a plaintext-checking oracle in a divide-and-surrender-style attack. We conclude that moving an entire VM into a setting with a privileged adversary increases the attack surface, given the vast amounts of code not vetted for this specific security setting.

I. Introduction

In cloud computing, customers run virtual machines (VMs) on the cloud provider's host hardware, with multiple tenants sharing the same host for execution, requiring trust in the provider. However, when processing privacy- or security-critical information, customers want to protect their data from untrusted cloud providers or compromised hypervisors. Confidential VM technologies like AMD Secure Encrypted Virtualization (SEV) [11], Intel Trust Domain Extensions (TDX) [45], or ARM Confidential Compute Architecture (CCA) [13] promise higher security by running the VM inside a trusted execution environment (TEE), designed to protect sensitive data from physical and privileged software access. In contrast to other TEEs, like Intel Software Guard Extensions (SGX) [46], or ARM TrustZone [14], which are designed to run only specialized, hardened code, AMD SEV and Intel TDX are designed to run an entire software stack supplied by the customer, including a fully-featured, general-purpose operating system. Customer software stacks tend to be complex and often consist of components that are not designed with the stronger attacker model in mind, where the physical host is compromised.

Numerous works [16], [77], [84], [91], [85], [80], [5], [88], [52], [55], [92], [98], [68], [69], [97], [95], [56] show attacks on TEEs. In particular, multiple works focus on side-channel attacks, leaking critical information from the processor's memory interface [91], processor caches [66], [78], [35], [17], [94], branch predictors [49], [43], memory disambiguation [65], and power consumption information [58]. Several of these attacks exploit that the threat model of trusted execution environments allows for much more powerful adversaries than a regular attack in a native or virtualized attack setting: For instance, Xu et al. [99] control the steps of an enclave by controlling the page mappings, effectively leading to fine-granular stepping through the victim program, comparable to attaching a debugger. Moghimi et al. [65] slow down the victim dramatically and amplify the leakage consequently. Van Bulck et al. [86] implemented a generic framework for single-stepping SGX enclaves that is now widely used in the scientific community. Skarlatos et al. [80] take this approach one step further by enabling the microarchitectural replay of operations inside enclaves. While the security research on SGX focused on attacking the workload inside SGX, research on AMD-SEV so far has not focused on the stronger attacker model but on breaking the security guarantees of AMD-SEV itself, e.g., insufficient encryption [40], [95], [27], [97]. Wilke et al. [98] implemented a single-stepping framework for AMD-SEV, similar to SGX-Step [86]. Same as for SGX, single-stepping on SEV enables more powerful attacks, e.g., leakage through the deterministic ciphertext channel [55], [52].

AMD explicitly states that SEV-SNP cannot protect against all possible side-channel attacks and shifts the responsibility

Network and Distributed System Security (NDSS) Symposium 2025
24 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241038
www.ndss-symposium.org

for additional side-channel protection to the owner of the VM [6]. They argue that modern cryptographic libraries are already hardened against side-channel leakage, e.g., using constant-time programming techniques, as most of these attacks do not rely on virtualization. However, attackers controlling the host can take advantage of their privileged position, using techniques such as letting a victim VM page-fault upon access to certain pages [99] or single-stepping a VM [86], [55], [98] to synchronize with the victim and increase the temporal resolution of their attack. This facilitates attacks that are impossible or difficult to perform in traditional, unprivileged settings. Thus, code deemed to be secure in these settings might turn out vulnerable in an SEV-SNP context. Furthermore, a complete virtual machine contains several off-the-shelf components with non-cryptographic code, possibly leaking critical information, e.g., inter-keystroke timings from user input. Consequently, hardening an entire VM against side-channel leakage, following constant-time principles, is likely not practical given the extreme performance overheads and also infeasible given the vast amount of code that has to be vetted. AMD specifies that they do not prevent hypervisors from tracking page accesses and performance counters. They argue that the sensitive information is the data, not the code executed and explicitly declare fingerprinting attacks based on performance and page fault monitoring to be out of scope [6].

In this paper, we introduce CounterSEVeillance, a novel side-channel attack leaking secret-dependent control-flow and operand properties from performance counter data observed while running SEV-SNP VMs. Our attack is the first to leverage performance counter side-channel leakage to recover secret data from SEV-SNP confidential VMs. This is a significant difference to prior work on Intel SGX enclaves, where performance counters are halted during enclave execution [78], [37]. CounterSEVeillance is based on a systematic analysis of performance counter events in SEV-SNP VMs: We analyze which performance counter events are observable for the hypervisor from outside confidential VMs, exposing the corresponding events to a potentially malicious hypervisor. Comparing the performance counter values from running nbench [63] natively and in an SEV-SNP VM shows that 228 performance counter events are observable from the hypervisor during execution of the confidential VM, effectively leaking side-channel information.

CounterSEVeillance records performance counter traces with an instruction-level resolution by single-stepping the victim VM using `APIC` interrupts in combination with page faults. While we analyze a total of 335 performance counter events, we exemplarily showcase the leakage from 6 of these performance counters, *Retired Instructions*, *Retired Branch Instructions* and *Retired Taken Branch Instructions*, *Div Op Count*, and *Div Cycles Busy*, in four attack case studies:

First, we show that CounterSEVeillance is precise and versatile enough to extract full RSA-4096 private keys from a single execution of an Mbed TLS [57] (version 3.5.2) signature process. For this purpose, CounterSEVeillance matches traces of the *Retired Instructions*, *Retired Branch Instructions* and *Retired Taken Branch Instructions* counters against a known binary. This allows us to precisely recover the outcome of any secret-dependent conditional branch within an SEV-SNP VM. Our single-trace attack successfully recovers all of our 10 evaluation keys within less than 8 min per key.

As a second case study based on the same performance counters, we present the first side-channel attack on time-based one-time password (TOTP) verification. With CounterSEVeillance, we recover 6-digit TOTPs with only 31.1 guesses on average from the COTP library [82], running in an AMD SEV-SNP virtual machine.

As a third case study based on these performance counters, we show that CounterSEVeillance can also leak the secret key from which the TOTPs are derived, due to the non-constant-time base32 decoder implementation of the COTP library.

Finally, as a fourth case study, with a focus on different performance counters, we show that CounterSEVeillance enables an attacker to construct a plaintext-checking oracle, enabling secret-key recovery from the Hamming Quasi Cyclic (HQC) key encapsulation mechanism (KEM) [2]. We use the *Div Op Count* and *Div Cycles Busy* performance counters, leaking information about the lengths of division results. Schröder et al. [76] have shown that the length of division results carries exploitable secret-dependent information on Zen 2. However, they reported a negative result on Zen 3, as they found the timing variations of the `div` instructions to be too subtle to observe any leakage. In contrast to their work, we are the first to demonstrate divide-and-surrender-style leakage on Zen 3, specifically when running the victim workload inside AMD SEV-SNP, using CounterSEVeillance.

In contrast to prior attacks on SEV that manipulated the state of the confidential VM [40], [95], [40], [27], [68], [53], [97], [102], CounterSEVeillance is a passive side-channel attack. Our attack works on fully patched AMD SEV-SNP systems, including AMD's *VMSA Register Protection* mitigation [9], [52] against CipherLeaks [55]. Hence, further mitigations against CounterSEVeillance are necessary and require trade-offs between multiple factors like security of host and guest, manageability of system load, and performance, orthogonal to mitigations presented in prior work.

In conclusion, our paper shows that AMD's assumption on the limited security impact of performance counter information, does not hold: With four case studies, we demonstrate that CounterSEVeillance attacks leak sensitive information from the SEV-SNP VM only by observing performance counters, despite AMD's declaration of performance counters as non-critical [6]. CounterSEVeillance also comes with the core insight that moving an entire VM into an environment that is exposed to a privileged adversary, introduces unforeseen security risks: The attack surface is greatly increased as any piece of the software stack, handling a variety of sensitive information, can be attacked.

Contributions. In summary, we make the following contributions:

- We perform a systematic analysis of performance counter events in native and SEV-SNP contexts. Our analysis shows

that 228 performance counter events are directly observable by the hypervisor on SEV-SNP VMs, resulting in our **novel CounterSEVeillance attack**.
- We demonstrate a CounterSEVeillance attack on an Mbed TLS RSA signature process. Our attack leaks a **full RSA-4096 private key** with only a single trace and within less than 8 min.
- We present the **first side-channel** attacks on the verification of time-based one-time passwords (**TOTPs**), implemented in the COTP library [82]. For a 6-digit TOTP, our performance-counter-based attack **recovers the TOTP** with only 31.1 guesses on average. We additionally present an attack **recovering the secret key** from which the TOTPs are derived.
- We present the **first divide-and-surrender-style leakage on Zen 3**, with a success rate of **99.5 %**, using our CounterSEVeillance plaintext-checking oracle on the HQC key encapsulation mechanism.

Outline. In Section II, we provide background and discuss related work on the security of AMD SEV. In Section III, we systematically analyze which performance counter events are observable for a hypervisor attacking SEV-SNP. In Section IV, we describe our framework. In Section V, we present our performance-counter attack on an Mbed TLS RSA signature process. In Section VI, we present our attacks on the time-based one-time password library COTP. In Section VII, we present our attack on HQC. In Section VIII, we contextualize insights of our work and discuss potential mitigations. We conclude in Section IX.

Responsible Disclosure. We disclosed our findings to AMD on April 29th, 2024. AMD confirmed our findings on May 22nd, 2024 and they are planning to publish a security brief (AMD-SB-3013) on October 14th, 2024. They are also planning to introduce additional protections via their PMC virtualization feature.

II. Background and Related Work

In this section, we discuss background on AMD confidential virtual machines (VMs), page-fault tracking, hardware performance counters, and interrupt-based single-stepping. We cover related works on the security of AMD SEV.

A. AMD Secure Encrypted Virtualization (SEV)

On a virtualization host, the hypervisor manages the VMs (guests) and the resources assigned to them. To accomplish this task, the hypervisor runs with a higher privilege level than the VMs. In traditional virtualization, this means that the hypervisor itself can access all resources of the VMs, including register values and memory. Confidential VM technologies are designed to protect VMs from the privileged hypervisor by running them in a *Trusted Execution Environment (TEE)*. In contrast to earlier TEEs, like Intel Software Guard Extensions (SGX) [46], or ARM TrustZone [14], confidential VM technologies run an entire operating system and multiple applications in the TEE, with a possibly much larger attack surface. AMD's confidential VM technology, *Secure Encrypted Virtualization (SEV)*, transparently encrypts and decrypts the DRAM content of VMs [48]. The hypervisor cannot trivially perform targeted modifications of the guest data, as any change in the ciphertext of a guest will lead to (ideally) unpredictable changes in the plaintext.

SEV-ES. The original incarnation of SEV did not protect the CPU state, *i.e.*, the register contents, from the hypervisor. Hetzelt et al. [40] showed that a malicious hypervisor can exploit this to read and write arbitrary memory inside the VM and even disable memory encryption entirely. Werner et al. [95] showed that the register contents obtained from targeted interruptions of the guest contain enough information to spy on TLS connections and perform application fingerprinting. Buhren et al. [18] showed that power glitching can trick the AMD secure processor into accepting a forged public key and successfully verifying and booting their self-signed payload.

With the *Encrypted State (ES)* extension, the VM control block is split into an unencrypted control area accessible to the hypervisor and an encrypted save area used by the CPU to automatically store and restore the CPU state. The original SEV-ES extension encrypted pairs of two 64 bit registers in single AES-128 blocks in the save area. Li et al. [55] showed that this allows the hypervisor to make fine-grained observations about changes in register states and even recover values of the registers. AMD addressed this issue by including a nonce in the encryption of the register state [52].

SEV-SNP. To mitigate malicious hypervisors corrupting the memory contents of a VM by overwriting encrypted memory or remapping guest memory pages, AMD introduced the *Secure Nested Paging (SNP)* extension [6]. SEV-SNP mitigates replay and remapping attacks [40], [27], [68] and the ciphertext reuse attack by Wilke et al. [97]. When accessing an invalidated page, the VM now receives an exception and can decide how to handle the invalidated page. The VM can *validate* the page, or it can deny this and, e.g., terminate with an error. In order to meet the desired integrity of SEV-SNP, the guest VM should never validate memory corresponding to the same guest physical address more than once [6]. Still, fault attacks exploiting architectural bugs or physical properties may yield full compromises of AMD SEV-SNP [102].

Wang et al. [92] demonstrated a software-based power side-channel attack on AMD SEV-SNP. By measuring the power consumption, they infer which instructions are executed by the VM and, in some cases, obtain information about the operands.

B. Page-Fault Tracking

Page faults are exceptions triggered when memory is accessed in a way that is not allowed for a specific memory region. Access permissions are configured in the page tables. Page faults are relatively common and are used to implement principles like on-demand page mapping or copy-on-write. In a traditional system, they are handled by the kernel. Attackers operating at that permission level have permission to access any data on the system. This means that, in the traditional model, any data leaked by memory accesses is insignificant

from a security standpoint. In contrast, with TEEs, the untrusted operating system or hypervisor is in charge of the page tables, and page faults become relevant side channels [99], [68], [53].

In SEV, the hypervisor can configure the access permissions for each virtual machine on a page-by-page basis via nested page tables. This is required since the hypervisor must be able to allocate pages to the guest physical address space dynamically. However, a malicious hypervisor can use those access permissions to track the execution of specific programs or entire operating systems.

For example, by unsetting the `present` bit on every page, every memory access on any page will cause a `vmexit` with a nested page fault, informing the hypervisor on which page was accessed. The hypervisor can determine access patterns with page size granularity by resetting the `present` bit for a page until the subsequent nested page fault occurs. While this technique is very slow, it allows an attacker to fingerprint the running software in a VM and find the required guest physical page numbers for subsequent attacks.

The attacker can start with more selective attacks when the required pages are known. Depending on the attack target, the attacker might only be interested in write operations, meaning they can remove the `writable` permission on the page. In the case of secret-dependent data access, they might only turn off the `present` bit for specific pages to recover the secret by observing the access patterns.

Another strategy is turning off the NX bit, which means the attacker will receive an interrupt when the guest tries to execute code from the target page. These interrupts can be used to find the entry point to a relevant part of the code without slowing down the system until the target code is about to be executed. The attacker can then switch to other attack primitives, like single-stepping, to have more fine-grained control over the execution.

C. Hardware Performance Counters

Hardware performance counters are CPU hardware registers that count certain hardware events, e.g., cache misses, branch mispredictions, and instructions executed. They can be used to determine application bottlenecks [72], identify resource-hogging applications or functions, and even detect some types of attacks [104], [50], [51], [25], [23], [19].

To monitor performance data, kernel-level code can select multiple performance-related events via the `PerfEvtSel` *Model Specific Registers (MSRs)*. This will cause the CPU to count the occurrence of the selected event in the corresponding `PerfCnt` MSR. The specific events that can be tracked depend on the processor family and can be found in the *Processor Programming Reference* for the specific processor [10].

To facilitate performance counter measurements from user space, Linux offers the perf subsystem [72]. Due to security concerns, the perf subsystem prevents unprivileged users from accessing the hardware performance counters by default [29], restricting their use in traditional, unprivileged attack scenarios. However, an attacker controlling the host operating system or hypervisor has full access to them.

Contrary to Intel's SGX [44], AMD's SEV does not disable performance counters when executing an SEV-enabled VM. Since SEV is developed for hosting confidential VMs on a (foreign) cloud provider, the cloud provider might have a legitimate interest in using performance counters to gain some insight into performance bottlenecks or to detect attacks from VMs. According to AMD [6], preventing application fingerprinting via performance counters is not in scope for SEV-SNP, since code is usually not confidential, but data is. In this paper, we demonstrate that the statement that performance counters can only leak information about the executed code, is false.

D. Interrupt-Based Single-Stepping

CounterSEVeillance requires precise control over the execution of the VM to inspect performance counters before and after the execution of single instructions. However, processor features commonly used for single-stepping regular applications, such as the single-step trap flag or hardware breakpoints, are disabled when executing a TEE in non-debug mode.

It is, however, possible to implement an (albeit less reliable) form of single-stepping by trying to interrupt the TEE shortly after allowing it to start, only allowing a single instruction to finish. Van Bulck et al. [86] have introduced the SGX-Step framework to single-step SGX enclaves. Similarly, Wilke et al. [98] have introduced SEV-Step for SEV-SNP VMs. These frameworks have been used in multiple prior attacks to gain instruction-granular control over the victim enclave [84], [85], [74], [16], [67] respective VM [102], [98].

Before transferring control to the VM via the `vmrun` instruction, SEV-Step arms the APIC timer to interrupt the core shortly after the VM resumes. The APIC timer interrupt causes a `vmexit`, transferring control back to the hypervisor. The specific timing depends on factors like the clock speed, CPU utilization, kernel- and hardware settings, and potential differences in the silicon. Since the hypervisor controls the system, the attacker can mitigate those issues by fixing the clock speed to a value known to work well, reserving an entire core for the target VM, and pinning the correct kernel and hardware parameters. The attacker can profile the hardware in advance to determine the timer interval and to account for any remaining variables.

Each attempt to single-step over an instruction in the VM can have one of the following outcomes: In the desired case, exactly one instruction is executed (single-step). If the VM is interrupted too early, no instruction is executed (zero-step) and the attacker retries. If the VM is interrupted too late, multiple instructions are executed (multi-step), reducing the temporal resolution of the attack.

SEV-Step uses the *Retired Instructions* performance counter to distinguish between zero-, single- and multi-steps. This method works even with *VMSA Register Protection* enabled and reports the exact number of instructions executed in a single `vmrun`.

The guest operating system relies on host-injected timer interrupts for preemptive task rescheduling. While single-stepping, the host does not forward timer interrupts to the guest. Consequently, inside the VM, the victim program is never interrupted or preempted, as the guest does not receive timer interrupts.

III. Systematic Analysis of Exposed Performance Counter Events

In this section, we present a systematic analysis of hardware performance counter events and show that various events leak from AMD SEV-SNP confidential virtual machines (VMs) to the hypervisor. We analyzed a total of 335 hardware performance events available under Linux 6.6.0 running on an AMD EPYC 7313P CPU. Out of these 335, we find that 100 events always return 0, regardless of our tests and regardless of whether they are observed for a native process or a confidential VM. Natively, we successfully observe activity on 226 events while running the nbench benchmark [63] (version 2.2.3). When running nbench in a confidential VM, we observe activity on 228 events. However, 7 events that show activity in the native setting show no activity while running the confidential VM. Vice versa, 9 events that show activity while running the confidential VM show no activity in the native setting. That is, almost all hardware performance events (219) that are observable in a native setting are also observable in the confidential VM setting, indicating the significant amount of information exposed to a potentially malicious hypervisor. In the following, we detail the setup for our systematic evaluation.

Experimental Setup. We compare the performance counter values from running nbench natively and inside a confidential VM. We choose nbench, since it performs a wide variety of individual, CPU-intense benchmarking tasks, like sorting, bit manipulation, data compression, and neural network computations, covering most of the performance counter events we can monitor.

First, we prepare a list of the available hardware performance counter events by selecting all events from the `perf list` output labeled as *Kernel PMU event*, resulting in 335 available events.

We then execute nbench natively on the host while monitoring each event with the Linux `perf stat` command. To reduce noise from other activity on the system, we pin the measurement to a fixed, isolated core. As we also want to record events triggered by kernel code, we do not restrict our measurement to userspace code. Our CPU has 6 hardware counters. Consequently, we can, at most, monitor 6 events together in a single run. However, some events require 2 hardware counters [10], requiring runs with even fewer events in parallel.

We finally repeat the measurement for an SEV-SNP VM on an isolated core running nbench. On the hypervisor (*i.e.*, outside the VM), we again monitor each performance counter. To constrain the measurement to only the VM, without recording any influences from the hypervisor, we use the `perf kvm stat` command with the `--guest` flag. This is explicitly supported by the hardware, by accordingly setting the `HG_ONLY` bitmask when programming the hardware counter [12]. Similar to the native measurement, this also captures events triggered by the guest kernel.

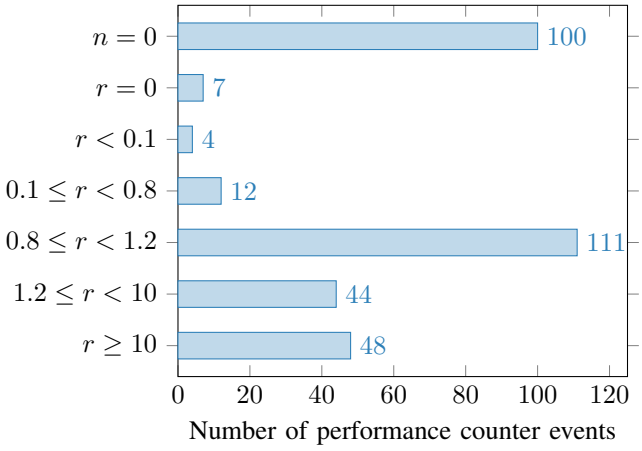


Fig. 1. Histogram from comparing performance counter values from running nbench natively with running it in an SEV-SNP VM. For 100 events, the performance counter value n is 0 after running nbench natively, *i.e.*, nbench did not trigger the event. For 7 events triggered natively, the performance counter value v is 0 after running nbench in the virtual machine, *i.e.*, the event was not observed. Out of the remaining 219 events observable for nbench running in a virtual machine, the ratio $r = \frac{v}{n}$ is between 0.8 and 1.2 for 111 of them.

Results. As stated initially, when executing nbench natively with `perf stat`, we observe that the benchmark triggers 226 out of 335 performance counter events in our list. When executing nbench within an SEV-SNP VM, we observe that this triggers 228 events. Out of the 226 events triggered natively, only 7 are not triggered by the VM.

Analyzing these 7 events in more detail showed that they have low counter values below 18 000, even when nbench is executed natively. Hence, we conclude that they are triggered by the natively running kernel and not by the benchmark itself. Among these events are *Interrupts Taken* and *Retired cpuid instructions*. When executing a VM, interrupts cause a `vmexit` for the guest and are counted for the hypervisor instead. Similarly, the `cpuid` instruction is intercepted and emulated by the hypervisor.

Additionally, we observe 9 performance counter events within the SEV-SNP VM, which are not triggered by nbench on the host. Among these events are `sse_avx_stalls` and various performance counters related to floating point operations. We assume this is caused by operations related to handling the confidential VM or other workloads inside the full Ubuntu installation running inside the VM, which trigger events that do not occur *on the isolated core* on the host.

Out of the 100 performance counter events that showed no activity, neither natively nor in the VM, 66 are `iommu` events. For instance, on our system, `perf` reports 4 groups of `amd_iommu` events, each with multiple specific performance events. However, 2 out of 4 groups do not report any data, accounting for 46 performance counter events. We hypothesize

that this might be due to the generic presence of the same performance counter events for one microarchitecture, regardless of the existence of the corresponding hardware components on the specific CPU model.

For each of the remaining 219 events triggered both natively and in the VM, we compute the counter ratio $r = v/n$ between the virtualized v and the native n execution of the benchmark.

For 111 events, the ratio r lies between 0.8 and 1.2, indicating that they are mostly unaffected by virtualization overhead. Among these are events like *Retired Instructions*, *Retired Branch Instructions*, *Retired Taken Branch Instructions* or *Retired SSE/AVX instructions*, caused by the computation-heavy CPU benchmark.

For 92 of the events, the ratio r is higher than 1.2, showing that they are positively correlated with virtualization overhead. Among these events are several events related to the L1 cache, likely caused by cache contention from other processes inside the VM, sharing the same core, which did not happen in the same way on our isolated core in the native setting. With nbench running in the VM, we also observe an increase in TLB accesses by approximately a factor of 5, which is explained by the longer page walk required for nested paging.

For 16 of the events, the ratio r is lower than 0.8, showing that they are negatively correlated with virtualization overhead. Interestingly, we see substantially fewer last-level cache hits and misses in the virtualized setting but a higher number of L2 cache hits and misses. This is surprising but indicates that the L2 cache is more successful when the workload is running inside a VM. This could be due to differences in the memory allocation, causing a different contention profile on the L1 and L2 caches.

Figure 1 summarizes our results. While attacks using other performance counters might also be possible, in the following, we select *Retired Instructions*, *Retired Branch Instructions*, *Retired Taken Branch Instructions*, *Div Op Count* and *Div Cycles Busy* as examples in our proof-of-concept attacks.

IV. The CounterSEVeillance Attack

CounterSEVeillance can reveal secret information if the victim executes any code that influences performance counters in a secret-dependent way. Since there are performance counters for a variety of microarchitectural elements, we can use CounterSEVeillance to leak the same information as side channels on these microarchitectural elements. We can leak the same information as side channels on, for instance, the cache [71], [100], branching logic [1], [30], the TLB [42], [36] or the integer divider [76]. However, in the SEV-SNP threat model with a privileged adversary, CounterSEVeillance leakage achieves a significantly higher resolution and accuracy: Instead of relying on side-channel measurements of a state change of the microarchitectural element, we have an architectural interface (performance counters) providing the desired information for each operation. Performance counters directly tell us whether the operation involved a cache hit or miss, a TLB hit or miss, or how many cycles a division required, or whether it involved a branch taken or not taken.

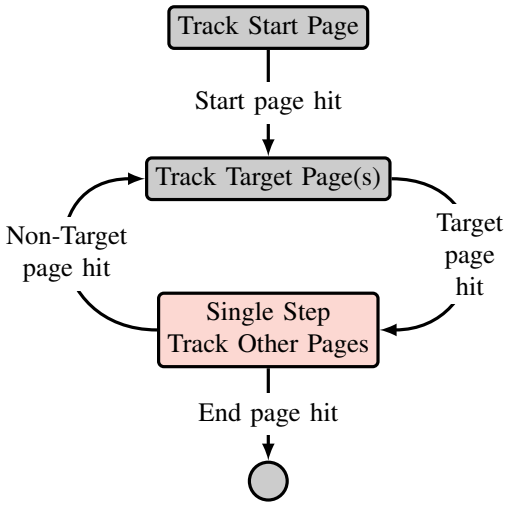


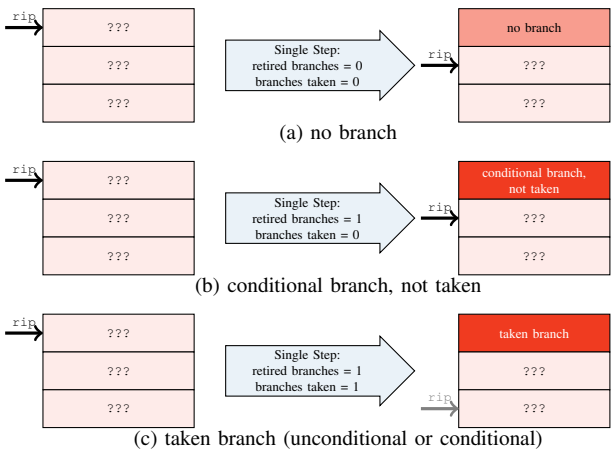
Fig. 2. Overview of the different states during recording. After the victim jumped from the start page to a target page, we start to single-step the VM. Single-stepping is paused when the victim calls a function irrelevant to the attack, residing on a non-target page. Single-stepping is continued when the called function returns to a target page and finally stopped when it jumps to the stop page.

In this section, we focus on branch outcomes as an illustrative example of the leakage of CounterSEVeillance, *i.e.*, we leak the secret data going into a branch decision. The attack consists of two phases. In the first phase, we generate a trace of the victim program, using an extended version of SEV-Step [98]. The trace includes information about all branch outcomes in the single-stepped section. In the second phase, we combine this data with knowledge of the executed code to recover data used for branching decisions. This second phase is completely offline and requires only a single trace. In Section VII we further show that CounterSEVeillance is not restricted to recovering branch outcomes and can also leak exploitable information about the results of `div` instructions.

A. Trace Recording

In the first phase, we record a trace of events while single-stepping through the attacked code in the victim SEV-SNP VM. In the following paragraphs, we outline how we use page-fault tracking to limit the single-stepping to the attacked code. We provide a detailed description of the start of single-stepping, the recording of events, the skipping of irrelevant functions, and the stopping of the recording.

Limiting Single-Stepping via Page-Fault Tracking. To synchronize attacker and victim, the attacker needs to know when the VM starts and stops executing the attacked code sequence. With page-fault tracking, we synchronize at a page-size granularity, minimizing the amount of code we have to single-step. For page-fault tracking, we identify the *target page(s)*, a *start page*, and an *end page*. The target page(s) contain the code we want to single-step. To prevent other code that might also be executed, residing on the same page(s), from spuriously activating single-stepping, we only start single-stepping when the victim jumps from the start



(a) no branch

(b) conditional branch, not taken

(c) taken branch (unconditional or conditional)

Fig. 3. Control-flow recovery by reading performance counters after each single-step: If the *Retired Branch Instructions* counter has not increased, the instruction executed was not a branch (a). If the *Retired Branch Instructions* counter has increased but *Retired Taken Branch Instructions* has not, the instruction executed was a conditional branch that has not been taken (b). If both counters have increased, the instruction executed was either a taken conditional or an unconditional branch (c).

page to a target page. We stop single-stepping when the victim jumps to the stop page. For example, when single-stepping an entire function, the start and end page is the page containing the caller of the function. Figure 2 illustrates the transitions between the different states during recording.

Li et al. [54] showed that it is possible to identify pages in memory by observing page access patterns, *i.e.*, for page-fault tracking, the attacker can learn the guest physical addresses of the start, end, and target pages. Thus, we assume that the attacker determined the correct pages in a previous profiling step in advance.

Starting Single-Stepping. We begin the attack by tracking execute accesses (*execute-tracking*) on the start page. Unless the victim executes code from this page, this is undetectable and has no performance impact. When the guest execution hits the start page, we track our target pages instead. We start single-stepping as soon as the guest jumps to a target page.

Recording Events from the Target Function. When we enter the target function, we start *execute-tracking* all guest pages except our target pages. We continue single-stepping while recording the increments of the *Retired Instructions*, *Retired Branch Instructions* and *Retired Taken Branch Instructions* performance counters. The *Retired Instructions* counter tells us how many instructions we missed in potential multi-steps. We use the *Retired Instructions* counter to recover from these cases during post-processing. As shown in Figure 3, from the *Retired Taken Branch Instructions* counter, we can directly derive whether the executed instruction was a taken (conditional) branch. The *Retired Branch Instructions* counter tells us whether an instruction was a not-taken branch or another instruction.

Skipping Irrelevant Functions. Whenever we receive a page fault on another page than our target, we stop single-stepping, untrack all pages, and start execute-tracking solely the target pages. This usually happens when the tracked code calls a function, which the compiler often places on different pages. Therefore, we record this event as a function call in our data. This allows irrelevant functions to run without any performance impact while minimizing the amount of unnecessary data recorded. When the called function returns, we receive a page fault on our target page, triggering the restart of the single-stepping while recording a function return event in our data. We also track all other pages again to prepare for the next function call or return.

Stopping Single-Stepping. We know our target code has finished when we receive a page fault on the stop page. We stop single-stepping and turn off all page-tracking, allowing the guest to continue running normally. We save the generated trace, which consists of single-step, multi-step, function call, and function return events, for offline post-processing.

B. Post-Processing of Single-Step Traces

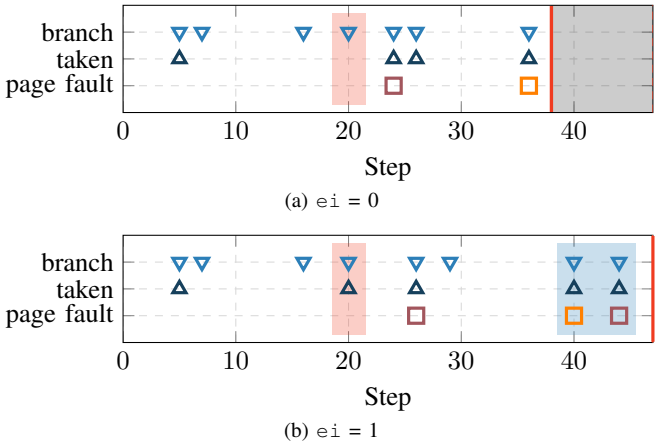
Once the data collection is complete, we can evaluate our results offline. We match the disassembled target program binary against the exact instruction patterns in the trace. Different compiler versions or optimization levels might cause changes, complicating the program flow reconstruction. We only partially automated the post-processing, where we still manually need to map some points in the trace to points in the disassembled code.

Matching Traces against the Instruction Pattern. By comparing the values of the *Retired Branch Instructions* and *Retired Taken Branch Instructions*, we can infer the branch history from the entire single-stepped execution with only a single trace. The execution of unconditional branches (e.g., `call`, `ret`, and `jmp`) will cause both performance counters to increment by one. We can use this fact for orientation, since the branch instructions create a repeating pattern in the data. We can also use the page-fault data as synchronization points to reliably align the trace with the instruction pattern from analyzing the binary. Since we know which instruction was executed in our single-step trace, we can find the single-step event corresponding to a conditional branch instruction and check the *Retired Taken Branch Instructions* counter for this point in time. If it is 1, we know the branch condition was true. If it is 0, we know the branch condition must have been false. While the post-processing for single-steps could already be applied as described above, we found it necessary to extend our post-processing to allow for multi-steps as well.

In our tests, there was a small chance that multiple instructions would execute in a single `vmrun`. We performed an empirical analysis of the multi-step probability. For an uninterrupted sequence of $1\,000$ instructions without function calls, we observed multi-steps in $19.3\,\%$ of the traces. Reducing this sequence to 500 instructions still incurred multi-steps in $13.4\,\%$ of the traces. Over all executed instructions, we observe a $0.02\,\%$ probability of a multi-step when attempting to perform a single-step. Thus, to allow our post-processing to handle these remaining $0.02\,\%$ correctly as well, we show how

```
1  for(;;) {
2    // ...
3    // ei contains the current bit of the exponent
4    if (ei == 0 && state == 1) {
5      MBEDTLS_MPI_CHK(mpi_select(&WW, W,
6          w_table_used_size, x_index));
7      mpi_montmul(&W[x_index], &WW, N, mm, &T);
8      continue;
9    }
10   state = 2;
11   nbits++;
12   exponent_bits_in_window |= (ei<<(windowsize-nbits));
13   if (nbits == windowsize) {
14     for (i = 0; i < windowsize; i++) {
15       MBEDTLS_MPI_CHK(mpi_select(&WW, W,
16           w_table_used_size, x_index));
17       mpi_montmul(&W[x_index], &WW, N, mm, &T);
18     }
19     MBEDTLS_MPI_CHK(mpi_select(&WW, W,
20         w_table_used_size, exponent_bits_in_window));
21     mpi_montmul(&W[x_index], &WW, N, mm, &T);
22     state--;
23     nbits = 0;
24     exponent_bits_in_window = 0;
25   }
26   // ...
27 }
```

Listing 1. The modular exponentiation loop of Mbed TLS' `mbedtls_mpi_exp_mod` function. By observing the outcome of the branching condition in Line 4, we can leak the private key bits.



(a) `ei` = 0

(b) `ei` = 1

Fig. 4. Event patterns for single iterations of the exponentiation loop. If `ei` is 0, the attacked branch instruction in Step 20 is not taken. If `ei` is 1, the attacked branch is taken, and a characteristic sequence of calls to `mpi_montmul` and `mpi_select` is executed. This sequence is used for reliable multi-step recovery.

we can still infer the same information as from a single-step in most cases in the following.

Recovery from Multi-steps. Given our attempt to single-step, multi-steps also involve only a small number of additional steps, *i.e.*, most likely a single missed branch. As we have precise performance counter information, we can often still infer the precise operations and secret inputs for the multi-step time frame.

If a multi-step contains only one unknown branch instruction, we can always recover the outcome of the unknown branch. From the *Retired Taken Branch Instructions* performance counter, we know the number b_1 of branches actually taken during the multi-step. From analyzing the binary, we know the number b_2 of other branches that must have been taken. By comparing these two numbers, we can reconstruct whether the unknown branch was taken: If $b_1 = b_2 + 1$, then the unknown branch was taken. If $b_1 = b_2$, then the unknown branch was not taken.

In the less likely case that a multi-step contains two or more unknown conditional branch instructions with branches of different lengths, we can still recover the outcome by matching the subsequently executed instructions (and page faults). Note that in the case of a multi-step, we still know precisely how many instructions were executed from the performance counter *Retired Instructions*, and the number of branches taken from the performance counter *Retired Branch Instructions*. Using the disassembled binary and the performance counter data, we can define constraints to identify all code paths that possibly could have been taken, e.g., using Ghidra [28], or more automated with angr [90]. If the constraints result in a single path, we have successfully recovered all branch outcomes. Otherwise, the attacker may include further constraints in the analysis, e.g., information about memory accesses, or other performance counters (e.g., *Div Op Count* or *Retired MMX/FP Instructions*) depending on the attack target. Still, in some cases, the analysis may yield multiple possible solutions. While this means we do not have the exact result, it tells us precisely which bits of the secret are unknown. This can be useful for a brute-force attack, but in some cases, it even allows for a direct derivation of the secret, e.g., for RSA, as the secret has to follow a specific structure [41].

Finally, the impact of multi-steps can be limited by page-faulting whenever the victim code jumps or calls into another page. This causes any multi-step to end and effectively acts as a synchronization point. Code that frequently calls other functions, therefore, is straightforward to attack with this method since the impact of a multi-step is negligible. Performance-optimized, concise code without function calls, such as a string-comparison loop, can be more challenging to attack since it consists of few instructions and does not jump or call into other pages. Still, we show in Section VI that attacks on such code are practical.

V. CounterSEVeillance Attack on RSA

In this section, we demonstrate that CounterSEVeillance is precise and versatile enough to recover full RSA-4096 keys from a single trace of an Mbed TLS [57] (version 3.5.2) signature process. Like prior work [78], [58], [32], we demonstrate an attack on the windowed square-and-multiply implementation of Mbed TLS. In contrast to multiple prior attacks on Mbed TLS RSA in trusted execution environments, our attack does not depend on noisy timing [78], [32] or power [58] side channels. Instead, our noise-free performance counter measurements facilitate fast key recovery within less than 8 min, emphasizing the additional threat from exposed performance counters.

Threat Model and Attack Setup. The attacker's goal is to steal the RSA secret key from the victim process running in an SEV-SNP VM on the attacker's host computer. The victim

process performs RSA-4096 signatures using the Mbed TLS library. The attacker knows that the victim runs an unmodified, up-to-date Mbed TLS library, e.g., version 3.5.2 in our case.[1] The attacker can obtain the guest physical addresses to which the victim binary is mapped via page tracking [54], enabling the attacker to detect the start and end of the signature process. Like prior works [78], [58], [32], we demonstrate our attack with a fixed window size of 1, since Liu et al. [59] have shown that attacks on window length 1 can be extended to arbitrary window lengths.

Attack Description. To recover the secret key, we target a key-dependent conditional branch in the `mbedtls_mpi_exp_mod` function, which is used to perform modular exponentiation in Mbed TLS. Listing 1 shows the relevant part of `mbedtls_mpi_exp_mod`. With a window size of 1, the condition in Line 13 will always be true and will be eliminated by the compiler when compiling with the default compiler flags from the Mbed TLS Makefile. Additionally, the compiler will eliminate the loop in Line 14.

The code loads the current bit of the key in `ei`. Line 4 decides on a code path based on `ei` and `state`. The `state` variable is used to skip leading zeroes and is never zero during the actual attack. Therefore, `ei` is the only unknown variable affecting which code path will be taken in each iteration.

To efficiently target the correct branching instruction, we find function calls to `mpi_select` in our generated trace. To leak the branch outcome, we iterate backwards through the recorded single-steps until we either find a branching instruction or another function call. The last conditional branch instruction before `mpi_select` is our secret-dependent target. We observe three patterns in the trace, as visible in Figure 4. In the first pattern, the observed branch is not taken. In our target binary, this means that `ei` was 0 (Figure 4a). In the second pattern, the observed branch is taken, which means that `ei` was 1 (Figure 4b). The third pattern (highlighted in blue in Figure 4b) is caused by consecutive calls to `mpi_montmul` in Line 17 and `mpi_select` in Lines 19 and 20, without a branch instruction in between. This pattern only appears when `ei` was 1 and is used for multi-step recovery.

Generating a full trace of a 4096-bit signature process requires about 200 000 single-steps. Due to the large number of executed instructions, our data also contains some multi-steps. However, since `mpi_select` and `mpi_montmul` reside on different pages than the target function, calling them causes page faults, limiting the extent of multi-steps. This leaves us with two cases, depending on the length of the multi-step.

If the multi-step only contains one conditional branch with an unknown outcome, we can directly recover its outcome as described in Section IV-B.

With a larger multi-step, this direct recovery is not possible anymore. However, we can detect the execution of the branchless code between `mpi_select` and `mpi_montmul` in Line 19. Since this pattern only occurs in the branch that gets taken if `ei` is 1, we can recover the branch information even in the presence of longer multi-steps. Thus, with this approach, we successfully recover the full 4096-bit RSA private key, without any bit errors, in a single execution trace.

Evaluation. We evaluate our attack on an AMD EPYC 7313P CPU, running Ubuntu 22.04.2 LTS as the host operating system and Ubuntu 22.04.2 LTS in the guest. We repeated the attack 10 times, each with a different private key. We successfully recovered the entire private key on every single attempt. The average recording time for each trace was 428.9 s, with a minimum of 420 s and a maximum of 439 s.

VI. CounterSEVeillance Attacks on TOTP Verification

In this section, we present two CounterSEVeillance attacks on a TOTP verification process. On a confidential VM, TOTPs might serve as a second authentification factor for login to the services running inside the VM, e.g., SSH or a web application. Our attacks exploit non-constant-time behaviour in the COTP library [82], at two different phases of the verification process:

We first demonstrate an attack on the comparison between the TOTP entered by the user and the expected, correct TOTP. Our attack guesses the correct 6-digit TOTP with 31.1 attempts on average, which is a significant reduction from the 500 000 average attempts required for a brute-force attack.

Finally, from a single execution of a TOTP verification, we also leak the secret key from which the TOTPs are derived. Consequently, having the secret key, the attacker can generate correct TOTPs offline at will and thereby completely bypass the second factor in this two-factor authentication setting. We recover the secret key within less than a second, with a success rate of 86 %.

A. Recovering Time-Based One-Time Passwords

Threat Model and Attack Setup. The attacker's goal is to guess the TOTP in a two-factor authentication running in an SEV-SNP VM on the attacker's host computer. The two-factor authentication is default-configured such that each TOTP has 6 digits and is valid for 30 s. The attacker is able to perform 2 TOTP guesses per second, *i.e.*, 60 guesses before the TOTP expires.[2] We assume the attacker can make side-channel observations about when the TOTP verification starts, e.g., via page-fault tracking [54].

For our proof-of-concept, we implement a victim program repeatedly asking for TOTP tokens and verifying them using the `totp_verify` function, provided by the COTP library. We deploy the victim program to the SEV-SNP VM. We prepare the host to perform CounterSEVeillance, as described in Section IV-A.

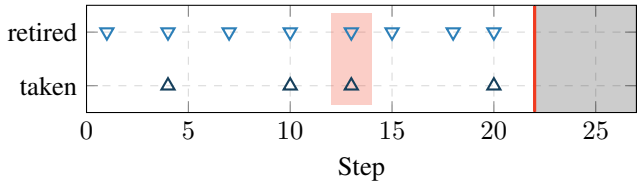
Attack Description. Our first attack is enabled by a secret-dependent conditional branch in COTP's `totp_compare`

[1] We actually found that the attack also works for slightly different library versions and compiler versions.

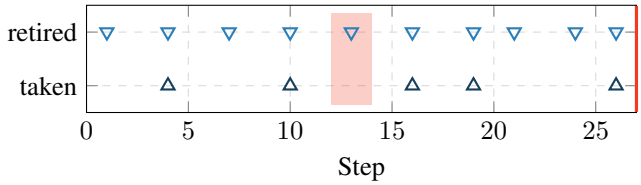
[2] Rate-limiting TOTP tokens is challenging as an attacker could exploit the rate-limiting as a denial-of-service attack on the actual user. Consequently, TOTP services typically implement no strict rate-limiting, in particular not for guesses from different IP addresses.

```
1  COTPRESULT totp_compare(OTPData* data, const char* key,
2      int64_t offset, uint64_t for_time)
3  {
4    char time_str[data->digits+1];
5    memset(time_str, 0, data->digits+1);
6    if (totp_at(data, for_time, offset, time_str) == 0)
7      return OTP_ERROR;
8    for (size_t i=0; i<data->digits; i++) {
9      if (key[i] != time_str[i])
10       return OTP_ERROR;
11   }
12   return OTP_OK;
13 }
```

Listing 2. COTP's `totp_compare` function is vulnerable to CounterSEVeillance, as the conditional branch in Line 9 leaks whether the guessed TOTP byte was correct.



(a) 2 correct digits



(b) 3 correct digits

Fig. 5. Event trace of the TOTP verification. The highlighted branch is taken on the first mismatching digit, allowing the attacker to guess the TOTP digit-by-digit.

function, which is called by `totp_verify` to compare a user-provided TOTP against the expected and correct TOTP for a given secret and time. Listing 2 shows the implementation of `totp_compare`. After obtaining the correct TOTP for the time given as `for_time` in Line 6, the function compares the user-provided TOTP `key` with it, by iterating over each digit in the loop starting at Line 8. In Line 10, the function performs an early exit on the first mismatching digit (Line 9).

With the *Retired Taken Branch Instructions* performance counter, the attacker can directly observe when the condition for the early exit is true, see Figure 5. This allows the attacker to guess each of the 6 digits of the TOTP individually, one after another. The attacker starts by probing the candidates 0, 1, 2 … 9 for the first digit. When the attacker observes that the early exit is *not taken* for the current candidate, the attacker knows that the guess for the digit was correct. The attacker then continues with the first digit fixed to the correct candidate and guessing the second digit, again observing whether the early exit is taken and continuing with the next guess if it is not. The attacker repeats this until all 6 digits and hence the entire TOTP is known. In the worst case, this requires 10 guesses per digit and, consequently 60 guesses in total. On average, we expect the attacker to succeed within 30 guesses.

To recover the length of the matching token, we require a section of 35 to 70 single-steps directly after a function call. In the case of multi-steps in the relevant part of our trace, we repeat the measurement. Due to the speed of the tracing process and the fact that the critical section of the trace is small, multi-steps did not have a significant impact on the reliability of our attack.

Evaluation. We evaluate our attack on an AMD EPYC 7313P CPU, running Ubuntu 22.04.2 LTS as the host operating system and Ubuntu 22.04.2 LTS in the guest. We repeat the attack 50 times, each with a different TOTP. In all 50 cases, we successfully guess the full TOTP within the 30 s validity timespan. Due to the fast tracing process, even with occasional multi-steps in the relevant section of the trace, we were able to repeat the measurements without impacting the reliability of the attack. On average, we require 31.1 guesses for a single TOTP, which is close to the expectation of 30 guesses. The full end-to-end attack (including the trace analysis), until the successful login takes 18.14 s on average.

Effectively, `totp_compare` performs a byte-by-byte comparison of the digits, with an early exit on the first mismatching digit, similar to the standard C function `memcmp`. Consequently, other applications or libraries using similar comparisons against a secret byte sequence might also be vulnerable to similar attacks, as they are not explicitly developed for usage in TEEs. For example, we found otplib [101], another TOTP library, implemented in TypeScript, to use a regular string comparison, which does not have constant-time guarantees.

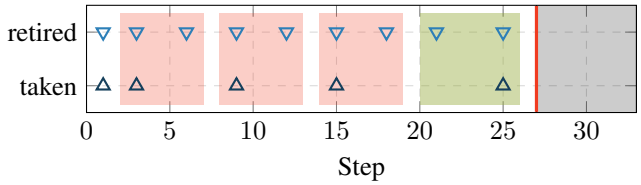
B. Recovering the Secret Key

Threat Model and Attack Setup. The attacker's goal is to leak the secret key from which the TOTPs for a target user in a two-factor authentication process are derived. Like in the previous attack, the two-factor authentication process runs in an SEV-SNP confidential VM on the attacker's host computer. The victim program again uses COTP's `totp_verify` function to verify the TOTPs. We again assume the attacker to be able to make side-channel observations about when the TOTP verification starts. However, for this attack, we do not assume the attacker to be able to perform multiple TOTP guesses.

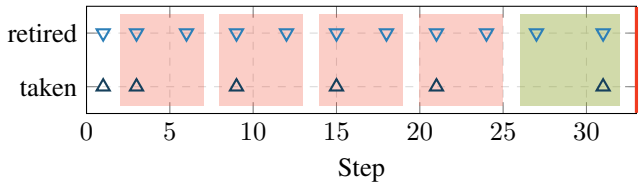
Attack Description. Our second attack is enabled by a secret-dependent conditional branch in COTP's base32 decoder. COTP internally stores the secret key in base32 form and decodes it on demand when a correct TOTP for a given time is needed to verify against. In particular, `totp_verify` calls `totp_compare`, which in turn calls `totp_now` and `otp_generate`, which invokes the base32 decoder `otp_byte_secret`. As depicted in Listing 3, the decoder iterates over the base32-encoded secret `data->base32_secret`, grouped into blocks of 8 base32 characters (Lines 10 and 12). To obtain the decoded value of each base32 character, the decoder linearly searches for the base32 character in the `OTP_DEFAULT_BASE32_CHARS` array (Line 15). As soon as the matching base32 character is found (Line 16), the corresponding index is stored as the decoded value (Line 17), and the search is stopped (Line 19). On our system, the compiler transforms the branch in Line 16

```
1  static const char OTP_DEFAULT_BASE32_CHARS[32] = {
2  'A','B','C','D','E','F','G','H','I','J','K',
3  'L','M','N','O','P','Q','R','S','T','U','V',
4  'W','X','Y','Z','2','3','4','5','6','7'
5  };
6
7  COTPRESULT otp_byte_secret(OTPData* data, char* out_str)
8  {
9    //...
10   for (size_t i = 0; i < num_blocks; i++) {
11     unsigned int block_values[8] = { 0 };
12     for (int j = 0; j < 8; j++) {
13       char c = data->base32_secret[i * 8 + j];
14       int found = 0;
15       for (int k = 0; k < 32; k++) {
16         if (c == OTP_DEFAULT_BASE32_CHARS[k]) {
17           block_values[j] = k;
18           found = 1;
19           break;
20         }
21       }
22       //...
23     }
24     //...
25   }
26
27   return OTP_OK;
28 }
```

Listing 3. COTP's `otp_byte_secret` function is vulnerable to CounterSEVeillance, as the search for the matching base32 character (Line 15) immediately aborts as soon as a match is found (Line 19).



(a) Base32 character is `D`



(b) Base32 character is `E`

Fig. 6. Event trace for the base32 decoder of the COTP library. Each loop iteration appears as two branches. The first branch is taken until the input character matches `OTP_DEFAULT_BASE32_CHARS[k]`. Consequently, the number of iterations and, hence, the base32 byte can directly be derived from the number of times the branch was taken.

into a `jnz` (jump if not zero) instruction, *i.e.*, the branch condition is inverted. Consequently, the number of times the branch is *taken* matches the index of the base32 character in the `OTP_DEFAULT_BASE32_CHARS` array. With CounterSEVeillance, an attacker can simply count how often the branch is *taken* for each base32-encoded character to obtain its index in the array and, consequently, its value. Figure 6 shows an example of this.

Evaluation. We again evaluate our attack on an AMD EPYC 7313P CPU, running Ubuntu 22.04.2 LTS as the host operating system and Ubuntu 22.04.2 LTS in the guest. We repeat the attack 86 times with a secret key consisting of 16 base32 characters. In 74 out of 86 cases, *i.e.*, with a success rate of 86 %, we recover the entire secret key within less than a second, without any errors. The remaining 12 cases failed due to multi-steps over multiple base32 characters, *i.e.*, multiple iterations of the loop starting in Line 12 in Listing 3. With the short time required for a single attempt, the attack can just be repeated until it finally succeeds in these cases.

Like with the string comparison in the TOTP guessing attack shown in Section VI-A, other libraries also have vulnerable base32 decoders. For example, otplib relies on the base32-decode NPM package [83], which performs the same linear search as COTP. We also found the widely deployed gnulib [31] to have its base32 decoder implemented as a cascade of branches, with a conditional branch for each possible input byte value.

VII. Divide and Surrender-style Attacks with CounterSEVeillance

Divide-and-surrender-style attacks [76] exploit variable division timings to recover secret keys from the Hamming Quasi Cyclic (HQC) [2]. HQC is a code-based key encapsulation mechanism (KEM) that has advanced to round 4 in the NIST effort for standardization of post-quantum cryptography. On a Zen 2 machine, Schröder et al. [76] observed timing variations for the loop shown in Listing 4 from a co-located attacker running on the sibling hardware thread on the same core as the victim. In the following, we show that a similar attack is also feasible on an SEV-SNP virtual machine running on Zen 3. In contrast to the attack by Schröder et al., our attack does not rely on contention of symmetric multiprocessing (SMP) resources. Instead, our attack single-steps the vulnerable code sequence and reads the *Div Cycles Busy* and *Div Op Count* performance counters after each single-stepping attempt, yielding the number of cycles the CPU was performing divisions and the number of divisions performed, respectively.

Threat Model and Attack Setup. The attacker's goal is to steal the secret key from the HQC decapsulation algorithm, running in an SEV-SNP VM on the attacker's host computer. The attacker is able to invoke the decapsulation multiple times, with chosen ciphertexts, e.g., via a web API. The attacker knows that the victim runs a version of the HQC reference implementation which is vulnerable to the attack by Schröder et al. , *i.e.*, the compiler generates `div` instructions from the code sequence in Listing 4. The attacker is able to obtain the guest physical addresses to which the victim binary is mapped via page tracking [54], enabling the attacker to detect the start and end of the decapsulation process.

Attack Overview. Divide-and-surrender-style attacks [76] rely on observing operand-dependent differences in the number of cycles the processor spends executing divisions. This timing side-channel is then used to construct a plaintext-checking oracle: Since the division operands depend on a seed that is derived from the encrypted message, different messages can be distinguished by their total division runtime. Such a distinguisher is sufficient for complete key recovery from HQC, since the attacker can now change ciphertexts in a way such that the distinguisher effectively reveals individual

```
1 for(size_t i=0; i<66; ++i)
2   rand_u32[i] = i + rand_u32[i] % (17668U - i);
```

Listing 4. The vulnerable code sequence in the HQC reference implementation. The modulo operation in Line 2 is compiled into `div` instructions with operand-dependent timings, enabling key-recovery.

bits of the secret key [76]. In the following, we show that the required operand-dependent division timings can be observed using CounterSEVeillance, on a Zen 3 CPU.

Division Timings on Zen 2 and Zen 3. Compared to the Zen 2 microarchitecture exploited by Schröder et al., the operand-dependent timing variations on Zen 3 are more subtle and much coarser-grained.

Whereas on Zen 2, a division takes an additional execution cycle for every 2 bits in the result [7], on Zen 3 only every 9 bits in the result add an extra cycle to the execution time [8]. Consequently, when dividing an unsigned 32 bit value by $17669 - i$, with $0 \leq i \leq 65$, as in Listing 4, on Zen 3, only two cases can be distinguished, by an increase of only a single cycle in execution time:

If `rand_u32[i]` is *below* $512 \cdot (17669 - i)$, then the result is *below* 512 and only has 9 or less significant bits. If `rand_u32[i]` is *equal or above* $512 \cdot (17669 - i)$, then the result is *equal or above* 512 and has 10 or more significant bits, resulting in an execution time larger by a single cycle. Due to the limited range of the 32 bit numerator `rand_u32[i]`, the maximum result is $\lfloor \frac{2^{32}-1}{17669-65} \rfloor = 243976$, with 18 significant bits, which does not increase the execution time further.

Additionally, the integer divider on Zen 3 can start a second division before the first one has fully completed [8], possibly concealing the already small timing difference between different operands. Furthermore, side-channel measurements typically are prone to measurement noise and temporal blind spots [75]. Consequently, Schröder et al. could not distinguish different numerators `rand_u32[i]` on Zen 3.

In contrast to this, by design, the *Div Cycles Busy* performance counter yields the exact number of cycles the CPU executed division operations, without measurement noise and blind spots. Using the Linux perf subsystem on our AMD EPYC 7313P CPU, we observe that the modulo operation in Line 2 of Listing 4 results in a *Div Cycles Busy* value of 7 for division results below 512 and in a value of 8 for division results equal or above 512, in line with AMD's documentation [8]. Consequently, by observing *Div Cycles Busy* after each single-step, we can distinguish these two cases.

Leaking Division Times via Performance Counters and Single Stepping. To leak the division times, we start single-stepping when the program calls the `vect_set_random_fixed_weight` function. We record both *Div Cycles Busy* and *Div Op Count* for 800 single-steps. When analyzing the resulting trace, we observe a specific pattern: *Div Cycles Busy* increments for up to 8 instructions before the `DIV` instruction is executed. We assume that this series of increments is caused by the pipeline already starting the division, but not committing the results since the execution is interrupted by a `vmexit`. Both counters seem to track hardware usage by the entire pipeline, instead of only counting usage of retired instructions. We can observe this behavior across some branching instructions, which confirms that the divisions are executed and tracked speculatively. Additionally, the first measurement in such a series of speculatively executed divisions consistently shows around double the expected number of busy cycles, which appears to correlate inversely with the result of the division instruction.

To distinguish between fast and slow divisions, we examine the *Div Cycles Busy* counter. When the performance counter measures 8 busy cycles, the result had $\geq$ 10bits. Conversely, we measure 7 busy cycles for smaller results. The position of the division measurements remains constant on each execution of the program. This allows us to determine the exact position of an expected division after an initial profiling phase. Because the *Div Cycles Busy* counter increments multiple times before the division instruction retires, we have multiple chances to measure the division timing. In our traces, we were able to measure each division between 4 and 8 times.

Occasionally, we measured higher than expected *Div Cycles Busy* values in our trace. Additionally, we saw some single-steps with more than one *Div Op*. We assume this is caused by the constant interrupts that occur during single-stepping, with potential µops still in flight. However, we only observed a maximum of one deviating value per measurement series. Because each division can be measured at least 4 times, we ignore these unexpected values and use the next single-step to determine our timing. Additionally, the possibility of measuring multiple times per division gives us a higher multi-step resilience.

Using the described techniques on an AMD EPYC 7313P CPU, we were able to distinguish large and small results for each division in `vect_set_random_fixed_weight` in 398 out of 400 (99.5 %) test runs. This suffices to build a plaintext-checking oracle that can distinguish a crafted ciphertext with a seed that generates fast divisions from random seeds. Using this plaintext-checking oracle we may employ prior art [89], [38], [39], [76] to break the security of the scheme and recover HQC secret keys.

VIII. Discussion and Mitigations

The working principle of SEV-SNP suggests that entire off-the-shelf VMs can be run under SEV-SNP, encrypted and secure against the outside world. The argument that developers should simply follow constant-time principles [6] may create a false sense of security: VMs contain software stacks that were not built for and not meant for constant-time computations – general-purpose code that is not trivial to make constant time. Even when attempting to audit an entire software stack for leakage of secrets, this easily reaches an order of magnitude of person-decades. It is unclear whether such a large-scale endeavor is feasible.

AMD states that performance counters may allow for fingerprinting but does not consider this a security concern as SEV-SNP does not protect against fingerprinting attacks [6]. However, we show that by monitoring performance counters

CounterSEVeillance can recover precise execution paths from confidential VMs: Any secret-dependent branch in the VM leaks information about the secret. We focused on illustrative and compact examples. However, CounterSEVeillance is generic and could equally be applied to confidential database systems, where sensitive data, e.g., medical data, could be leaked. We showed in Section VI that we can recover information byte-by-byte from a string comparison, in a schoolbook-like example (cf. [33]). Considering large code bases, it is likely that many code paths expose similar leakage if not in the source code then in the compiled binary [79], [73].

Numerous works attacked different variants of AMD SEV [18], [27], [40], [40], [55], [52], [68], [95], [97], [102]. These works exploit concrete implementation flaws in AMD SEV and, hence, can be mitigated without loss of functionality. In contrast to these works, we highlight a generic information leakage problem: Adjusting how performance counters are handled while SEV operation, would remove legitimate functionality and valuable information the host can use for debugging, load balancing, and accounting. Consequently, we see our attacks more in line with other generic attacks, such as the attack by Wang et al. [92].

Mitigations. CounterSEVeillance exploits that hardware performance counters provide precise information during the execution of confidential VMs. Combined with single-stepping and page-tracking, this enables attacks on cryptographic secrets and also general-purpose code. While this is a design choice for AMD SEV-SNP, Intel followed a different design for SGX, disabling most hardware performance counters on a processor core while it is running an SGX enclave in *production* mode. Although this does not prevent attackers from single-stepping the enclave and obtaining some information through side channels like page faults [99] or interrupt latencies [87], the attacker has to resort to less reliable side-channel information. Thus, a straightforward solution would be to disable performance counters while an SEV-SNP VM is running. This would mitigate our attack and also reduce the reliability of SEV-Step.

However, in a shared cloud environment, cloud providers may need to acquire statistics from VMs for load-balancing and billing purposes. Some works also proposed to use performance counters as a means to detect or mitigate attacks [19], [26], [62], [4], or to detect activity that violates the terms of service. For example, some providers restrict crypto mining [34], [64], [15]. Even in an encrypted machine, such activities are detectable using performance counters [61], [81].

Even worse, disabling performance counters, as with Intel SGX, opens the door to hiding malicious workloads, as multiple works have demonstrated [78], [47], [37], [93], [103]. Arguably, in the use case of confidential VMs, it is more realistic for a customer to spawn a malicious workload on a target system than with an SGX enclave.

AMD plans to mitigate leakage from performance counters with *Performance Monitoring Counter Virtualization* [12]. While this feature is not available on our Zen 3 and Zen 4 machines, it might be added in a future microcode update from AMD. Intel TDX already supports similar functionality [3]. With this feature, the processor automatically switches performance counter state (along with register state) when entering and leaving a virtual machine. While switching performance counter state prevents our attack, it also restricts legitimate monitoring by the hypervisor.

Concurrent work [60] suggests to insert additional instructions into the guest's execution stream to introduce noise into coarse-grained performance counter measurements. However, this mitigation would be ineffective against our attack, as single-stepping yields a measurement for each executed instruction, enabling us to filter out the additional instructions.

Single-stepping introduces significant delays to the execution of a program. In the context of Intel SGX enclaves, several works studied how an enclave could detect that it is being attacked [22], [21], [20], [70]. Given the lack of a trusted timing source, these works use counting threads. Given a trusted timing source, the enclave can monitor whether it is interrupted, as any interruption would cause jumps in the timer. We suggest a similar detection mechanism for confidential VMs. However, unlike Intel SGX, AMD SEV-SNP supports a feature called *SecureTSC* to provide confidential VMs with a trusted timing source. With SecureTSC enabled, the `rdtsc` instruction cannot be manipulated by the hypervisor. Consequently, the confidential VM could, without requiring a counting thread, use `rdtsc` to monitor whether there is an unusual frequency of interruptions, e.g., due to single-stepping or page-fault tracking. Similarly, future processors could also directly notify confidential VMs about interruptions [24].

Intel's TDX technology rate-limits interruptions of confidential VMs to mitigate attacks based on single-stepping [3]. However, a recent work [96] shows that these mitigations are insufficient to eliminate single-stepping attacks.

A recent, generic approach for improving the security of confidential VMs is Core Slicing [105], removing the need for an untrusted hypervisor. Core Slicing partitions the cores, memory and I/O devices, enabling the VMs to run directly on the hardware by enforcing isolation between the VMs in (trusted) hardware. While this would prevent single-stepping and performance counter reads, this would also prevent the cloud provider from detecting malicious workloads.

IX. Conclusion

In this paper, we introduced CounterSEVeillance, a new side-channel attack to reconstruct control-flow information and leaking operand properties from performance counter data. CounterSEVeillance is the first attack exploiting per-instruction performance-counter leakage on SEV-SNP virtual machines. Our systematic analysis showed that 228 performance counter events are exposed to a potentially malicious hypervisor while running confidential virtual machines. We record performance counter traces with an instruction-level resolution by single-stepping and match the resulting traces against binaries to precisely recover the outcome of any secret-dependent conditional branch. We presented four attack case studies, leaking a full RSA-4096 private key from Mbed TLS,

the first side-channel attack on TOTP verification, the first side channel on TOTP secret keys exploiting the non-constant-time base32 decoder implementation of the COTP library, as well as divide-and-surrender-style leakage on Zen 3. As our attacks work on fully patched AMD SEV-SNP systems, we conclude that new mitigations against CounterSEVeillance are necessary. Furthermore, we conclude that moving an entire virtual machine into a setting with a privileged adversary can significantly increase the attack surface, given the vast amounts of code not vetted for this specific security setting.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback. This research is supported in part by the European Research Council (ERC project FSSec 101076409), by the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85), and by the National Research Center for Applied Cybersecurity ATHENE as part of the PORTUNUS project in the research area Crypto. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.

[2] C. Aguilar Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J. Bos, J.-C. Deneuville, A. Dion, P. Gaborit, L. Jérôme, E. Persichetti, J.-M. Robert, P. Véron, and G. Zémor, "Hamming Quasi-Cyclic (HQC)," 2024. [Online]. Available: https://pqc-hqc.org/doc/hqc-specification_2023-04-30.pdf

[3] E. Aktas, C. Cohen, J. Eads, J. Forshaw, and F. Wilhelm, "Intel Trust Domain Extensions (TDX) Security Review," 2024. [Online]. Available: https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf

[4] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," *Cryptology ePrint Archive, Report 2017/564*, 2017.

[5] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port Contention for Fun and Profit," in *S&P*, 2019.

[6] AMD, "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," 2020. [Online]. Available: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[7] ——, "Software Optimization Guide for AMD EPYC 7002 Processors," 3 2020.

[8] ——, "Software Optimization Guide for AMD EPYC 7003 Processors," 11 2020.

[9] ——, "AMD Secure Encryption Virtualization (SEV) Information Disclosure," 2021. [Online]. Available: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1013.html

[10] ——, "Processor Programming Reference (PPR) for AMD Family 19h Model 21h, Revision B0 Processors," 2021.

[11] ——, "AMD Secure Encrypted Virtualization (SEV)," 2022. [Online]. Available: https://developer.amd.com/sev/

[12] ——, "AMD64 Architecture Programmer's Manual," 2024.

[13] ARM, "Arm Confidential Compute Architecture," 2024. [Online]. Available: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[14] ——, "TrustZone for Arm Cortex-M Processors," 2024. [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a

[15] AWS, "AWS Free Tier Terms," 2018. [Online]. Available: https://aws.amazon.com/free/terms

[16] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture," in *USENIX Security*, 2022.

[17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *WOOT*, 2017.

[18] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization," in *CCS*, 2021.

[19] S. Carnà, S. Ferracci, F. Quaglia, and A. Pellegrini, "Fight Hardware with Hardware: Systemwide Detection and Mitigation of Side-channel Attacks Using Performance Counters," *Digital Threats: Research and Practice (DTRAP)*, vol. 4, no. 1, pp. 1–24, 2023.

[20] G. Chen, M. Li, F. Zhang, and Y. Zhang, "Defeating Speculative-Execution Attacks on SGX with HyperRace," in *Dependable and Secure Computing (DSC)*, 2019.

[21] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: closing hyper-threading side channels on SGX with contrived data races," in *S&P*, 2018.

[22] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu," in *AsiaCCS*, 2017.

[23] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor," *Applied Sciences*, vol. 10, no. 3, p. 984, 2020.

[24] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "{AEX-Notify}: Thwarting precise {Single-Stepping} attacks through interrupt awareness for intel {SGX} enclaves," in *USENIX Security*, 2023.

[25] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *S&P*, 2019.

[26] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.

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

[28] C. Eagle and K. Nance, *The Ghidra Book: The Definitive Guide*. no starch press, 2020.

[29] J. Edge, "Disallowing perf_event_open()," 2016. [Online]. Available: https://lwn.net/Articles/696216/

[30] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *ASPLOS*, 2018.

[31] Free Software Foundation, "Gnulib – The GNU Portability Library," 2021. [Online]. Available: https://www.gnu.org/software/gnulib/

[32] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss, "SQUIP: Exploiting the Scheduler Queue Contention Side Channel," in *S&P*, 2023.

[33] T. Goodspeed, "A Side-channel Timing Attack of the MSP430 BSL," in *Black Hat USA*, 2008.

[34] Google, "Google Cloud Platform Terms of Service," 2024. [Online]. Available: https://cloud.google.com/terms

[35] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *EuroSec*, 2017.

[36] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security*, 2018.

[37] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *S&P*, 2018.

[38] Q. Guo, C. Hlauschek, T. Johansson, N. Lahr, A. Nilsson, and R. L. Schröder, "Don't reject this: Key-recovery timing attacks due to rejection-sampling in hqc and bike," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 223–263, 2022.

[39] Q. Guo, D. Nabokov, A. Nilsson, and T. Johansson, "Sca-ldpc: A code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2023.

[40] F. Hetzelt and R. Buhren, "Security analysis of encrypted virtual machines," *ACM SIGPLAN Notices*, vol. 52, no. 7, pp. 129–142, 2017.

[41] M. J. Hinek, *Cryptanalysis of RSA and Its Variants*. Chapman and Hall/CRC, 2009.

[42] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *S&P*, 2013.

[43] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX," in *CHES*, 2020.

[44] Intel, "Intel Software Guard Extensions (Intel SGX) Debug and Build Configurations," 2020.

[45] ——, "Intel Trust Domain Extensions," 2021. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf

[46] ——, "Intel Software Guard Extensions (Intel SGX)," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html

[47] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *SysTEX*, 2017.

[48] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," 2016.

[49] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security*, 2017.

[50] C. Li and J.-L. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters," in *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018.

[51] ——, "Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters," in *COMPSAC*, 2019.

[52] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A systematic look at ciphertext side channels on AMD SEV-SNP," in *S&P*, 2022.

[53] M. Li, Y. Zhang, and Z. Lin, "Crossline: Breaking "security-by-crash" based Memory Isolation in AMD SEV," in *CCS*, 2021.

[54] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected {I/O} operations in {AMD's} secure encrypted virtualization," in *USENIX Security*, 2019.

[55] 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 *USENIX Security*, 2021.

[56] ——, "TLB Poisoning Attacks on AMD Secure Encrypted Virtualization," in *ACSAC*, 2021.

[57] Linaro, "MBed TLS," 2024. [Online]. Available: https://www.trustedfirmware.org/projects/mbed-tls/

[58] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2021.

[59] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[60] X. Lou, K. Chen, G. Xu, H. Qiu, G. Shangwei, and T. Zhang, "Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels," in *DSN*, 2024.

[61] G. Mani, V. Pasumarti, B. Bhargava, F. T. Vora, J. MacDonald, J. King, and J. Kobes, "Decrypto pro: Deep learning based cryptomining malware detection using performance counters," in *Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020.

[62] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," *ACM SIGARCH Computer Architecture News*, 2012.

[63] U. F. Mayer, "Linux/Unix nbench," 2017. [Online]. Available: https://www.math.utah.edu/~mayer/linux/bmark.html

[64] Microsoft, "Azure free account," 2024. [Online]. Available: https://azure.microsoft.com/en-us/pricing/offers/ms-azr-0044p/

[65] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack against Constant-Time Crypto Implementations in SGX," in *CT-RSA*, 2018.

[66] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[67] D. Moghimi, "Downfall: Exploiting Speculative Data Gathering," in *USENIX Security*, 2023.

[68] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "Severed: Subverting AMD's virtual machine encryption," in *EuroSec*, 2018.

[69] M. Morbitzer, S. Proskurin, M. Radev, and M. Dorfhuber, "SEVerity: Code Injection Attacks against Encrypted Virtual Machines," in *WOOT*, 2021.

[70] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *USENIX ATC*, 2018.

[71] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[72] Perf Wiki, "Main Page," 2020. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[73] PQShield Ltd, "PQShield plugs timing leaks in Kyber / ML-KEM to improve PQC implementation maturity," 2024. [Online]. Available: https://pqshield.com/pqshield-plugs-timing-leaks-in-kyber-ml-kem-to-improve-pqc-implementation-maturity/

[74] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend," in *USENIX Security*, 2021.

[75] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.

[76] R. L. Schröder, S. Gast, and Q. Guo, "Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks," in *USENIX Security*, 2024.

[77] M. Schwarz and D. Gruss, "How Trusted Execution Environments Fuel Research on Microarchitectural Attacks," *IEEE Security & Privacy*, 2020.

[78] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

[79] M. Schwarzl, E. Kraft, and D. Gruss, "Layered Binary Templating," in *ACNS*, 2023.

[80] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling Microarchitectural Replay Attacks," in *ISCA*, 2019.

[81] R. Tahir, M. Huzaifa, A. Das, M. Ahmad, C. Gunter, F. Zaffar, M. Caesar, and N. Borisov, "Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises," in *RAID*, 2017.

[82] C. Tilkins, "GitHub – tilkinsc/COTP: A simple One Time Password (OTP) library in C, supports C++," 2023. [Online]. Available: https://github.com/tilkinsc/COTP

[83] L. Unnebäck, "Base32 Decode," 2017. [Online]. Available: https://github.com/LinusU/base32-decode

[84] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security*, 2018.

[85] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *S&P*, 2020.

[86] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Workshop on System Software for Trusted Execution*, 2017.

[87] ——, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *CCS*, 2018.

[88] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, 2019.

[89] G. Wafo-Tapa, S. Bettaieb, L. Bidoux, P. Gaborit, and E. Marcatel, "A practicable timing attack against hqc and its countermeasure," *Cryptology ePrint Archive*, 2019.

[90] F. Wang and Y. Shoshitaishvili, "Angr - The Next Generation of Binary Analysis," in *Cybersecurity Development (SecDev)*, 2017.

[91] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *CCS*, 2017.

[92] W. Wang, M. Li, Y. Zhang, and Z. Lin, "PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV," in *DIMVA*, 2023.

[93] S. Weiser, L. Mayr, M. Schwarz, and D. Gruss, "SGXJail: Defeating Enclave Malware via Confinement," in *RAID*, 2019.

[94] S. Weiser, R. Spreitzer, and L. Bodner, "Single Trace Attack Against RSA Key Generation in Intel SGX SSL," in *AsiaCCS*, 2018.

[95] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose, "The severest of them all: Inference attacks against secure virtual enclaves," in *AsiaCCS*, 2019.

[96] L. Wilke, F. Sieck, and T. Eisenbarth, "TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX," in *CCS*, 2024.

[97] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "SEVurity: No Security Without Integrity–Breaking Integrity-Free Memory Encryption with Minimal Assumptions," in *S&P*, 2020.

[98] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "Sev-step: A single-stepping framework for amd-sev," *TCHES*, pp. 180–206, 2024.

[99] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *S&P*, 2015.

[100] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.

[101] G. Yeo, "otplib: Time-based (TOTP) and HMAC-based (HOTP) One-Time Password library," 2021. [Online]. Available: https://github.com/yeojz/otplib

[102] R. Zhang, C. H. Center, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *USENIX Security*, 2024.

[103] Z. Zhang, X. Zhang, Q. Li, K. Sun, Y. Zhang, S. Liu, Y. Liu, and X. Li, "See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer," in *AsiaCCS*, 2021.

[104] H. Zhou, X. Wu, W. Shi, J. Yuan, and B. Liang, "HDROP: Detecting ROP Attacks Using Performance Monitoring Counters," in *ISPEC*, 2014.

[105] Z. Zhou, Y. Shan, W. Cui, X. Ge, M. Peinado, and A. Baumann, "Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud," in *USENIX OSDI*, 2023.