TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX

Fabian Rauscher
Graz University of Technology

Luca Wilke
University of Lübeck

Hannes Weissteiner
Graz University of Technology

Thomas Eisenbarth
University of Lübeck

Daniel Gruss
Graz University of Technology

Abstract

Intel TDX is a trusted execution environment (TEE) protecting arbitrary code, e.g., an entire OS, from the host system in trust domains (TDs). While TDX isolates the memory of TDs, side channels are still a threat due to shared hardware. Prior work showed that single-stepping is a powerful technique for attacking TEEs. After TDX was found vulnerable to these attacks, Intel improved their mitigations with TDX module version 1.5.06, stopping all known single-stepping attacks.

In this paper, we introduce TDXploit, a novel technique to revive single-stepping attacks on Intel TDX. TDXploit exploits a fundamental flaw in Intel's single-stepping mitigation, ironically, achieving a higher (>99.99 %) single-stepping accuracy than without mitigations. We recover the mitigation's internal state using an attacker-controlled TD. We not only predict the mitigation's behavior without any side channel but also manipulate it for reliable single- and multi-stepping. TDXploit can perform one single-step every 3.7 ms. We evaluate TDXploit with an attack on ECDSA in OpenSSL. Furthermore, we systematically evaluate 6 state-of-the-art side-channel attack techniques on TDX and their compatibility with TDXploit. A key finding is that `clflush` bypasses Intel's defenses, allowing Flush+Flush attacks on TDX guest physical memory. Compared to all previous Flush+Flush attacks, our Flush+Flush attack requires no shared memory and can target any memory location of a TD. We demonstrate the impact of this finding in a full key recovery on an AES T-Table implementation, requiring only 8 986 encryption traces. Finally, we combine our novel Flush+Flush with TDXploit to leak TOTP secrets with a single trace. We conclude that further mitigations against single-stepping and side channels on TDX are necessary.

1 Introduction

Outsourcing tasks involving confidential data, e.g., customer data and company secrets, to the cloud can be a threat to data confidentiality and privacy as the cloud provider has complete access to all the data when using traditional virtual machines (VMs). Therefore, a data breach in a cloud provider's infrastructure can have severe consequences for their customers. Trusted execution environments (TEEs) provide an environment where the host system does not have access to the memory and register state of the application. Traditional TEEs, e.g., Intel SGX, require applications to be written specifically for this environment [13], limiting its applicability to a narrow set of use cases. Using a library OS approach, prior work demonstrated that generic applications can be ported to Intel SGX [47], but there are still several limitations due to the process-scoped nature of SGX. The recently introduced AMD SEV-SNP and Intel TDX extensions are TEEs that allow entire VMs to run inside the secure environment, allowing the deployment of applications without any TEE-specific adaptations.

As the TEE threat model includes a malicious host, the attack surface of them is significantly larger than of non-TEE environments [13]. Intel SGX, in particular, received a significant amount of attention from the scientific community [53]. Early works focused on cache attacks [9,16,35,42,54], demonstrating that the isolation between the TEE and the rest of the system is limited due to shared hardware. A very powerful attack technique in the TEE threat model is single-stepping. The ability to single-step TEEs can be used to mount attacks such as instruction counting [36], determining executed instructions through timing [52], targeting specific code parts with side channels [15], or enabling new side-channel attacks that would otherwise suffer from too much unrelated code being executed [32, 35]. On SGX, Van Bulck et al. [51] use APIC timer interrupts to force enclave exits for single-stepping. Given these high-impact works, single-stepping is often considered the most powerful attack primitive on TEEs due to the instruction-granular control over the victim's execution.

Wilke et al. [56] demonstrated the first single-stepping attacks on AMD SEV-SNP VMs. However, Intel TDX includes a mitigation against single-stepping, which detects single-stepping attempts and steps a random number of guest instructions. To bypass this detection, Wilke et al. [55] de-

crease the core frequency, allowing them to, again, use the timer interrupt for the first single-stepping attack on TDX. Intel mitigated their attack with TDX module version 1.5.06 by introducing Instruction-Count Single-Step Defense (ICSSD). ICSSD utilizes performance counters for the single-stepping detection instead of the time-stamp counter, safeguarding the detection mechanism against CPU frequency manipulations [24, §17.3.2]. Wilke et al. [55] also present a second attack primitive that is not mitigated by ICSSD. However, it only leaks fuzzy information about the number of executed instructions and does not enable single-stepping. Thus, currently, there is no known single-stepping primitive for Intel TDX thwarting the most powerful attacks on TEEs.

In this paper, we introduce TDXploit, a novel single-stepping attack on Intel TDX that completely bypasses Intel's recent ICSSD defense. TDXploit exploits a fundamental flaw in Intel's single-stepping mitigation, which, ironically, results in a higher single-stepping accuracy (>99.99 %) than without any mitigations without relying on side channels. Our attack exploits two flaws in the random number generator (RNG) of the mitigation: First, we exploit that the RNG state is *per-core*, not per-TD. Second, we exploit that the state of the RNG can be reverse-engineered with only 32 samples. TDXploit continuously triggers Intel's single-stepping mitigation using an attacker-controlled TD to recover the RNG state and predict all future outputs, *i.e.*, the number of instructions executed by the next invocation of the mitigation. When the next output is '1', we schedule the victim TD, which then, ironically, is single-stepped by the mitigation itself. Afterward, we pause the victim until the RNG state allows for the next single-step. In contrast to prior work, this also enables controlled multi-stepping of TEEs for the first time.

We evaluate TDXploit in a microbenchmark on 2 end-to-end attacks and 6 state-of-the-art side-channel attacks. For this purpose, we systematically analyze the viability of 6 state-of-the-art side-channel attack techniques on Intel TDX, comprising several cache attack variants and the PortSmash attack. We find that 4 of the analyzed attacks work in host-to-guest attacks, *i.e.*, in the traditional TEE threat model. We also find that 2 of the analyzed attacks work in guest-to-host attacks, and 2 work in guest-to-guest attacks, *i.e.*, in the malicious TEE threat model, which has not been studied for Intel TDX before. During our analysis, we discovered that the `clflush` instruction ignores all the architectural isolation used by TDX, allowing the host to perform Flush+Flush attacks on the TD memory. Compared to non-TEE Flush+Flush attacks, this does not require shared memory and can target any TD memory location. Our results indicate a change in the microarchitecture on 5th generation Xeon Scalable CPUs, as prior work [1, 55] that primarily analyzed 4th generation CPUs found `clflush` ineffective for flushing TD memory. Unlike cache attacks demonstrated in these works, our `clflush`-based attack does not depend on parts of the coherence protocol that will most likely be abandoned with future CPU generations [24, §9.8]. We demonstrate the impact of our Flush+Flush attack as a standalone primitive by performing a full key recovery on the OpenSSL AES T-Table implementation that leaks the secret key after 8 986 encryptions.

Our microbenchmark of the single-stepping primitive shows that we can perform one single-step every 3.7 ms and successfully single-step in >99.99 % of cases. We perform an end-to-end attack on a time-based one-time password (TOTP) token generation library used in prior work on AMD SEV [15]. While their attack uses performance counter leakage that is already mitigated on Intel TDX, we use the Flush+Flush primitive in combination with TDXploit to leak TOTP secret keys with a single trace. In a second end-to-end attack, we re-create the attack on OpenSSL ECDSA by Wilke et al. [55] that Intel mitigated with ICSSD, showing again that TDXploit can reliably bypass Intel's mitigation. In line with previous results, we leak the full secret key by observing 33 biased signatures, which requires approximately 200 000 signature generations due to the low probability of the event that introduces the bias.

In summary, our work makes the following contributions:

- We introduce TDXploit, exploiting a fundamental flaw in Intel's single-stepping mitigation that not only can be bypassed but even yields a significantly more reliable single-stepping attack than without any mitigation.
- We systematically analyze 6 state-of-the-art side-channel attack techniques and their effectiveness with Intel TDX in host-to-guest, guest-to-host, and guest-to-host attack scenarios.
- We discover that `clflush` works on Intel TDX private memory and demonstrate a Flush+Flush attack on TDX guest physical memory, *i.e.*, we do not require shared memory and can attack all memory of the TD.
- We mount 2 end-to-end attacks with TDXploit, leaking TOTP secret keys with a *single trace* and re-enabling the previously mitigated attack on OpenSSL ECDSA.

Outline. We provide background in Section 2. In Section 3, we introduce and evaluate our novel single-stepping attack TDXploit. Next, we show Flush+Flush on TDX private memory and the potential of the attack in Section 4. Afterward, we evaluate the applicability of several other state-of-the-art side-channel attacks on TDX in Section 5. We discuss our results and their impact in Section 6. Finally, we conclude in Section 7.

Responsible Disclosure. We responsibly disclosed our findings to Intel on November 27, 2024. Intel confirmed the issue on January 22, 2025. They plan to publish the vulnerability in August 2025.

2 Background

In this section, we first discuss different trusted execution environments (TEEs). We then briefly discuss side channels, in particular in the context of TEEs. Finally, we discuss single-stepping as a powerful primitive for attacks on TEEs.

2.1 Trusted Execution Environments

TEEs are specialized environments that offer increased confidentiality and integrity guarantees, even against privileged or physical access [4, 7, 23, 24]. TEE technology is often seen as split into two generations: The first generation targets primarily personal computers, with the goal of protecting application components from a malicious user or compromised system. This is particularly interesting for storing and handling sensitive data, e.g., fingerprints or cryptographic material, or for digital rights management (DRM). Widely used examples are Intel Software Guard Extensions (SGX) [23] and ARM TrustZone [8], which have been practically deployed on millions and billions of devices. The second generation targets primarily cloud servers, with the goal of protecting entire VMs from a malicious or compromised host. These VMs are also called confidential virtual machines (CVMs). Practically deployed are AMD Secure Encrypted Virtualization (SEV) [6] and Intel Trust Domain Extensions (TDX) [22]. However, Arm also published a draft for their own TEE for the cloud scenario called Confidential Compute Architecture (CCA) [7]. Confidential virtual machines do not require writing code specifically for the TEE; instead, they allow the entire VM to run without modification in a secure environment.

2.2 Intel TDX

Intel Trust Domain Extensions (TDX) is Intel's second generation TEE, which allows running entire VMs as TEEs on Intel CPUs [24]. These guests are also referred to as trust domains (TDs). Guest memory and stored guest state are encrypted and managed by the TDX module in the trust domain virtual processor state area (TDVPS). The TDX module, an open-source software module signed by Intel, runs in the new SEAM root execution mode protected from the host and handles interactions between the TEE and the host, e.g., to create VMs, map pages, and run VMs. For VM management, Intel TDX uses the existing Intel virtual machine extension (VMX). To protect the guest's memory while still allowing for fast communication with the host, the guest's physical memory is split into a private encrypted part, only accessible by the guest and the TDX module, and a shared part, accessible by the host and the guest. The page tables of the private memory are managed by the TDX module, while the host manages the page tables of the shared memory. The guest can differentiate between shared memory and private memory through a bit in the guest physical address, making it possible for the guest to only interact with unsafe shared memory when it intends to. Furthermore, shared memory is only readable and writable for the guest but not executable to avoid certain bug types. Intel TDX makes use of Intel's Total Memory Encryption - Multi Key (TME-MK) to encrypt the memory of TDs. To this end, the key ID (HKID) range of TME-MK is split into a public and a private range, with the private range being exclusive to the TDX module and TDs. To protect the private memory against corruption, e.g., through Rowhammer, each 64-bit memory region can be protected with a cryptographic MAC that is automatically validated on each memory access. If the integrity check fails, an exception is thrown. Still, the underlying hardware of the CPU itself and the memory subsystem are shared across mutually untrusted TDX guests as well as between the TEE and any host workloads.

2.3 Attacks on Trusted Execution

Both practically and scientifically, TEEs are an attractive attack target. The reason for this is the combination of holding the most valuable security assets on the one side and, on the other hand, operating in a threat model that permits high-privilege adversaries, e.g., attacks from the kernel level. Consequently, numerous attacks have been published attacking TEEs. The first side-channel attacks on SGX were cache attacks on SGX enclaves by Brasser et al. [9] and Götzfried et al. [16], as well as cache attacks from SGX enclaves by Schwarz et al. [42]. Weiser et al. [54] demonstrated an attack on RSA key generation in SGX. Moghimi et al. [35] showed that the SGX threat model even allows amplifying cache side channels, given the ability of the host to control enclave execution. Wang et al. [53] presented a systematic analysis of many side channels in the context of SGX enclaves. Evtyushkin et al. [14] and Huo et al. [20] exploited the pattern-history table (PHT). Similarly, Lee et al. [28] presented a side channel exploiting collisions in the branch-target buffer (BTB). Both the PHT and the BTB were later on also exploited in Spectre attacks [27], e.g., by Chen et al. [11] on SGX enclaves. Other works demonstrated software-based power side channels [32], controlled channels [57], and the interrupt side channel [51]. Lou et al. [33] and subsequently Gast et al. [15] investigated leakage from AMD-SEV CVMs through performance counters. Gast et al. [15] demonstrated several attacks, including the recovery of RSA keys as well as the leakage of TOTP tokens and TOTP keys. Similar to many other works, they relied on precise single-stepping of TEEs, as we detail in Section 2.4. Fewer works explored the possibility of attacks from the inside of TEEs [17, 26, 42, 44], mainly focusing on side channels and Rowhammer attacks.

2.4 Single-Stepping Trusted Execution

Crucial for the success of many attacks on TEEs is the possibility to reliably single-step [11, 15, 32, 38, 43, 48, 49, 55, 56, 59].

Hence, single-stepping frameworks have been published both for Intel SGX and AMD SEV [51, 56], building a cornerstone for many sophisticated attacks [15, 32, 49, 55, 56, 59]. Van Bulck et al. [51] used the APIC timer to interrupt SGX enclaves continuously after each executed instructions. Wilke et al. [56] implemented a similar mechanism for AMD SEV. In response, Intel's TDX module contains a mitigation against single-stepping attacks [24]. The original version of this mitigation uses the timestamp counter and instruction pointer differences to detect single-stepping attempts. If TDX detects a single-stepping attempt, it masks all external interrupts and single-steps the guest for a random number of instructions before returning control back to the host. Wilke et al. [55] bypassed this initial single-stepping mitigation in Intel TDX by reducing the CPU frequency and counting VM entries through a side channel. Consequently, with TDX module version 1.5.06, Intel improved their defense by using performance counters to more effectively detect single-stepping attempts. While this mitigates the single-stepping attack by Wilke et al. [55], it still leaks information about the number of instructions the mitigation executed in the TD.

3 TDXploit

In this section, we present TDXploit. First, we detail the inner workings of the Intel TDX single-stepping mitigation. Second, we describe our novel single-stepping attack TDXploit and how it exploits the mitigation. Third, we evaluate TDXploit by evaluating it against a synthetic target and by recreating a previously demonstrated end-to-end attack against the ECDSA implementation of OpenSSL. Lastly, we discuss possible mitigations for TDXploit.

3.1 Single-Stepping Mitigation

To combat single-stepping, Intel TDX includes a mitigation as part of the TDX module [25]. The mitigation consists of two parts: the detection of a single-stepping attempts by the host and the introduction of noise in case of a detected attack.

For detection of single-stepping attacks, the mitigation focuses on interrupt-based VM exits (external interrupts, non-maskable interrupts, system management interrupts, and INIT interrupts), as past single-stepping attacks are based on triggering an external interrupt after one instruction in the guest is executed [51]. Unlike synchronous VM exits, such as VM calls, interrupts can occur at any time and are often required to be handled as soon as possible. Otherwise, it might result in the slowdown of other cores, e.g., for TLB shootdowns, or the slowdown of external devices. Hence, interrupts are used in existing single-stepping attacks as a simple way to force the control back to the untrusted host at the attacker's will. On VMX, an interrupt-based VM exit would normally result in control being forwarded back to the host, which then handles the interrupt. However, with TDX all VM exits return control to the TDX module instead. The TDX module will eventually invoke the untrusted host to handle the exit. Intel TDX employs two methods to detect malicious interrupts. Which method is used depends on the guest configuration. The first method uses a heuristic to detect potential attacks. If the last VM entry occurred less than 2 µs to 3 µs ago and the instruction pointer changed by less than 32 B (two times the maximum length an instruction in x86 can have), the TDX module assumes an attack is in progress. The time required for a VM entry and a VM exit is highly dependent on the CPU and the frequency the CPU is running on, with the latter being fully controllable by the untrusted host. Therefore, this detection approach can be bypassed on CPUs that take more than 2 µs to 3 µs to execute a VM entry and exit on their minimum operating frequency [55].

The second detection method, Instruction-Count Single-Step Defense (ICSSD), is Intel's response to the aforementioned attack. ICSSD takes advantage of performance counters to determine the number of instructions executed by the guest since the last VM entry. If this number is too low, it decides that an attack is occurring and triggers the mitigation. While ICSSD is significantly more reliable than the heuristic-based approach, it is only activated if the guest is not allowed to use performance counters. Otherwise, the mitigation falls back to the previously described heuristic.

When a potential attack is detected by the TDX module, noise is introduced to thwart the attack. This noise consists of a random number of instructions, between 1 and 32, being executed in the guest. The random number is generated through a per core 32 bit LFSR (linear-feedback shift register), initialized only once when the TDX module is loaded. LFSRs output the least significant bit of their state and generate new bits through an XOR of multiple bits at fixed positions of their internal state. To ensure that the correct number of instructions are executed in the guest, TDX uses the VMX monitor flag to single-step the guest. During this process, all external interrupts are masked, and control is **not** returned to the untrusted host. Consequently, the host is unable to determine the exact number of instructions executed within the guest, effectively mitigating single-stepping-based attacks [25].

Unlike previous single-stepping attacks on TEEs, such as TDXdown [55] and SGX-Step [51], TDXploit does not rely on cache invalidation for reliability. Therefore, prefetching address translations and other information every time a guest continues, as proposed by the AEX-Notify [12] mitigation for SGX, would not mitigate TDXploit. Informing the guest about frequent single-stepping attempts would allow the guest to react accordingly to a potential attack, but in itself does not prevent single-stepping.

3.2 Threat Model

We assume the classical TEE threat model of a compromised host [38, 43, 48, 51, 55, 56, 59]. Specifically, we exploit that the

```
1: xor rax, rax;      \\TDCALL leaf VMCALL
mov rcx, 0xff00;      \\TDCALL register bitmap
mov r11, 42;          \\VMCALL code (custom)
tdcall;
.rept 32;
add qword [rsi], 1;   \\add to shared memory
.endr;
jmp 1b                \\reset
```

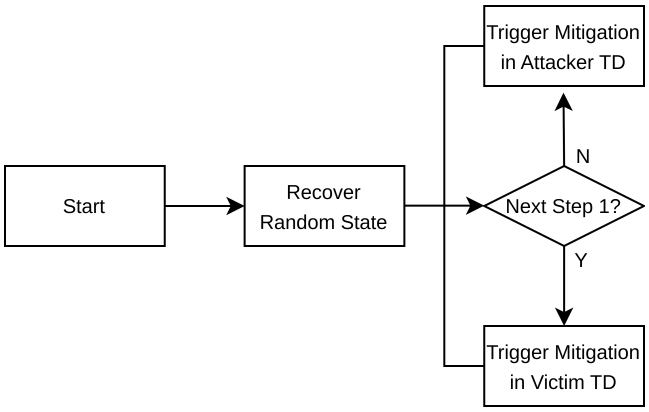
Listing 1: Attacker TD assembly loop to leak the LFSR state.

host has full control over TD scheduling, can launch its own host-controlled TDs, and is able to arbitrarily trigger external interrupts, e.g., through the APIC timer. The CPU runs the most recent version of the TDX module at the time of writing (TDX module 1.5.06 [25]). The victim guest enforces that the ICSSD method for single-stepping detection is enabled and that hyper-threading is disabled. These assumptions follow the TDX threat model provided by Intel [22], which assumes a compromised cloud provider.

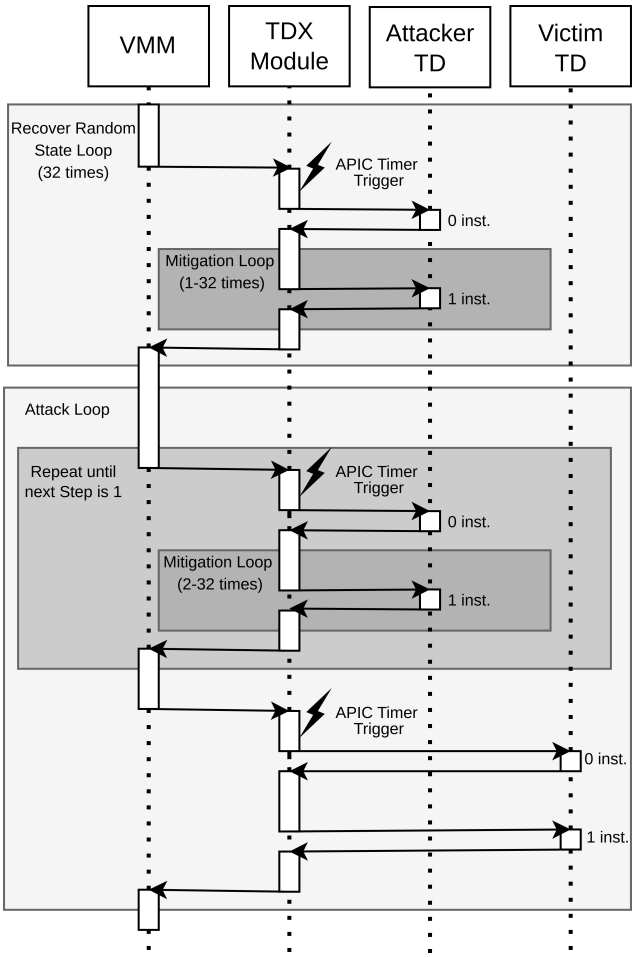
3.3 Attack

A high-level overview of our TDXploit attack is provided in Figure 1 as a flow chart in Figure 1a and a sequence diagram in Figure 1b visualizing the internal details of the attack. Contrary to previous single-stepping attacks on TEEs [51, 55, 56], TDXploit exploits the attacker's ability to spawn malicious TEE instances. The attack code consists of two parts. First, the compromised virtual machine monitor (VMM) is used to modify the scheduling of the TDs and to trigger the single-stepping mitigation through external interrupts. Second, an attacker-controlled TD is used to relay information about the current mitigation state to the VMM.

To mount the attack, first, the attacker has to recover the LFSR state on the logical CPU core used for the attack, as shown in the upper part of Figure 1b. As an LFSR outputs its internal state as a random number and the updates to the state consist of linear operations, the full state can be reversed by observing the output. The polynomial used for the LFSR is `0xB4BCD35C` and is publicly available [25]. The 5 least significant bits of the LFSR output plus 1 is the number of single-steps the mitigation does before returning to the host. The plus 1 is required to ensure that at least one instruction is executed. To recover the output of the LFSR, we exclusively schedule our attacker-controlled TD and continuously trigger the mitigation. The attacker TD runs the small code gadget depicted in Listing 1 to report the number of executed instructions to the VMM. In more detail, this step works as follows. The attacker TD signals the VMM through a VM call that it is ready for a measurement. To start the measurement, the VMM sets up the APIC timer to immediately trigger an external interrupt and resumes the attacker TD. As interrupts are masked until the attacker TD is entered, the pending interrupt causes



(a) Flow Chart



(b) Sequence Diagram

Figure 1: A flow chart (Figure 1a) for a rough overview of TDXploit and a sequence diagram Figure 1b providing more detailed information. First, we recover the internal RNG state by triggering the mitigation on the attacker TD and reversing the state. Next, we schedule the attacker TD and trigger the mitigation until the RNG state is not 1. Only when the next random number is 1 the victim TD is scheduled.

a VM exit before the attacker TD can execute an instruction. As no instruction is executed in the guest, the single-stepping mitigation is triggered by ICSSD. The instructions in the attacker TD following the VM call consist of multiple `add` instructions that increase a counter residing in an unencrypted memory location that is shared with the VMM. When control is returned to the VMM, they can infer the number of instructions executed by the mitigation by computing the delta of the counter in the shared memory region.

We use the least significant bit of the number of instructions executed by the mitigation for the LFSR state recovery. While it is possible to use all 5 recovered bits for the state recovery, this would only marginally improve the attack, as each new mitigation trigger only reveals one new additional bit. As we recover one bit per mitigation trigger, we schedule the attacker TD and trigger the mitigation 32 times to recover the full LFSR state. With the state recovered, we are able to predict the mitigation behavior on this core. This information can already be directly used for instruction counting attacks on victim TDs by triggering the mitigation every time the victim TD is scheduled. However, by abusing control over scheduling, this can also be used to carry out single-stepping attacks, as depicted in the lower part of Figure 1b. Again, the attacker TD and the victim TD are scheduled to run on the same logical core, exploiting the shared LFSR state. The VMM constantly triggers the mitigation and schedules the attacker TD until the next invocation of the mitigation would execute exactly one instruction. At this point, the VMM schedules the victim TD. Afterward, the VMM again schedules the attacker TD until the mitigation allows for the next single-stepping opportunity. The resulting attack, TDXploit, is highly reliable, as the number of instructions executed is architecturally ensured by the TDX module, removing the possibility for unintentional zero-steps or multi-steps entirely. Furthermore, we do not rely on any side channels, removing possible sources of noise and inaccuracies. To perform targeted multi-steps, e.g., to skip uninteresting victim code, the same method can be used by scheduling the victim when the next random number equals the desired number of steps. This provides a middle ground between the fast but coarse-grained page fault-controlled channel and fine-grained but slow single-stepping.

3.4 Evaluation

In this section, we evaluate the speed and accuracy of TDXploit and demonstrate its feasibility in an end-to-end attack against the ECDSA implementation of OpenSSL. As previously mentioned, the evaluation was performed on an Intel Xeon Silver 4514Y running Ubuntu 24.04 using the TDX-enabled software stack from Canonical [10]. In line with prior work [55, 56], we assume a VM with a single core to ease the implementation effort.

Accuracy In our first evaluation scenario, we use the established [51, 55, 56] synthetic example of a tight loop consisting only of `nop` instructions. We single-step the loop for a total of 85 000 000 instructions and observe only 2 misclassifications, achieving an accuracy of $>99.99\,\%$. A single-step takes 3.7 ms ($\sigma_{\bar{x}} = 0.06\,\text{ms}$, $n = 10\,000$) on our system allowing for 270 single-steps per second.

Attack on OpenSSL In this section, we use our TDXploit primitive to perform the end-to-end attack against the modular reduction implementation of ECDSA curves in OpenSSL that was presented in [55]. ECDSA is a signature scheme that requires a nonce for each signature, which must remain secret. In addition, the nonce must be smaller than the group order of the ECDSA curve. One approach for generating such a nonce is modular reduction, where an initial random value k' is reduced until it matches the imposed size limitations. The attack exploits that the modular reduction code of the ECDSA implementation in OpenSSL executes a different amount of instructions depending on the secret value k', which leaks information about the final nonce k used for the signature. For the brainpoolp224r1 curve, certain instruction counts leak the values of the 7 most significant bits of the nonce k, as detailed in Listing 2. For brainpoolp224r1, executions where the control flow passes Line 7 but not Line 11 correspond to signatures with a biased nonce. Such nonce biases can be used to recover the secret key [2]. The attack requires 33 biased signatures, which amounts to approximately 200 000 observed signature generations due to the low probability of the event. The vulnerability was at least present since OpenSSL version 3.2.0, and, to the best of our knowledge, there was no constant time alternative available. OpenSSL mitigated the attack by switching to rejection sampling starting with version 3.3.1. To orchestrate the attack, we use the attackers' ability to force and observe page faults to look for a unique page fault pattern that indicates that the vulnerable function is about to execute. This pattern has previously been determined in an offline step. Afterward, we single-step the victim TD until the end of the vulnerable code section is reached, which we again infer via page faults. For our PoC, we assume that the attacker has already located the OpenSSL library in the GPA space of the TD as, e.g., demonstrated in [29–31, 37]. With single-stepping we need 119.7 ms per signature without we need 0.062 58 ms per signature. Compared to the now mitigated single-stepping attack in [55], our attack is slower by a factor of 4. We did not further analyze the cause of the performance difference.

3.5 Mitigations

Unlike existing single-stepping attacks that are based on external interrupts forcing control back to the host, TDXploit exploits the single-stepping mitigation itself to achieve reliable single-stepping. There are multiple parts in the TDX single-stepping mitigation that make TDXploit possible and

```
int bn_div_fixed_top(BIGNUM* dv, rm, num,
-> divisor, BN_CTX *ctx) {
for (i = 0; i < loop; i++, wnumtop--) {
  for (;;) {
    if ((t2h < rem) ||
        ((t2h == rem) && (t2l <= n2)))
      break;
    q--;
    rem += d0;
    if (rem < d0) //don't let rem overflow
      break;
    if (t2l < d1)
      t2h--;
    t2l -= d1;
  }}};

```

Listing 2: Simplified version of bn_div_fixed_top from openssl/crypto/bn/bn_div.c. This function divides *num* by *divisor* and is called during the nonce generation. Based on figure from [55]

need to be changed to not only mitigate TDXploit but also make similar future attacks more challenging.

Improved RNG LFSRs are very useful for generating number sequences, but they are very bad random number generators (RNGs), especially for security-related purposes such as Intel's mitigation. The linearity of LFSRs makes their internal state trivial to reverse. Furthermore, they output part of their internal state directly as the random number. Replacing the LFSR with a proper pseudo RNG would prevent state recovery entirely. While LFSRs have the advantage of being extremely fast in generating numbers, performance is not an important factor for the RNG used in this mitigation. As the mitigation, if triggered, results in multiple extremely expensive VM entries and VM exits, the additional overhead of a proper pseudo RNG would not significantly affect the overall overhead of the mitigation. Furthermore, the mitigation, and therefore the random number generation, is rarely triggered under normal execution, which is the main reason a mitigation with such a large overhead is even viable to begin with.

Per vCPU RNG State TDXploit relies on an attacker-controlled TD to recover the RNG state and to skip undesirable step counts. This only works because of the per-core RNG state. Using a per TD or per TD vCPU RNG state would prevent our attack approach, even if a bad RNG like an LFSR is used. While the former requires slightly less memory, the latter would work without locking. Furthermore, the TDVPS mechanism already allows storing per TD vCPU state, making the addition of the RNG state an easy change. However, the StumbleStepping attack from Wilke et al. [55] might still be used to recover the LFSR state, although its noisy signal might make reversing the LFSR more challenging. In summary, using a per TD (vCPU) RNG state significantly reduces the attack surface but is not sufficient on its own to protect a weak RNG like an LFSR.

Improved RNG range A larger value range for the amount of instructions executed by the mitigation would make it harder for any attacker to exploit side effects of the mitigation. In its current state, the mitigation already excludes the number 0 from its random number generation, as it would result in no progress. We would further recommend removing at least 1 as well, as this would result in a single-step, which is what this mitigation is trying to actively defend against. Even with this change, the range of noise the mitigation introduces is very limited, with a maximum of 32 steps. We suspect that the main reason for this low number is the enormous overhead each additional step introduces with the currently used stepping mechanism. Thus, we propose changing the mechanism the TDX module uses to execute additional instructions inside the TD. The VMX-preemption timer could be a promising solution. This is a counter that decrements with every cycle executed in the guest and triggers a VM exit when it reaches 0. Thus, the number of VM entries and exits is constant instead of scaling with the number of executed steps. As a result, significantly more instructions could be executed in the guest without any additional performance overhead. This mechanism would also significantly reduce the attack surface for side channels that try to infer the amount of executed instructions. As the VMX-preemption timer relies on the time stamp counter (TSC) and not on instructions executed, the performance counter already used for the ICSSD detection method can be used to ensure that a certain range of instructions were executed eliminating the possibility of a vulnerability similar to TDXdown [55]. The preemption timer would, therefore, only speed up the instruction execution, while the performance counter ensures that enough instructions are executed.

An alternative to the TSC-based preemption timer would be performance monitoring interrupts (PMIs). PMIs can be configured to trigger whenever a performance counter overflows. With one performance counter already counting instructions retired for ICSSD, the same counter can be used to trigger a PMI after a random number of instructions executed when the mitigation is active. As PMIs can only be delivered when they are not masked, the CR8 register can not be used to mask all external interrupts. Alternatively, the "acknowledge interrupt on VM exit" feature can be enabled for the TD to avoid returning to the host, and encountered interrupts can be buffered by the TDX module. The buffered interrupts can then be triggered again through self-IPIs (inter-processor interrupts) directly before returning to the host when the mitigation is done. Similar to the VMX-preemption timer method, this would also not require any hardware changes while at the same time significantly increasing the possible number of instructions executed by the mitigation without the high overhead of constant VM entries and VM exits.

In summary, we recommend using a secure RNG that cannot easily be reversed to mitigate our attack and to switch to a dedicated RNG state per TD vCPU to hedge against potential

attacks. Finally, using a larger range for the amount of instructions that can be executed by the mitigation would further increase its security.

4 TDX Flush+Flush

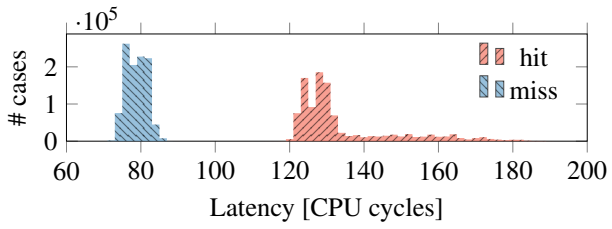
In this section, we show that Flush+Flush can be used on TDX private memory, contrary to prior findings. First, we demonstrate that the `clflush` instruction ignores the HKIDs used by the memory encryption system. In combination with the VMM's ability to create mappings to all memory locations, this enables Flush+Flush attacks on arbitrary TD memory without requiring shared memory. To demonstrate the effectiveness of this attack, we perform a last-round AES T-Table attack on OpenSSL from the host on a TDX guest, recovering the full encryption key.

4.1 Attack

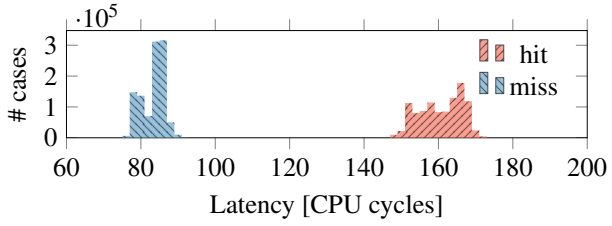
The Flush+Flush attack introduced by Gruss et al. [18] exploits the timing difference between a `clflush` on a cached memory location and an uncached memory location. If the memory location that is being flushed is in the cache, `clflush` has to evict the cache line, causing an increased latency. Using this time difference Flush+Flush can detect if a memory location has been recently accessed. Furthermore, Flush+Flush is fast and has no blindspot, making it a very powerful attack primitive [41]. The main disadvantage of traditional Flush+Flush is that it requires shared memory between the attacker and the victim.

On Intel TDX, guest private memory is encrypted using TME-MK, which allows the use of multiple encryption keys specified in the upper parts of the physical address to encode the HKID. With TDX, any read access by the host to a memory location encrypted with a private HKID returns all-zero data. Returning all-zero data instead of the ciphertext or the decryption of the ciphertext with the public HKID is crucial for preventing ciphertext side-channel attacks [31]. Nonetheless, accessing a memory location with a different HKID still loads the data into the cache. Since the HKID is encoded in the physical address bits and thus influences the cache tag, it, in principle, enables multiple decryptions of the same memory location to reside in the cache at the same time. However, an additional coherence mechanism flushes any existing cache entries that only differ in the HKID. Prior work [1, 55] exploited this mechanism to build a cache attack where the attacker continuously loads the targeted address with a different HKID to observe latency spikes due to the coherence protocol. Intel implied that they plan on removing this coherence mechanism in future CPUs [24].

While the TDX documentation implies that `clflush` on TDX private memory should not be possible [24, §8.5.1], we discovered that the `clflush` instruction ignores HKIDs and flushes all cache lines related to a physical address. Prior work



(a) Native



(b) TDX

Figure 2: Flush+Flush hit-miss histograms for a regular native attack (Figure 2a) and a TDX host attacker targeting a guest VM (Figure 2b). The miss (not cached) timings for both scenarios are in a similar range. On average, the hit (cached) timings for the TDX attack are 30 cycles higher, falling into the higher end of measured hits in the native scenario, increasing the hit-miss margin significantly. The increased hit timings could be the result of overhead introduced with the memory encryption used for TDX guest memory.

that primarily analyzed 4th generation Xeon Scalable CPUs reported the expected `clflush` behavior, leading us to assume that only 5th geneneration Xeon Scalable CPUs upwards are affected by our attack. As TDX-enabled 4th generation Xeon Scalable CPUs are not publicly available, we are unable to verify this assumption.

The results of our measurements are shown in Figure 2. Figure 2a contains the hit-miss histogram for Flush+Flush in a native scenario without TDX. Cache misses (memory not cached) require ∼80 cycles, and most cache hits require ∼130 cycles with outliers in the range of 135 to 180 cycles. Figure 2b contains the hit-miss histogram for Flush+Flush on TDX private memory with a host attacker. The host executes `clflush` on the target physical memory using the zero HKID, while the victim TD accesses the memory through its private mapping. Similar to the native scenario, the misses are at ∼80 cycles, which is to be expected as nothing has to be done. The hit case is at ∼160 cycles and thus significantly higher than the ∼130 cycles required for the majority of cases in the native scenario. As we also observed multiple instances in the native scenario where `clflush` required ∼160 cycles, this might be a less optimal case for `clflush` that is reliably triggered by flushing TD private memory.

Not only is the timing increased for `clflush` if the private memory is cached, but the cache line is also evicted, as shown

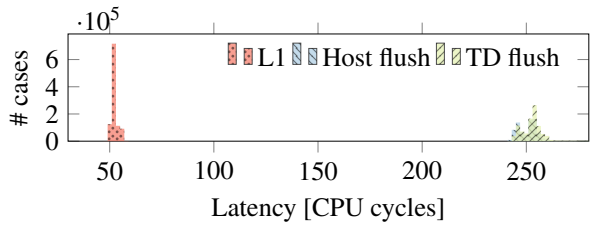


Figure 3: TDX private memory access timings when the host flushes the memory using a different HKID, when the guest flushes the memory before the access, and when only L1 accesses are performed. The access timings after a host flush and after a TD flush are almost identical.

in Figure 3. The L1 hit timings in the TD are at 52.4 cycles, with the miss timings if the guest flushes its own memory at 252.5 cycles. After the host evicts the guest memory with a different HKID, the average access time for the guest is 250.6 cycles, which is almost identical to the timings of a guest flush. Therefore, it is likely that guest memory was evicted by the host flush.

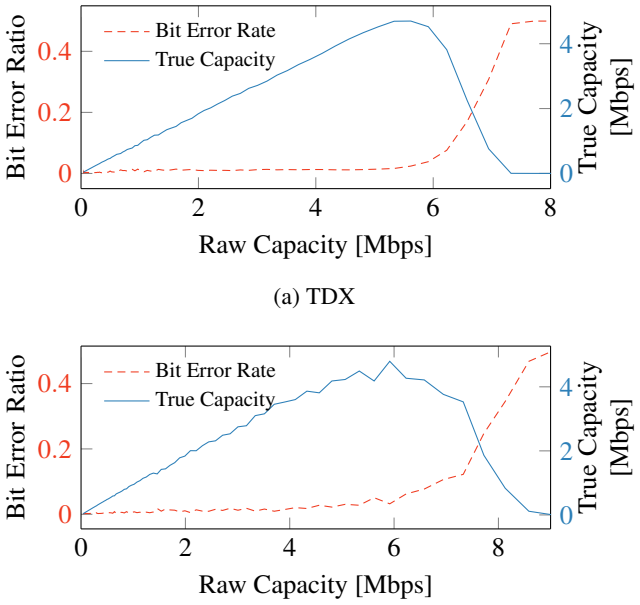
4.2 Evaluation

In this section, we evaluate the effectiveness of Flush+Flush on TDs and compare it to a native attack. To determine the potential leakage rate of Flush+Flush on TDs, we first build a covert channel and compare it to the native case. For further comparison, we perform an AES T-Table first-round attack on a TD and compare it to the native attack using the OpenSSL AES implementation. Finally, we perform a last-round attack on a TD and recover the full AES key. We assume the threat model described in Section 3.2.

4.2.1 Covert Channel

Our covert channel is based on our observations discussed in Section 4.1. The TD sends data through memory accesses and the host receives data through flushing the same memory location. We use a time-sliced approach for data transmission, with one bit transmitted in each time slice. For a clock, we use the TSC, which can be accessed through `rdtsc`. While the absolute TSC value of the host and the guest may differ by a fixed offset, they still increment at the same rate [25]. We assume that the sender and receiver are perfectly synchronized at the start of the transmission as this is a separate challenge that has already been discussed and solved in previous work [34].

To find the optimal time-slice length for our native and TDX Flush+Flush attack, we perform our attack with decreasing time-slice length. With a decreasing time-slice length, the raw capacity of the transmission increases due to more bits sent in a given time frame. At the same time, the error ratio increases as time slices become too short to correctly transmit bits. We use the binary symmetric channel model to compute the true capacity based on the error ratio and the raw capacity and determine the optimal time-slice length based on this.



(a) TDX

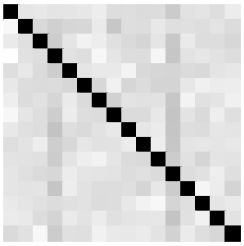
(b) Native

Figure 4: Covert-channel true capacity and error ratio with different raw capacities for a Flush+Flush covert channel, executed natively (Figure 4b) and with Intel TDX with the TD sending and the host receiving (Figure 4a). The channel on TDX reaches its maximum true capacity of 4.6 Mbit/s at a raw capacity of 5.6 Mbit/s and the native channel reaches its maximum of 4.8 Mbit/s at a raw capacity of 5.9 Mbit/s.

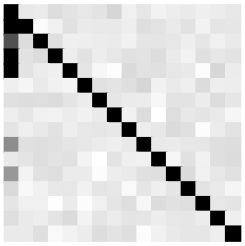
The results of these measurements are provided in Figure 4. Our native Flush+Flush attack has a true capacity of 4.8 Mbit/s ($\sigma_{\bar{x}} = 0.01$ Mbit/s, $n = 20$) with an error ratio of 3.2 % ($\sigma_{\bar{x}} = 0.7\,\%$, $n = 20$) at a raw capacity of 5.9 Mbit/s. Our Flush+Flush attack with a host receiver and a TDX guest sender has a true capacity of 4.6 Mbit/s ($\sigma_{\bar{x}} = 0.01$ Mbit/s, $n = 20$) with an error ratio of 2.6 % ($\sigma_{\bar{x}} = 0.06\,\%$, $n = 20$) at a raw capacity of 5.6 Mbit/s. The Flush+Flush on TDX is only marginally slower than the native version, most likely due to the slightly higher average hit-timings, as discussed in Section 4.1 and slightly worse synchronization, due to the lack of a shared TSC in the TDX-based attack.

4.2.2 AES T-Table Attack

To demonstrate the effectiveness of Flush+Flush on TDX private memory, we perform an AES T-Table attack on OpenSSL. Contrary to a native Flush+Flush, a malicious VMM can not rely on shared memory for an attack, as shared libraries used inside a TD would normally be loaded into private memory, inaccessible by the host. First, the attacker has to find the page



(a) TDX



(b) Native

Figure 5: Comparison of a first-round AES T-Table key recovery between native and TDX Flush+Flush over 10 000 encryptions with a zero key. A highly visible diagonal indicates a low amount of noise. The attack on the TD (Figure 5a) performs similarly well as the native attack (Figure 5b).

containing the T-Tables of the guest's AES implementation. As the guest's private memory is not readable to the host, we use Flush+Flush to search through the guest's physical address space for the T-Tables. During a typical AES encryption, almost all cache lines in a T-Table are accessed. To find the correct memory locations, we use the page offset for the T-Tables in the AES library and search for a large amount of cached memory that has the size of the T-Tables at these offsets after the AES encryption has been executed by the guest. The execution of the AES code can be detected through different ways, e.g., network transmissions or page fault tracking. To filter out false positives due to other applications, we repeat our measurements several times. After only 10 encryptions, we can determine the correct memory location with 99.2 % accuracy on our system. This search method is similar to the one proposed by Gruss et al. [19] for cache template attacks.

To visualize the accuracy of the tested attacks, we perform a first-round attack with known plaintext on one T-Table with a zero key. The access number heatmaps for the T-Table after 10 000 encryptions for both native Flush+Flush and the TDX scenario are shown in Figure 5. The visibility of the diagonal indicates the amount of noise each attack has. A strongly visible diagonal means more correctly detected accesses and, therefore, less noise. As expected, both the native Flush+Flush and the TDX scenario perform almost identically, indicating no significant additional noise in the TDX-based scenario.

In addition to the first-round attack, we perform a last-round attack on a TD, assuming known ciphertext. We are able to recover the full AES key after 8 986 ($\sigma_{\bar{x}} = 119$, $n = 100$) encryptions when attacking a TD. This number is similar to the results of a native attack reported in previous work [41].

4.2.3 OTP Recovery

To showcase the combined capabilities of TDXploit and Flush+Flush, we use the two primitives concurrently to attack a vulnerable TOTP library [46]. Gast et al. [15] demonstrated

```
    for (int k = 0; k < 32; k++) {
      if (c == OTP_DEFAULT_BASE32_CHARS[k]) {
        // c is the current character
        block_values[j] = k;
        found = 1;
        break;
      }
    }

```

Listing 3: TOTP base32 decoding loop, executed for each character in the input [46].

that on AMD SEV, it is possible to leak the TOTP secret of the same TOTP library by tracking loop iterations in the base32 decoder function. They tracked the guest by monitoring performance counters while single-stepping. Although TDX does not expose guest performance counter data to the host, we can mount a similar attack using Flush+Flush.

Whenever the TOTP library verifies or generates a TOTP token, the secret is base32-decoded. To decode the secret, the algorithm iterates over each character and compares it with the entries of an internal lookup table called `OTP_DEFAULT_BASE32_CHARS` as shown in Listing 3. Whenever the character matches the table entry, the comparison loop breaks, and the index of the matching entry is saved. Afterwards, the algorithm continues with the next character.

We can attack this decoding algorithm by single-stepping and using Flush+Flush to monitor the physical memory location of `OTP_DEFAULT_BASE32_CHARS`. The size of the lookup table is 32 B, which is small enough to fit into a single cache line. Thus, we only need to monitor a single address to detect all accesses. For our evaluation, we assume the memory location to be known to the attacker as it can be found through profiling the guest's physical memory similar to our attack described in Section 4.2.2. While the library is comparing a character with entries in the lookup table, we detect accesses every 5 instructions. The processing of correct characters causes another branch to execute, leading to a delay of 6 instructions between accesses. By distinguishing the two access patterns, we can follow the execution of the decoding algorithm. In addition, the data is decoded in chunks of 8 characters, after which we measure an overhead of 58 instructions before the algorithm continues processing secret characters, which we need to consider when extracting the secret key.

When executing the attack, we experience additional cache hits for `OTP_DEFAULT_BASE32_CHARS`. We assume that this is caused by branch prediction, prefetching of the lookup table, and out-of-order execution. We experimentally verified this assumption by adding the `serialize` instruction before the comparison, which reduced the additional cache hits significantly. However, the detected extra accesses mostly do not interfere with the decoding process. Even without the added `serialize` instruction, we are able to correctly extract the TOTP secret key **from a single trace** in 79.2 % of cases. We

Table 1: All analyzed attacks and their applicability to Host-to-Guest, Guest-to-Host, and Guest-to-Guest attack scenarios.

Attack	Host-to-Guest	Guest-to-Host	Guest-to-Guest
Analyzed in prior Work			
HKID Coherence [1]	✓	✗	✗
Analyzed in our Work			
Flush+Reload	✗	✗	✗
Flush+Flush	✓	✗	✗
PortSmash	✓	✓	✓
Prime+Probe (L1)	✓	✓	✓
Evict+Reload	✗	✗	✗

✗ denotes that this attack is not possible. ✓ denotes that this attack works. We are the first to demonstrate all of these attacks on TDX, except for HKID coherence, which was demonstrated in prior work [1].

achieve a 99.0 % success rate in key recovery by evaluating three traces and deriving the key using a majority vote. The single-stepping process takes approximately 9.1 s to generate a single trace. Thus, the total attack time to recover the TOTP secret is below 30 s.

This attack is only possible due to the combination of a reliable single-stepping technique and Flush+Flush. When only using single-stepping, it is only possible to determine the total number of instructions executed for the translation of all characters, as there is no way for the host to determine which character is currently translated. When only using Flush+Flush, the temporal resolution is not enough to differentiate between memory access after 5 or 6 instructions, making it not possible to determine the exact characters of the secret or which character is currently being translated. Consequently, this is an example of how reliable single-stepping in combination with existing primitives can be used to perform attacks that would otherwise not be impossible.

5 Systematic Evaluation

While our single-stepping attack discussed in Section 3 can by itself used to mount attacks, the main benefit of single-stepping to an attacker lies in strengthening other attacks, in particular, side-channel attacks. In this section, we look at different side-channel attacks and their applicability and efficacy in the context of Intel TDX and TDXploit. We discuss the viability of the attacks in different scenarios with Intel TDX, including the traditional host-to-guest (host attacking a guest) scenario, as well as the malicious guest scenarios guest-to-host (guest attacking the host) and guest-to-guest (guest attacking another guest). We verify the viability with a covert channel for each attack in each possible scenario.

All experiments and assumptions are based on an Intel Xeon Silver 4514Y running a stock Ubuntu 24.04 with the TDX changes provided by Canonical [10]. The CPU runs the most recent version of the TDX module at the time of writing (TDX module 1.5.06 [25]). The victim guest runs an Ubuntu 24.04 created through Canonical's TDX support scripts [10].

For each covert channel, we provide the true capacity based on the error rate and raw capacity, similar to previous work [34]. The covert channels are based on a time-sliced approach using the `rdtsc` instruction as the clock. While host and guest do not share the same TSC, the TSC in TDX guests increments at the same rate as the TSC of the host [25], which suffices for synchronization. As we only build the covert channels to show that a given attack works, we do not further optimize each channel beyond the point of showing that it can correctly transmit data, *i.e.*, the goal is not to outperform channels from prior work but only demonstrate the practicality. The results of our analysis are summarized in Table 1.

Flush+Reload. A typical Flush+Reload [58] attack requires shared memory between the attacker and the victim. In the guest-to-guest scenario, there is no shared memory between the victim and the attacker. With regular VMs, guest-to-guest attacks with Flush+Reload can be possible through page deduplication by the host. As page deduplication between different TDs is not possible, this eliminates the possibility for Flush+Reload between guests. Similar to the guest-to-guest scenario, in the guest-to-host scenario, there is also no shared memory between host and guest that holds relevant information. While the host can provide shared memory pages with TDX, these pages are intended to transfer data between the guest and the host. In the host-to-guest scenario, the host has access to all physical pages used by the guest. Despite this, the host cannot use the private HKID used for the guest memory for an attack. As accessing memory with a different HKID results in a separate cache line being loaded, it is technically not possible to perform Flush+Reload on a specific cache line. Instead, as we discuss later in this section, such accesses trigger a HKID coherence side channel. Hence, **Flush+Reload is not applicable in any of the three scenarios**.

Evict+Reload. Evict+Reload [19] is based on the same principle as Flush+Reload but replaces the flush-step with an eviction using an eviction set. Therefore, Evict+Reload also requires shared memory between attacker and victim, which is not available as discussed for Flush+Reload. Hence, **Evict+Reload is not applicable in any of the three scenarios**.

Flush+Flush. Similar to Flush+Reload, a traditional Flush+Flush [18] attack is not possible in the guest-to-host and guest-to-guest scenarios due to a lack of shared memory. Contrary to Flush+Reload, Flush+Flush is possible in the host-to-guest scenario, despite the lack of a shared cache line. As outlined in Section 4, the `clflush` instruction ignores the HKID when performing the flush, allowing the host to flush cache lines belonging to private guest memory. This enables Flush+Flush attacks on arbitrary guest physical memory by the host, without requiring any shared memory. We confirmed that **host-to-guest Flush+Flush is possible** and measured its capacity in

a covert channel. We achieved a true capacity of 4.6 Mbit/s, as shown in Section 4.2.1. Note that our Flush+Flush attack does not trigger the HKID coherence side channel and, hence, is not mitigated by any measures taken against the HKID coherence side channel. As we showed in Section 4.2.3, there is also no interference between Flush+Flush and TDXploit.

PortSmash. The PortSmash [3] attack relies on contention in the execution ports of a physical CPU core. Consequently, PortSmash requires the attacker and the victim to run on the same physical core at the same time, albeit on separate logical cores. The attacker detects throughput changes on the target execution port through victim code execution. In principle, a TD can mitigate this attack by enforcing that SMT has to be disabled for it to run. Similarly, the host can mitigate guest-to-host attacks by either disabling SMT or not scheduling sensitive code on the same physical core as TDs while they are active. Still, we practically confirmed that **host-to-guest, guest-to-host, and guest-to-guest PortSmash attacks are possible** and built a covert channel for each scenario. For all three attack scenarios, we use the `divsd` instruction, which is handled by execution port 1 on our CPU. The true capacities for our channels are 356.0 kbit/s (guest-to-guest), 395.9 kbit/s (host-to-guest), and 396.0 kbit/s (guest-to-host). We observed no interference between TDXploit and PortSmash when using TDXploit to target a specific part of the code.

Prime+Probe (L1). An attacker can use Prime+Probe [40] to target different caches. Initial works target the L1 cache [40], whereas later works also target inclusive last-level caches [34]. However, recent Intel server processors abandoned this design in favor of non-inclusive last-level caches, which cannot be attacked by the known Prime+Probe attacks, *i.e.*, only Prime+Probe on the L1 cache has been demonstrated on these CPUs so far [41]. Similarly, as we target an Intel Xeon Silver 4514Y, which has a non-inclusive last-level cache, like all CPUs supporting TDX, we can only target the L1 cache. We fill the L1 cache sets with attacker cache lines and detect any evictions caused by the victim. We find that **host-to-guest, guest-to-host, and guest-to-guest Prime+Probe attacks against the L1 cache are possible**, as the L1 remains a hardware component shared across security contexts and Prime+Probe does not require any shared memory. To measure the capacity of the Prime+Probe attacks, we use covert channels, achieving transmission rates of 620.1 kbit/s (guest-to-guest), 561.0 kbit/s (host-to-guest), and 526.8 kbit/s (guest-to-host). We observed no interference between TDXploit and Prime+Probe when using TDXploit to target a specific part of the code.

HKID Coherence side channel. In their security report on TDX, Google mentions a possible attack based on a coherence mechanism related to HKIDs [1]. When a memory location is loaded with one HKID, all cache lines with the same physical address and a different HKID are evicted from the cache.[1] The HKID coherence side channel can be used by the host to detect guest memory accesses, which lead to coherence-induced changes in the cache state. The HKID coherence side channel requires access to the victim's memory with different HKIDs, which is not possible from inside a TD. Hence, we find that only the **host-to-guest HKID coherence side channel is possible**. We measured the capacity of the HKID coherence side channel in a host-to-guest scenario covert channel and achieve a capacity of 3.0 Mbit/s.

6 Discussion & Related Work

In this section, we discuss the impact of our findings. TDXploit, unlike previous single-stepping approaches, exploits the possibility of spawning attacker-controlled TDs. With this we can leak information from the TDX module without noise.

The possibility of malicious code inside a TEE has previously been explored by Schwarz et al. [45]. They assume that a malicious or buggy piece of code is signed by Intel and, therefore, allowed to run inside Intel SGX and investigate the possibility of attacking the host through Prime+Probe from inside the TEE. Consequently, the attack is significantly more challenging for the host to detect, as it is protected by SGX. More recent Intel CPUs no longer require enclaves to be signed by Intel, making such attacks even more realistic [21]. Van Bulck et al. [50] analyzed 8 major open-source shielding frameworks for SGX enclaves and found 35 vulnerabilities. Schwarz et al. [44] show that the unrestricted access of SGX enclaves to the memory of their host application can be abused to manipulate the host to execute arbitrary code. This makes malware almost invisible to existing detection methods, as the actual attacker code is hidden inside the enclave. Jang et al. [26] use Rowhammer to flip SGX memory from inside the enclave, leading to a processor lockdown. Gruss et al. [17] flip bits in host memory from within an SGX enclave using Rowhammer. Contrary to existing Rowhammer attacks, their attack is almost invisible to the host.

The closest work to TDXploit is TDXdown [55]. TDXdown uses a different flaw in the TDX single-stepping mitigation to bypass it. In the initial version of the single-stepping mitigation, an attack was detected only through heuristics, most notably through the TSC. The TSC-based heuristic can

[1] While the authors mention that their attack could be seen as a Flush+Reload or Flush+Flush attack, this naming is not consistent with the published literature: Loading a cache line triggers coherence and, thereby, evicts one or more other cache lines. This is not the same as cache eviction through cache or cache-set contention, *i.e.*, it is different from the eviction in an Evict+Reload attack. It is also not a flush operation which specifically removes a single specified cache line from the cache, *i.e.*, it is also different from the flush in Flush+Reload or Flush+Flush attacks. Hence, we believe the attacks by Aktas et al. [1] should rather be referred to as HKID coherence side channel to avoid confusion with other distinct and already known attack techniques like Evict+Reload, Flush+Reload, or Flush+Flush.

be bypassed by fixing the CPU to its minimal frequency, tricking the mitigation into thinking a large number of instructions were executed in the guest. They then applied the APIC timer-based single-stepping technique already used in previous work, such as SGX-Step [51] and SEV-Step [56]. This bug is mitigated in the current TDX module version through the introduction of ICSSD [24]. Our TDXploit attack does not bypass the TDX single-stepping mitigation but exploits a flaw to abuse the mitigation and guarantee reliable single-stepping. Furthermore, unlike the single-stepping introduced in TDXdown, TDXploit works on the current version of the TDX module with ICSSD enabled.

To mitigate the Flush+Flush attack on TD private memory, software changes are not enough. As long as the physical memory used for TDs can be mapped by the host, Flush+Flush can be executed on them. While it is possible to avoid Flush+Flush for specific cases by only executing code not vulnerable to Flush+Flush inside TDs, this does not properly mitigate the vulnerability. The ability of TDX to run regular applications inside TDs leads to users unaware of this issue executing vulnerable code regardless of the existence of Flush+Flush resilient code. Furthermore, as it is possible to run general-purpose operating systems inside TDs, this would require all software that could potentially leak sensitive information to be resilient against Flush+Flush, which is impractical. Alternatively, this issue can be mitigated through a simple hardware change that makes `clflush` aware of HKIDs, as it seems to be the case on 4th generation Xeon Scalable CPUs [1,55]. We are unaware of a valid use case for `clflush` ignoring HKIDs. In any case, a dedicated, page granular flushing mechanism, as e.g. implemented by AMD SEV [5], should always be sufficient. Pages that are returned to the host are already written back to main memory by the TDX module. Pages given to the TDX module for private memory can be written back to main memory by the host before they are provided to the module.

The closest work to our Flush+Flush attack on TDX guests is the HKID coherence-based attack briefly mentioned in the Google security report on TDX [1]. The coherence protocol allows a memory location to be in the cache with only one HKID at a time. This can be abused to detect guest memory accesses, as the attacker can load the memory with a public HKID and detect memory accesses that use a private HKID on the same memory location through the cache line evictions enforced by the coherence mechanism. Contrary to our Flush+Flush attack, which exploits that `clflush` ignores HKIDs, the coherence-based attack exploits the current implementation of the coherence protocol regarding HKIDs, making them inherently different. Additionally, the Flush+Flush-based approach does not trigger cache misses on the host side, allowing for significantly faster resets of the cache state in case of a victim access. Finally, Intel implies in the TDX base specification that they, with the introduction of the introduction of the `TDX_FEATURES.CLFLUSH_BEFORE_ALLOC` feature flag, plan to mitigate the HKID coherence-based channel [24].

Similar to our work, Wang et al. [53] analyze multiple side channels, including Prime+Probe, and their effectiveness on Intel SGX. Additionally, they analyzed how the tested attacks, in combination with the Intel SGX threat model, can be used to build significantly stronger attacks. Rauscher et al. [41] analyzed and compared a wide range of cache side-channel attacks using 9 metrics on Intel Sapphire Rapids and Emerald Rapids CPUs. Contrary to our work, which analyzes the viability of various side-channel attacks on Intel TDX, their work focuses only on cache side channels in a native scenario. Nilsson et al. [39] created a survey of published attacks on Intel SGX, which includes if they are SGX specific, the attack target of each attack, and possible mitigations for each attack.

7 Conclusion

We introduced a novel technique for single-stepping attacks on Intel TDX, named TDXploit. TDXploit exploits that an attacker can control and predict Intel's single-stepping mitigation. TDXploit achieves a higher ($>$99.99 %) single-stepping accuracy than if there were no mitigation in the first place. Furthermore, TDXploit is the first technique for reliable multi-stepping. While TDXploit does not rely on any side channels, we show that it can be combined with various side channels to mount powerful attacks: We discover a previously unknown microarchitectural behavior with Flush+Flush on TDX guest physical memory, allowing Flush+Flush attacks on TDX guests without shared memory. We demonstrate the impact of this finding by performing a full key recovery on the OpenSSL AES T-Table implementation using Flush+Flush, requiring only 8 986 encryption traces. We also systematically evaluated 6 different state-of-the-art side-channel attacks in the context of Intel TDX and TDXploit and found that only PortSmash and Prime+Probe (on the L1 cache) work in the more dangerous malicious guest scenario. However, Flush+Flush, PortSmash, Prime+Probe, and the HKID coherence side channel all work to attack TDX guests from a malicious host. Finally, we demonstrated the real-world impact of TDXploit by re-creating a previously mitigated attack on the ECSDA implementation of OpenSSL as well as an end-to-end attack on TOTP secret keys, previously only demonstrated with a different side channel on AMD SEV. We conclude that further mitigations are necessary, in particular mitigating single-stepping, which often acts as an amplifier for side-channel attacks.

Acknowledgements

This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85), and the BMBF (projects SASVI and AnoMed). Additional funding was provided by generous gifts from Red Hat, and Intel.

Ethics Considerations

We responsibly disclosed our findings to Intel on November 27, 2024. Intel confirmed the issue on January 22, 2025. They plan to publish the vulnerability in August 2025. All our experiments were done on our own machines with no code from other users running on them.

Open Science

We plan to publish all our code used in this paper on Zenodo (https://doi.org/10.5281/zenodo.15536636) and GitHub (https://github.com/isec-tugraz/TDXploit).

References

[1] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel trust domain extensions (TDX) security review. Technical report, Google, 2023.

[2] Martin R. Albrecht and Nadia Heninger. On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem, 2021.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *S&P*, 2019.

[4] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[5] AMD. AMD64 Architecture Programmer's Manual, 2023.

[6] AMD. AMD Secure Encrypted Virtualization (SEV), 2024. URL: https://developer.amd.com/sev/.

[7] ARM. Arm Confidential Compute Architecture, 2024. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[8] ARM. TrustZone for Arm Cortex-M Processors, 2024. URL: https://www.arm.com/technologies/trustzone-for-cortex-a.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[10] Canonical. Intel® Trust Domain Extensions (TDX) on Ubuntu, 2025. URL: https://github.com/canonical/tdx.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[12] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. {AEX-Notify}: Thwarting precise {Single-Stepping} attacks through interrupt awareness for intel {SGX} enclaves. In *USENIX Security*, 2023.

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086*, 2016.

[14] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[15] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *NDSS*, 2025.

[16] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EuroSec*, 2017.

[17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[18] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*, 2015.

[20] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In *CHES*, 2020.

[21] Intel. An update on 3rd Party Attestation, 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/an-update-on-3rd-party-attestation.html.

[22] Intel. Intel Trust Domain Extensions, 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf.

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

[24] Intel. Intel Trust Domain Extensions Module Base Architecture Specification, 2024. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html.

[25] Intel. TDX Module 1.5.06 Source Code, 2024. URL: https://github.com/intel/tdx-module.

[26] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[27] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[28] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[29] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In *S&P*, 2022.

[30] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected {I/O} operations in {AMD's} secure encrypted virtualization. In *USENIX Security*, 2019.

[31] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security*, 2021.

[32] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*, 2021.

[33] Xiaoxuan Lou, Kangjie Chen, Guowen Xu, Han Qiu, Guo Shangwei, and Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In *DSN*, 2024.

[34] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

[35] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[36] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction. In *USENIX Security*, 2020.

[37] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In *EuroSec*, 2018.

[38] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[39] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. *A Survey of Published Attacks on Intel SGX*. 2020.

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[41] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In *NDSS*, 2025.

[42] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[43] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[44] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In *DIMVA*, 2019.

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

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

[47] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*, 2017.

[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[49] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[50] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS*, 2019.

[51] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*, 2017.

[52] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[53] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[54] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *AsiaCCS*, 2018.

[55] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *CCS*, 2024.

[56] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV. In *CHES*, 2024.

[57] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.

[58] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[59] Ruiyi Zhang, CISPA Helmholtz Center, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*, 2024.