TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX

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

Florian Sieck*
florian.sieck@uni-luebeck.de
University of Lübeck
Lübeck, Germany

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

ABSTRACT

Trusted Execution Environments are a promising solution for solving the data privacy and trust issues introduced by cloud computing. As a result, all major CPU vendors integrated Trusted Execution Environments (TEEs) into their CPUs. The biggest threat to TEE security are side-channel attacks, of which single-stepping attacks turned out to be the most powerful ones. Enabled by the TEE attacker model, single-stepping attacks allow the attacker to execute the TEE one instruction at a time, enabling numerous controlled- and side-channel based security issues. Intel recently launched Intel TDX, its second generation TEE, which protects whole virtual machines (VMs). To minimize the attack surface to side-channels, TDX comes with a dedicated single-stepping attack countermeasure.

In this paper, we systematically analyze the single-stepping countermeasure of Intel TDX and show, for the first time, that both, the built-in detection heuristic as well as the prevention mechanism, can be circumvented. We reliably single-step TDX-protected VMs by deluding the TDX security monitor about the elapsed processing time used as part of the detection heuristic. Moreover, our study reveals a design flaw in the single-stepping countermeasure that turns the prevention mechanism against itself: An inherent side-channel within the prevention mechanism leaks the number of instructions executed by the TDX-protected VM, enabling a novel attack we refer to as *StumbleStepping*. Both attacks, single-stepping and *StumbleStepping*, work on the most recent Intel TDX enabled Xeon Scalable CPUs.

Using *StumbleStepping*, we demonstrate a novel end-to-end attack against wolfSSL's ECDSA implementation, exploiting a control flow side-channel in its truncation-based nonce generation algorithm. We provide a systematic study of nonce-truncation implementations, revealing similar leakages in OpenSSL, which we exploit with our single-stepping primitive. Finally, we propose design changes to TDX to mitigate our attacks.

CCS CONCEPTS

• **Security and privacy** → *Domain-specific security and privacy architectures*; **Trusted computing**; **Cryptanalysis and other attacks**.

*These authors contributed equally to this work

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

KEYWORDS

Trusted execution; virtualization; TDX; side-channel; microarchitectural attacks; constant-time; ECDSA key reconstruction

ACM Reference Format:
Luca Wilke, Florian Sieck, and Thomas Eisenbarth. 2024. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690230

1 INTRODUCTION

Data privacy concerns and legal regulations still hinder processing of sensitive data in the cloud. Such outsourced computation requires implicit trust in the cloud service provider, that has full control over the machines that make up the cloud, and thus over the processed data. Trusted Execution Environments (TEE) are thriving due to the promise of protecting computations and data even from privileged adversaries with full control over the systems, e.g. over the hypervisor software. Effectively, TEEs lock out the cloud service provider and enable verifiably protected data processing on remote machines. Early designs like Intel SGX focused on protecting single processes and required the developer to adjust their application to the TEE. While the introduction of the library OS approach [6, 7, 49] partially solved this problem, a newer and more scalable approach is taken by the newest generation of TEEs, namely Intel Trust Domain Extensions (TDX) [21], AMD SEV [3, 25, 26], ARM Confidential Compute Architecture [5], as well as IBM Secure Execution [16]. This newest TEE generation protects entire virtual machines (VMs) and can thus be used as a drop-in solution to protect existing applications in the cloud with only minimal adjustments.

While removing the hypervisor from the trust base promises a wide range of use cases, it also comes with challenges and has severe implications for security. In fact, it introduces an attacker model in which the attacker has full system control. Thus, the adversary has a broad arsenal of mechanisms to gleam information from the protected code running inside the TEE. Since their introduction, TEEs have been extensively scrutinized by security researchers, revealing that microarchitectural side-channels remain an Achilles heel of current TEE designs: Soon after the release of SGX it was shown that control over the page table allows the hypervisor or OS to learn detailed information about the control flow of TEE-protected code [58]. Subsequent work demonstrated that numerous microarchitectural features such as caches [14, 32, 50] or branch prediction [13, 27] provide an even more fine-grained resolution of secret-dependent control flows or data accesses.

Interrupt driven single-stepping [10, 56], a powerful attack technique against Intel SGX and AMD SEV, greatly increases the temporal and spatial resolution of side-channel attacks on SGX [11, 31, 41, 45–48] and on AMD SEV [43, 44, 51, 56, 59]. As such it poses a particularly severe threat for the security of TEEs. One especially powerful attack that builds on single-stepping is *instruction counting*. There, the attacker combines page fault information with single-stepping to reveal the target's control flow with intra-page precision, allowing them to exploit even the smallest secret-dependent control flow deviations [10, 32, 33].

Thus, a good defense mechanism against single-stepping attacks is an important building block for securing TEEs. Given the long line of attacks on SGX, Intel recently published AEX-Notify [12] in collaboration with academic researchers. AEX-Notify introduces a hardware-software co-design that makes the enclave interrupt-aware and allows it to prevent single-stepping attacks by providing a special interrupt handler.

To ensure resistance against similar attacks on Intel TDX, Intel early on conducted several security reviews for TDX [1, 20]. As part of this effort, TDX provides a built-in countermeasure against single-stepping attacks. In contrast to the AEX-Notify approach, the TDX single-stepping countermeasure does not depend on the software inside the TEE. Instead, the countermeasure is implemented inside the TDX module, TDX's security monitor. The countermeasure consists of a detection heuristic and a special single-stepping prevention mode that gets activated by the heuristic. We present the first systematic investigation of Intel's single-stepping countermeasure and show two attacks that overcome different aspects of the countermeasure.

The first attack on the Intel TDX single-stepping countermeasure exploits a weakness in its detection heuristic to prevent the activation of the single-stepping prevention mode. The heuristic is partially based on the elapsed time between entering and exiting the protected VM, which is very small if the VM is single-stepped. We manipulate the TDX module's sense of time, causing it to observe normal execution times, although the VM is single-stepped. While this vulnerability should be mitigatable by updating the detection logic, defining a safe and sound rule set is not trivial.

The second attack, *StumbleStepping*, exploits the inherent side-channel attack surface of the TDX single-stepping prevention. The intended effect of the single-stepping prevention mode is to stop the hypervisor from obtaining fine-grained insights into the protected VM's progress. *StumbleStepping*, however, exploits the prevention mode's inherent cache side-channel to leak the number of instructions executed by the protected VM. As such, *StumbleStepping* exploits a systematic issue that is not easily fixable, meanwhile providing a somewhat weaker leakage than single-stepping.

To demonstrate the capabilities of *StumbleStepping* in exploiting small leakages, we target a minuscule leakage found in the current ECDSA implementations of wolfSSL. The leakage was already identified in [52], but was deemed unexploitable by the authors. We show that this leakage can actually be captured by *StumbleStepping* and is exploitable for select choices of elliptic curves. We provide an extensive analysis of the ECDSA leakage, revealing similar problems in OpenSSL, which we exploit with our single-stepping primitive.

In summary, we:

- Demonstrate an attack that renders the TDX single-stepping countermeasure inoperative and enables single-stepping on TDX
- Introduce the *StumbleStepping* attack, an inherent cache side-channel in the single-stepping countermeasure of TDX that leaks the number of instructions executed by the TD
- Use *StumbleStepping* and our single-stepping primitive, to leak ECDSA keys in a novel nonce truncation-based attack against wolfSSL and OpenSSL
- Provide an extensive analysis of nonce truncation leakages in ECDSA implementations including wolfSSL and OpenSSL

The code to reproduce our results is available at *https://github.com/UzL-ITS/tdxdown*.

The remainder of this paper is structured as follows: Section 2 introduces required background information. Next, Section 3 analyzes the TDX single-stepping countermeasure in detail. Section 4 and Section 5 introduce and evaluate the two main attack primitives of this paper. Section 6 analyzes nonce truncation-based control flow leakages in ECDSA implementations. Afterwards, in Section 7 we present two attack case studies, exploiting wolfSSL's ECDSA leakage using *StumbleStepping* and OpenSSL's ECDSA leakage via our single-stepping primitive. Finally, we discuss limitations and countermeasures in Section 8.

Responsible Disclosure. We officially reported our findings to Intel's PSIRT team on October 11, 2023. Using our proof of concept code they reproduced our attacks and issued CVE 2024-27457. Intel is working on a countermeasure against the single-stepping attack and states that future TDX module versions after 1.5.0.6 should no longer be affected. Intel will not provide countermeasures against instruction counting attacks like *StumbleStepping* as part of the TDX module and instead refers to their Software Security Guidance information [24] to solve this issue on the application level.

We contacted wolfSSL and OpenSSL with our findings concerning leaking nonce bits in their ECDSA implementations. The findings were acknowledged for both libraries. At the time of submission, wolfSSL fixed the vulnerability in version 5.6.6 and assigned CVE 2024-1544. OpenSSL does not assign CVEs for "same physical system side-channel" [40] vulnerabilities but acknowledged it and is working on a fix.

2 BACKGROUND

2.1 TDX

Intel *Trust Domain Extensions (TDX)* [21] is a Trusted Execution Environment (TEE) that protects whole VMs. The protected VMs are called *Trust Domains (TD)*. Figure 1 shows an overview of the TDX architecture. The so-called *TDX module* forms the core of the design. In contrast to regular VMs, the hypervisor needs to invoke the TDX module's *SEAMCALL API* to manage TDs. Crucially, only the TDX module can enter TDs. Exits from a TD return control to the TDX module instead of to the hypervisor. Thus, the TDX module forms a trusted layer between the untrusted hypervisor and the TD.

To protect the TDX module, it resides in a newly-added, protected memory range. Furthermore, TDX introduces a new CPU mode called *Secure Arbitration Mode (SEAM)* that is split into two

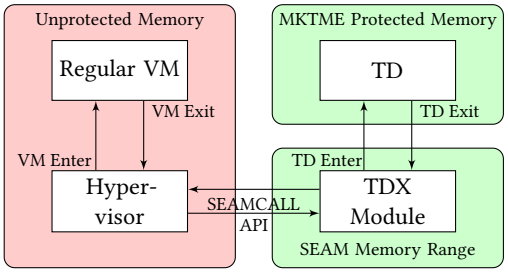


Figure 1: Unlike with regular VMs, the hypervisor does not have direct access to TDs but has to manage them via the TDX module. Based on Figure 5.2 from [21].

sub modes *VMX root* and *VMX non-root*. The TDX module runs in the SEAM VMX root mode, while TDs run in the SEAM VMX non-root mode. To protect against physical attackers the memory used by the TDs is encrypted using *Multi Key Total Memory Encryption (MKTME)* [18]. MKTME allows the use of different encryption keys based on the *KeyID*, an identifier that is encoded by re-purposing the upper bits of the physical address. With TDX, the KeyID range is split into shared and private. Using private KeyIDs is restricted to the new SEAM CPU mode and thus to the TDX module and TDs. In addition, an access right-based mechanism is used for additional security when the CPU is outside the SEAM mode. Reading protected memory returns a fixed pattern to guard against ciphertext side-channel attacks [28, 30]. Writing taints the memory location, leading to a fatal error the next time the TD tries to access it.

2.2 TDX Control Data Structures

The data structures describing a TD are managed by the TDX module [17, Sec. 6]. One such control data structure is the *Trust Domain Virtual Processor State (TDVPS)*, which describes the state of each virtual CPU of a given TD. The memory for the data structure is initially allocated by the hypervisor and then passed to the TDX module, which encrypts the memory with a private MKTME KeyID. Thus, while knowing the memory addresses of the TDVPS, the hypervisor is forced to use the TDX module's API to interact with the content of the data structure.

2.3 Cache Attacks on Intel TDX

As explained in Section 2.1, TDX encrypts the TD's memory with MKTME which has a severe impact on the applicable cache side-channels. Since the MKTME KeyID is encoded in the physical address bits, it is part of the cache tag. As a result, accessing the same data with different KeyIDs would, in theory, lead to different cache tags and thus in different decryptions of the same physical data residing in the cache at the same time. However, a coherency mechanism ensures that an existing entry using a different KeyID is flushed prior to loading the data with the new KeyID. This behavior enables *Flush+Reload* style cache attacks where the attacker accesses TD memory with a different KeyID, to evict it from the cache [1, 22]. Without this mechanism, an attacker could not perform *Flush+Reload* attacks, as they can neither perform flushes nor memory accesses with the TD's KeyID.

2.4 Instruction Counting Attacks

Instruction counting is a single-stepping-based side-channel attack against TEEs that aims to infer fine-grained control flow information. The attacker is assumed to know the executed binary. Then, they combine the coarse grained page fault controlled-channel [58] with a single-stepping attack, to reconstruct the victim's control flow with instruction granularity. Instruction counting attacks can even detect if two code branches of equal length execute memory accesses at different points in the instruction sequence [10, 33]. To guard against such attacks, security critical code, should employ the data oblivious constant-time paradigm: Neither the execution path nor any memory accesses may depend on secret data. While hard to implement, these properties defeat all known side-channel attacks.

2.5 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a variant of DSA on elliptic curves [38]. In order to sign a message, one chooses an elliptic curve $E(\mathbb{F}_p)$ over a finite field $\mathbb{F}_p$ and a generator G of order n. Next, the signer generates a long-term secret key $d \in [1, n-1]$ and the corresponding public key $Q = d \cdot G$, which is a point on the elliptic curve. Finally, for every signature, the signer generates a new unique nonce $k \in [1, n-1]$ which they must keep secret. The signature is then computed over the hash h of the message and comprises the tuple (r, s). The signer calculates $r = x_r \bmod n$ with $(x_r, y_r) = k \cdot G$ and $s \equiv k^{-1}(h + r \cdot d) \bmod n$.

The security of the long-term key d in ECDSA depends heavily on the choice of the nonce k. A slight bias in the randomness of k allows to recover the secret key. While the sampling algorithm used by most ECDSA implementations produces sufficiently random numbers, an attacker can also obtain such a bias via side-channel information [4, 42, 52]. To reconstruct the secret key d from signatures for which the attacker has partial information about k, there are two common approaches. Both first recover k and then use it to compute d. The first approach formulates the problem as a shortest vector problem (SVP) and solves it via lattice reduction techniques like LLL or BKZ [15, 39]. Recently, Albrecht et al. [2] improved the performance of the lattice reduction approach by combining it with enumeration and sieving with predicate, allowing to exploit smaller biases while also requiring fewer biased signatures. The second approach by Bleichenbacher [8] is based on Fourier analysis.

2.6 Attacker Model

We assume the default TEE attacker model, where an attacker with root privileges on the system tries to attack a program running inside the TEE [10, 29, 35, 55]. Most importantly for this work, the attacker is capable of using the page fault controlled-channel, programming the APIC timer and controlling the processor frequency. For our experiments, we disabled all hardware cache prefetchers, Intel *SpeedStep*, as well as hardware controlled P-states. The latter is important to allow the Linux *cpufreq* driver to control the CPU frequency. We do not assume physical access.

This is in line with the Intel TDX attacker model [21]. A real world example is a malicious or compromised cloud service provider that tries to leak data from a customer's TD.

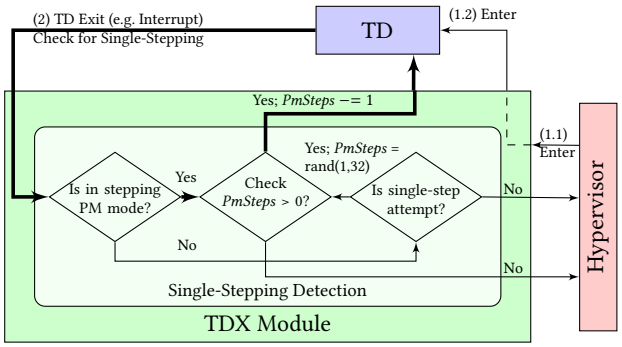


Figure 2: The hypervisor starts the TD by calling the TDX module (1.1) which in turn enters the TD (1.2). When the TD exits (2), the TDX module applies a heuristic to check for single-stepping. If yes, it activates the prevention mode (PM) and re-enters the TD for a randomized number of times (*PmSteps*), before disabling the PM mode, as indicated by the circle in the top left part. On each entry, a special configuration is used to ensure that the TD only executes a single instruction.

3 TDX SINGLE-STEPPING COUNTERMEASURE

Single-stepping is a powerful attack mechanism that has been successfully deployed against major TEEs like SGX [10, 31, 33, 41, 45, 46] and AMD SEV [43, 44, 51, 56, 59]. It uses the attacker's control over the APIC timer to interrupt the TEE after every instruction. Besides enhancing existing side-channel attacks, it can also be used for so-called instruction counting attacks, to reveal the TEE's control flow at the instruction level, allowing the exploitation of minuscule leakages in e.g. cryptographic libraries. Given the devastating effects of single-stepping attacks, Intel TDX comes with a built-in countermeasure which consists of a heuristic to detect single-stepping interrupt patterns and subsequently activates a prevention mode. In the remainder of this section we explain both mechanisms in detail. An overview of the single-stepping countermeasure is given in Figure 2.

TDX Interrupt Architecture. On both, Intel SGX and AMD SEV, single-stepping attacks exploit the fact that external interrupts, like the APIC timer interrupt, abort the execution of the TEE and return control to the attacker controlled OS or hypervisor. Intel TDX, however, has a different design. Neither can the attacker controlled hypervisor directly enter the TD nor do exits from the TD immediately return control to the hypervisor, as discussed in Section 2.1. Instead, there is a trusted intermediate layer, called the TDX module, which runs in a special CPU mode and offers so-called *SEAMCALLs* to the hypervisor for managing TDs. While this design still allows a malicious hypervisor to program the APIC timer such that it interrupts the TD shortly after it is entered, the resulting exit is now handled by the TDX module, as shown in Figure 2. Thus, the malicious hypervisor cannot immediately observe if the TD was interrupted. Since the TDX module is not intended to replace the hypervisor, there are many instances in which the TDX module eventually needs to notify the hypervisor about the TD interruption to allow the expected virtualization behavior. To prevent interrupt-based single-stepping attacks, the TDX module makes an attempt on deciding whether a certain interrupt pattern is benign or if it is part of an attack. In the former case, the TDX module immediately returns to the hypervisor, while in the case of a detected attack it continues to execute the TD for a randomized amount of cycles via the single-stepping prevention mode, before eventually handing back control to the hypervisor.

Single-Stepping Detection. Whenever the TD is exited due to an interrupt, the TDX module measures and evaluates two properties to decide if the current interrupt behavior is benign or if the hypervisor tries to perform a single-stepping attack. In Figure 2 both checks are summarized as "*Is single-step attempt?*".

The first analyzed property is the number of bytes the TD's instruction pointer (RIP) has advanced since the last exit. If RIP has advanced by more than two times the number of bytes required for the longest x86 instruction, the TDX module can be sure that at least two instructions have been executed, i.e., the TD has not been single-stepped [23].

The second analyzed property is the time that has elapsed since the TD was entered via the TDX module. To obtain the elapsed time, the TDX module stores a *rdtsc* timestamp t_B before entering and a timestamp t_A after exiting the TD. If "sufficient" time $t_d = t_A - t_B$ has passed, the current interrupt behavior is classified as benign. In version 1.5 of the TDX module the threshold is set to 4096 *rdtsc* increments [23]. If either at least two instructions have been executed or sufficient time has passed between entries, the behavior is classified as normal. Otherwise, the TDX module activates the single-stepping *prevention mode*.

Prevention Mode. The core idea of the prevention mode is to execute a randomized number of instructions *PmSteps* inside the TD before finally informing the hypervisor about the initial interrupt. To implement this, after detecting a potential single-stepping attempt, the TDX module first disables all interrupts to block the hypervisor from further interfering with the execution of the prevention mode. Since the TDX module runs in the privileged SEAM VMX root mode, the prevented interrupts even include NMIs and SMIs [36, Sec. 1.3.4]. Next, the TDX module enters the TD with the Monitor Trap Flag, causing it to exit after executing exactly one instruction [19, Sec 26.5.2, Vol 3C]. This process is repeated in a loop until *PmSteps* instructions have been executed, as depicted by the circle in the top left of Figure 4.

In essence, the single-stepping prevention mode single-steps the TD *PmSteps* times from the TDX module while preventing the hypervisor from architecturally observing or interrupting the TD. When the control is eventually returned to the hypervisor, it is unaware of the TD's exact progress.

4 SINGLE-STEPPING TRUST DOMAINS

In this section, we demonstrate how to circumvent the single-stepping detection heuristic from the previous section, re-enabling single-stepping attacks on TDX. In essence, we delude the TDX module about the elapsed time between entering and exiting the TD by reducing the frequency of the CPU core running the TD.

Thereby, we exploit that the *rdtsc* timestamp counter's frequency is unaffected by CPU frequency scaling.

4.1 Attack Primitive Description

The TDX module classifies a TD interruption as benign if sufficient time has passed since entering the TD, as explained in Section 3. More precisely, the time span t_d between entering and exiting the TD is measured inside the TDX module by comparing the *rdtsc* value t_B shortly before entering and t_A shortly after exiting the TD. The goal of the attacker is to trick the TDX module into measuring $t_d \geq 4096$ while only executing one instruction in the TD.

On modern Intel CPUs the *rdtsc* timestamp counter is incremented at a constant frequency instead of being tied to the current CPU frequency [19, Sec 18.17, Vol 3B]. We combine this *rdtsc* behavior with the malicious hypervisor's ability to configure the current operating frequency of the core on which the TDX module and the TD are running: Using a constant frequency for *rdtsc* implies that the demanded 4096 timestamp counter increments always take the same wall clock time. Meanwhile, lowering the frequency of the victim core slows down its execution speed. By setting the frequency low enough, entering the TD, executing one instruction and leaving the TD already takes more than 4096 timestamp counter increments. Consequently, with the modified CPU frequency in place, we are able to use the APIC timer to interrupt the TD after every instruction while still ensuring that the TDX module measures $t_d \geq 4096$ and does not trigger the single-stepping prevention.

Reliable Single-Stepping. To use the APIC timer for single-stepping, we need to ensure that the corresponding interrupt hits during the execution of the first instruction. As described in [10, 12, 56], we flush the TD's Translation Lookaside Buffer (TLB) to prolong the execution time of the first instruction and therefore increase the timing window that leads to single-stepping. By choosing the timer such that it arrives rather at the start of the single-stepping window than at the end, we prevent multi-stepping.

While this causes occasional zero-steps, we can detect them by running a cache attack against the code page currently executed by the TD. We use the KeyID-based *Flush+Reload* mechanism from [1, 22], which results in a very strong signal with access timing differences higher than DRAM reads. The hypervisor flushes all cache lines corresponding to the code page before entering the TD and reloads all lines after exiting the TD. A long reload time signals the execution of an instruction within the TD. In the next paragraph, we describe how we obtain the address of the code page.

Finally, similar to SEV-Step [56], we have to modify the hypervisor to suppress virtual APIC timer interrupts while single-stepping the TD. Otherwise, the TD would jump to the corresponding interrupt handler, instead of executing the targeted code.

Adding Spatial Information. For a meaningful interpretation of the single-stepping data, we need to correlate it with the currently executing code in the TD. To achieve this, we use the well known page fault side-channel [29, 33, 35, 37, 45, 55], that leaks both code and data accesses with page granularity, in order to detect the currently executing application via template attacks. In contrast to other TEEs like SGX or SEV, the page tables for the TD's private memory are inaccessible to the attacker, since they are managed by the TDX module. As a result, we cannot modify the access permission bits to force page faults. However, the TDX module still offers a dedicated API that allows the hypervisor to temporarily block access to any TD page, albeit without the ability to only remove individual access permissions from the page.

Zero-Stepping. While we try to minimize zero-stepping for instruction counting attacks, another line of research has shown that it can be used to boost microarchitectural side-channels by allowing to repeatedly measure the effect of the same instruction [48]. For this work, we consider further exploration of zero-stepping attack primitives on TDX out of scope.

4.2 Attack Primitive Evaluation

In this section, we evaluate our primitive for single-stepping TDX with a synthetic target. The experiments were performed on a 5th generation Intel Xeon Gold 6526Y with a base frequency of 2800 MHz. The processor runs the TDX module in version 1.5. We ran the evaluation in a default Ubuntu 23.10 environment and we implemented the code of the attack primitive on top of the Ubuntu Linux kernel in version 6.5 with the official TDX patches. To break the *rdtsc* based time check in the single-stepping heuristic, we configure the CPU frequency of the core running the TD and the TDX module to the lowest possible value of 800 MHz.

To validate that our single-stepping primitive works reliably, we verify that we do not multi-step and that we dependably detect zero-steps. Therefore, we run the loop from Listing 1 in the TD and measure 3 different scenarios: Executing the loop once (8 instructions), nine times (56 instructions) and ten times (62 instructions). We repeat the measurement for every scenario 10 000 times. The code for the single-stepping evaluation purposely contains `NOP`s as these are the shortest instructions and do not load any parameters from memory. For the evaluation we assume that the address of the code page of the target program as well as the addresses pointed to by *r8* and *r9* are known to the attacker.

We evaluate the attack in the release TDX mode as well as in debug mode. In debug mode, the TDX module offers additional API calls, e.g. reading the current instruction counter. Thereby, we can immediately check that no multi-steps occur and that all zero-steps are detected by the cache attack. In release mode, we check that after filtering zero-steps, the remaining event count matches the expected number of instructions. Again, we do not encounter any errors. We did not observe meaningful differences between the two modes with regard to single-stepping. When running the three evaluation scenarios we execute 1 260 000 instructions and observe only 0.8% zero-step events.

5 STUMBLESTEPPING: LEAKING INSTRUCTION COUNTS

In this section, we describe a second attack primitive, that we dub *StumbleStepping*, allowing an attacker to perform instruction counting attacks against the TD. As discussed in Section 2.4, instruction counting attacks are commonly used to exploit secret-dependent control flow in, e.g., cryptographic libraries. *StumbleStepping* works, even if the single-stepping attack from the previous section is mitigated, i.e., the TDX module correctly activates the single-stepping prevention mode. In contrast to our single-stepping attack, it exploits a conceptual weakness of the countermeasure instead of a

Listing 1: Evaluation target for single-stepping.

```
  mov  qword ptr[r8], 42
loop_label:
  dec rax
  nop
  nop
  nop
  nop
  jnz loop_label
  mov qword ptr [r9], 42
```

Listing 2: Evaluation target for *StumbleStepping*.

```
  mov  qword ptr[r8], 42
loop_label:
  dec rax
  jnz loop_label
  mov qword ptr [r9], 42
```

Figure 3: Synthetic programs for evaluating the stepping primitives from Section 4 and Section 5. We block access to the memory locations pointed to by *r8* and *r9*, using the resulting page faults as the start and end trigger for the attack.

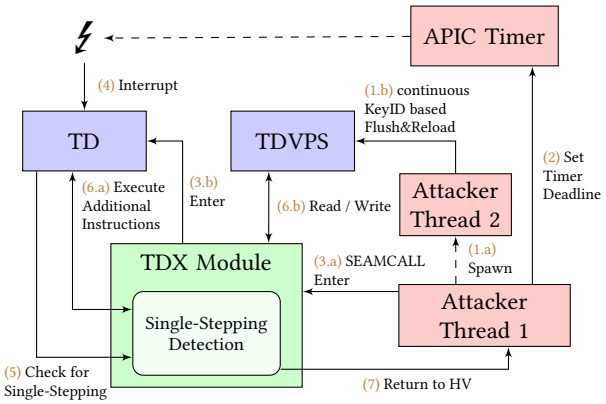


Figure 4: Overview of the steps required for APIC timer-based *StumbleStepping* attacks against TDX. The *Attacker* spawns a new thread to concurrently probe the pages with the victim's TDVPS using the KeyID-based *Flush+Reload* attack. The double ended arrow in step (6.a) represents the prevention mode (c.f. Section 3) and means that the TDX module re-enters the TD several times, before eventually resuming with step (7).

flawed checking logic. In essence, *StumbleStepping* turns the prevention mode upon itself and employs a cache attack to leak the number of instructions executed by the TD. An overview of the attack procedure is shown in Figure 4.

5.1 Attack Primitive Description

The core idea of *StumbleStepping* is to employ a side-channel attack (1.b) against the single-stepping prevention mode implemented inside the TDX module. The side-channel attack runs in parallel to the execution of the TDX module, on a separate core (1.a). In contrast to the single-stepping primitive, we do not want to avoid detection but deliberately trigger the countermeasure. After detecting a potentially malicious interrupt pattern (4 and 5), the single-stepping prevention mode of the TDX module re-enters the TD several times (6.a), before returning control to the hypervisor (7). The TDX module configures the TD such that on each entry the TD executes only a single instruction (6.a). For *StumbleStepping*, we exploit that the TDX module leaks the number of times it re-enters the TD via a cache side-channel (1.b). For each TD entry (6.a), the TD's TDVPS pages are accessed (6.b). This data structure describes the state of the TD, e.g., the vCPU's register file. Note that the information stored in the TDVPS pages is encrypted and thus inaccessible to the hypervisor. However, by running a cache attack in parallel to the execution of the countermeasure inside the TDX module, the hypervisor can leak the number of accesses to the TDVPS pages and thus the number of instructions executed by the TD. To continuously leak the number of executed instructions, the hypervisor sends APIC timer interrupts such that the countermeasure mode is always active (4). For the cache attack, we again use the KeyID-based *Flush+Reload* mechanism.

Improving Temporal Resolution. To obtain reliable information, we require a high temporal resolution for our cache attack, such that we do not miss any of the TDX module's accesses to the *TDVPS* structure. Thus, we only monitor a single cache line of the *TDVPS*. In addition, we again decrease the CPU frequency of the core running the TDX module while setting the frequency of the attacker's core to the highest possible value, increasing the effective sampling rate of our attack.

However, in order to ensure that we do not accidentally trigger the single-stepping attack from the previous section, circumventing the activation of the single-stepping prevention mode, we cannot clock down the TD's core to the lowest possible value of 800 MHz. Instead, we have to keep it running above 1.6 GHz. Thus, once the single-stepping detection heuristic has been fixed, the temporal resolution could be doubled by using the lowest frequency.

Adding Spatial Information. As with the single-stepping primitive in Section 4, we use the page fault controlled-channel to correlate the information from *StumbleStepping* with the currently executing code page for a meaningful interpretation. The randomized bursts in which *StumbleStepping* executes the TD prevent the attacker from terminating the attack at a arbitrary instruction. However, we can exploit that the TDX module aborts the single-stepping prevention mode upon page faults to precisely stop the execution at a defined code location.

In summary, combined with page fault information, *StumbleStepping* reveals the TD's control flow with intra-page resolution, allowing to exploit minuscule secret-dependent control flow leakages. In contrast to single-stepping, it does not allow to pause the execution after every instruction.

5.2 Attack Primitive Evaluation

For evaluating the *StumbleStepping* primitive, we performed all experiments remotely on an Intel provided machine with a TDX enabled 4th generation Xeon Platinum 8480CTDX processor. Furthermore, we verified that the attack primitive still works on a 5th generation Intel Xeon Gold 6526Y which introduces public availability of Intel TDX. The 4th generation CPU used the TDX module software version 1.0 and the 5th generation CPU used version 1.5. For the 5th generation Intel Xeon processor, we ran the evaluation in a default Ubuntu 23.10 environment and implemented the code of the attack primitive on top of the Ubuntu Linux kernel in version

6.5 with the official TDX patches. The evaluation on the 4th generation CPU was conducted on Ubuntu 22.04 with kernel version 5.19. We evaluate *StumbleStepping* with a synthetic target.

Profiling TDVPS Accesses. For *StumbleStepping*, we exploit that each TD entry leads to accesses to the TDVPS data structure which we want to observe via a cache attack in parallel to the execution of the TDX module. We again use the KeyID-based *Flush+Reload* mechanism and measure between 600 and 1000 cycles when accessing a cache line that has previously been accessed by the TDX module, which is much higher than the DRAM access time caused by a regular cache attack. To maximize the temporal resolution, we only observe one cache line of the TDVPS structure. We observe, that the number of observed cache misses per TD entry varies depending on the monitored offset inside the TDVPS pages. In an offline profiling step, we determine the offset with the lowest amount of noise, by running *StumbleStepping* against a calibration target several times, sweeping over every cache line aligned offset. On our machine, offset *0x128* in the third TDVPS page gives a stable correlation, with two accesses per TD entry.

Accuracy. To evaluate the accuracy of our counting primitive, we use the synthetic code snippet from Listing 2. We choose a loop with only one instruction instead of an if-else construct as it allows us to easily scale the number of executed instructions while still allowing differences as small as two executed instructions between two runs. For the evaluation, we assume that the memory locations pointed to by *r8* and *r9* are known to the attacker.

Figure 5 shows the resulting data for 1 to 5 loop iterations which corresponds to 4 to 12 executed instructions. Figure 6 shows the data for 100 to 105 loop repetitions which corresponds to 202 to 212 instructions. The results clearly show that the measurement noise increases when we observe longer program sequences. However, the distributions for different iteration counts only partially overlap and the means are easily distinguishable. Thus, when using only a single measurement, there is a certain error probability when trying to distinguish events with almost the same amount of executed instructions. However, repeating the measurement multiple times eliminates the error. For events with larger instruction differences, a single measurement is sufficient.

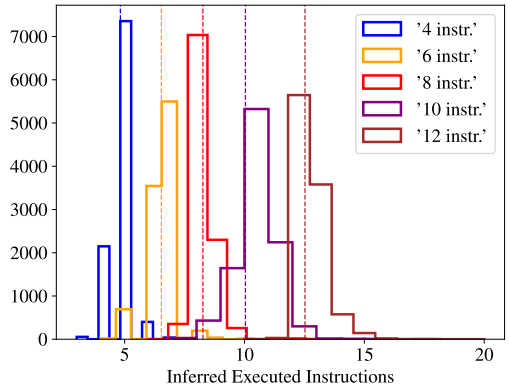


Figure 5: Inferred instruction count for 1 to 5 repetitions of the loop from Listing 2 repeating using 10 000 measurements. The dotted lines show the mean value.

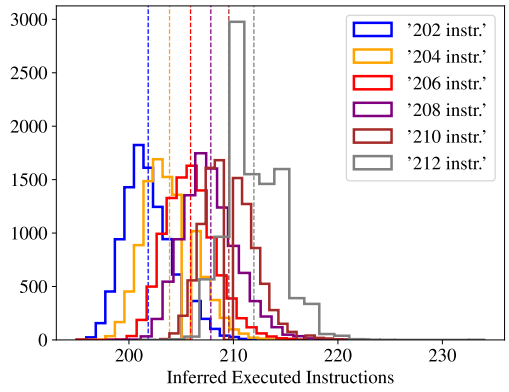


Figure 6: Inferred instruction count for 100 to 105 repetitions of the loop from Listing 2 using 10 000 measurements. The dotted lines show the mean value. We removed a total of 17 outliers above 300 inferred instructions.

6 LEAKING ECDSA KEYS FROM BIASED NONCE TRUNCATION

As discussed in the preceding sections, single-stepping attacks are frequently used to leak secret-dependent control flow. To protect against such attacks without relying on countermeasures employed by the TEE, cryptographic libraries should use the data oblivious constant-time programming paradigm. However, developing constant-time code at the instruction level is a challenging task. In this section we present, in detail, a control flow-based leakage during the derivation of the random and secret nonce k of the ECDSA signing process.

In essence, there are two established ways to generate a random nonce mod n: A modular reduction-based truncation of the randomly generated value or rejection sampling of random values until a value $k < n$ is drawn. Implementations of the latter method usually do not leak a nonce bias. However, implementations of the former are more prone to leak information, as they require a division which is more complex to implement in a side-channel resistant manner. Both methods are listed in the FIPS digital signature standard [38, Sec. A.3.1, A.3.2].

While Weiser et al. [52] already discussed this leakage in modular reduction-based truncation, they deemed it negligible and did not further investigate it. We analyze this leakage in full detail and show that, depending on the curve, in fact up to 15 bits of the nonce are leaked. Additionally, we systematically investigate the usage of truncation for nonce computation in multiple cryptographic libraries, finding leakages in wolfSSL and OpenSSL. We evaluate the introduced leakage and its exploitability depending on the curve and the curve's modulus.

Root Cause. To ensure that the nonce k is smaller than the curve's modulus n, both wolfSSL and OpenSSL use truncation via modular reduction of a random byte string. The byte string has a bit

Table 1: ECDSA nonce generation in different libraries.

Library	Version	Nonce Derivation	Vuln.	c'time version inc.
wolfSSL	5.6.4	truncation	yes	limited[1]
OpenSSL	3.2.0	truncation	yes	no[2]
Nettle	3.9.1	rejection sampling	no	N/A[3]
Mbed TLS	3.5.1	rejection sampling	no	N/A[3]
botan	3.2.0	rejection sampling	no	N/A[3]
nss	3.9.4	rejection sampling	no	N/A[3]

[1] Only for curves secp256,384,521; not enabled by default
[2] Constant-time variant not yet implemented
[3] Not applicable; the default is rejection sampling

length greater than the bit length of the curve order n. Next, both libraries perform a modular reduction, reducing the random byte string to a value smaller than n. Therefore, on a high level, both libraries consider the two top most words of the numerator k_{top} and the top most word of the denominator n_{top}. Next, they compute $q_{top} = \frac{k_{top}}{n_{top}}$ to estimate the quotient $q = \frac{k}{n}$. Afterwards, they check whether $n \cdot q_{top} > k$. If this condition is true, q_{top} is decremented. This decrementation is implemented as a loop, meaning the number of times q_{top} has to be decremented is reflected in the execution count of the loop. The number of loop iterations in turn can be observed by a side-channel attacker and leaks information about the nonce's most significant bits.

Investigated Libraries and Curves. Table 1 lists all libraries we investigated during this work and whether they use truncation or rejection sampling. Of the analyzed libraries, only wolfSSL and OpenSSL use truncation. We initially found the leakage by analyzing wolfSSL with Microwalk [53, 54]. Using the obtained knowledge, we were able to analyze the remaining libraries manually.

The remainder of this section is structured as follows. First, we give details on our analysis methodology. Next, we present the discovered leakages in wolfSSL and OpenSSL in more detail and discuss their exploitability.

6.1 Analysis Methodology

Before giving the results on the individual implementations in the analyzed libraries, we describe our analysis workflow.

Simulated Side-Channel Traces. In order to calculate the maximum obtainable information and plot the bias introduced to the nonce k, we simulate side-channel traces by adding counters to the targeted code, to observe the occurrence of certain control flow events. We give more details on these events in the next sections. For each curve and library we collect 10 million signatures. Per signature, we store the values of the injected event counters, the signed hash h, the ECDSA signature values r and s, as well as the nonce k. For the latter, we again modify the libraries as the nonce is not usually exported. We stress that these modifications *do not* introduce secret-dependent changes to the control flow and thus do not influence the code's leakage properties. Afterwards, we divide the collected samples into sets, one set per observable control flow event combination. Within each set, we analyze the distribution of the bit values of k. We refer to a distribution of nonce bit values simply as distribution.

Table 2: Maximum obtainable leakage in terms of mutual information (MI) and fully leaked bits (FB) for different curves in wolfSSL and OpenSSL. The MI values are rounded. The full bit (FB) value reports on those bits which have the same value for all nonces in a distribution. The event column specifies the event combination which corresponds to the leakage and the probability of the event combination.

	wolfSSL		OpenSSL	
Curve	Event	MI / FB	Event	MI / FB
	(W_1, W_2) $Pr[A = a]$	[bit / bit]	(O_1, O_2) $Pr[A = a]$	[bit / bit]
bp224r1	(2, *) 0.09	1.6 / 1	(1, 0) $1.6 \cdot 10^{-4}$	7 / 6
bp320r1	(3, *) < 0.002	3 / 3	(2, *) $1.7 \cdot 10^{-3}$	3 / 3
bp384r1	(2, *) 0.05	3.5 / 0	(1, *) 0.05	3.5 / 0
secp160r1	(2, *) $1.5 \cdot 10^{-5}$	15.6 / 15	(1, *) $1.3 \cdot 10^{-5}$	15.8 / 15

Leakage Quantification. To quantify the leakage, we calculate I, the mutual information (MI) per distribution. Therefore, we use $I = \sum_{i=0}^{i<\text{bitlen}(G)} H(B) - H(B|A = a)$ with B being the random variable describing a single bit value over the alphabet $\{0, 1\}$, A the random variable describing the distribution of nonces, and G the generator of the curve. The number of distributions per curve depends on the number of discernible events. The probability $Pr[A = a]$ of a nonce falling into one of the distributions is calculated by dividing the number of samples with a specific event combination by the total number of signatures collected for the curve.

Since we subdivide all nonces recorded during sampling into disjoint sets, we are interested in the overall information gain on all nonce bits per distribution rather than the gain over all distributions. Thus, we do not sum over all distributions when calculating the MI but only consider the distribution of the considered event combination.

While the MI precisely captures the leakage from an information theoretic point of view, most key reconstruction algorithms require knowing the value of individual bits with high certainty. Thus, we also analyzed which bits of each nonce always have the same value for a given event combination. We call these *full bits*. Comparing MI and *full bits* gives an insight on how much of the MI is distributed over small biases in different bits. All leaking bits are most significant bits (MSB). We analyzed the curves secp128r1, secp160r1 and secp192r1 as well as the R1 Brainpool curves for 160, 192, 224, 256, 320 and 384 bits for wolfSSL and OpenSSL.

The most important findings are summarized in Table 2 and the results for the remaining curves and control flow events can be found in Table 3 in Appendix A. Per curve and library, we specify the

Listing 3: Simplified version of the leaking _sp_div_impl (wolfssl/wolfcrypt/src/sp_int.c) function which divides *a* by *d* and is called during the nonce generation. The snippet is not self-contained and only intended to highlight the control flow. The colored *Event* comments mark points in the control flow that leak information about the nonce.

```
1   int _sp_div_impl(sp_int* a, d, r, trial) {
2
3    for (i = a->used - 1; i >= d->used; i--) {
4     //Calculate trial quotient
5     t = sp_div_word(a->dp[i], a->dp[i-1], dt);
6     do {
7       for (j = 0; j < d->used; j++) {...}
8       for (j = d->used; j > 0; j--)
9           //Event W2
10          if (trial->dp[j] != a->dp[j + o])
11              break;
12
13      if (trial->dp[j] > a->dp[j + o]) { t--; }
14      //Event W1
15     }while (trial->dp[j] > a->dp[j + o]);
16    }
17  };
```

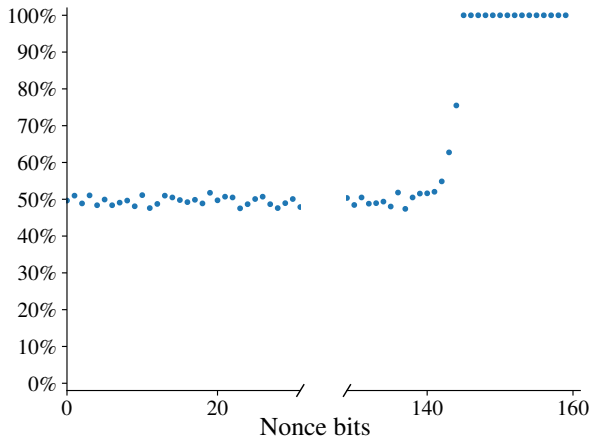


Figure 7: Distribution of the nonce bits for curve secp160r1 in wolfSSL given event ($W_1 = 2, W_2 = *$). The y-axis shows the percentage of nonces for which the value of the corresponding bit is 1. The 15 most significant bits are always 1.

control flow events which cause leakage in the event column. In the next two sections, we describe the leakages and the corresponding events in more detail.

6.2 Nonce Leakage in wolfSSL

For analyzing the leakage in wolfSSL, we use the default compile configuration with additional hardening parameters and options to enable smaller ECC curves as well as Brainpool curves. While the default implementation of wolfSSL's math functionality is supposedly constant-time [57], the default ECC sign functionality makes use of truncation for generating k. With additional, non-default compiler flags, wolfSSL includes implementations which use rejection sampling, but these are only available for the curves secp256, secp384 and secp521.

Leaking Control Flow Events. A simplified version of the algorithm for nonce truncation in wolfSSL is shown in Listing 3. Event W_1 in line 14 describes the number of times the do-while loop was executed and thus how often the estimated quotient was decremented. As a shorthand, we use $W_1 = x$ if W_1 occurred x times. The event W_2 counts which words of the estimated quotient and nominator are relevant for the comparison to decide on the decrementation of the variable `t`.

Leakage Quantification. Figure 7 shows the bias introduced to the nonce when $W_1 = 2$, i.e. there are two iterations of the do-while loop. While this event only happens with a probability of approximately $1.5 \cdot 10^{-5}$, it reveals that 15 MSBs of the nonce are 1. Note, the bit length of the curve order of secp160r1 is 161 bit, contrary to what the name suggests. However, the order's MSBs are all 0, except for the most significant bit. Thus, the order of secp160r1 is only slightly larger than 2^{160}, meaning the likelihood of a k with 161 bits is very small. Within the 10 million samples we collected, most (about 50%) have a k of size 160 bit, but there was no sample with a k of size 161 bit. Consequently, we assume the most significant bit of k to be 0 and known by default.

The curve brainpoolp320r1 and brainpoolp384r1 show a leakage of 3 bit and 3.5 bit. The distributions are shown in Figure 10 in the Appendix. Though the brainpoolp384r1 curve does not leak any bit without error, i.e. no *full bits* (c.f. Section 6.1), there is less than 2% error in each of the biases of the 4 top most significant bits. For the brainpoolp224r1 curve, wolfSSL only shows negligible leakage.

Leakage Exploitability. The leakage observed for secp160r1 is exploitable with conventional LLL reduction techniques. In Section 7.3 we demonstrate the key reconstruction for secp160r1 from real side-channel traces as a case study for *StumbleStepping*.

For evaluating the exploitability of the leakages for the curves brainpoolp320r1 and brainpoolp384r1, we use the predicate with sieving technique from Albrecht et al. [2] and extend their implementation to also support MSB prefixes containing bits other than 0, as the leaked MSB prefixes are `0b110` and `0b1000`, respectively. The work of Albrecht et al. suggests, that 4 bits are required to reconstruct keys for 384 bit curves. Thus, our 3.5 bit leakage in the brainpoolp384r1 curve is a borderline case. However, our data shows that we can also use the 4 MSBs as *full bits*, accepting a small additional error. The error can be countered by resampling the subsets of the obtained signatures used for reconstruction and running the key reconstruction multiples times with different subsets.

While the key reconstruction terminates in a reasonable time, it never succeeds. To verify, that the error is not due to the small error probability of the individual bits, we also performed additional key reconstruction experiments on simulated data: We simulate the leading 4 bit `0b1000` leakage from our side-channel experiments without errors. However, the reconstruction does not succeed either. Using a simulated, error free 5 bit leakage, the reconstructions succeeds. To verify the correctness of our changes to the implementation of Albrecht et al. [2] we validate that the reconstruction for a simulated 4 bit non-zero MSB leakage for the NISTP384 curve, which they used as a benchmark, succeeds. Since this validation

Listing 4: Simplified version of the leaking bn_div_fixed_top (openssl/crypto/bn/bn_div.c) function which divides *num* by *divisor* and is called during the nonce generation. The snippet is not self-contained and only intended to highlight the control flow. The colored *Event* comments mark points in the control flow that leak information about the nonce.

```
1  int bn_div_fixed_top(BIGNUM* dv, rm, num, divisor,
2    BN_CTX *ctx) {
3
4  for (i = 0; i < loop; i++, wnumtop--) {
5    for (;;) {
6      if ((t2h < rem) ||
7          ((t2h == rem) && (t2l <= n2)))
8        break;
9      //Event O1
10     q--;
11     rem += d0;
12     if (rem < d0) //don't let rem overflow
13       break;
14     //Event O2
15     if (t2l < d1)
16       t2h--;
17     t2l -= d1;
18   }}};
```

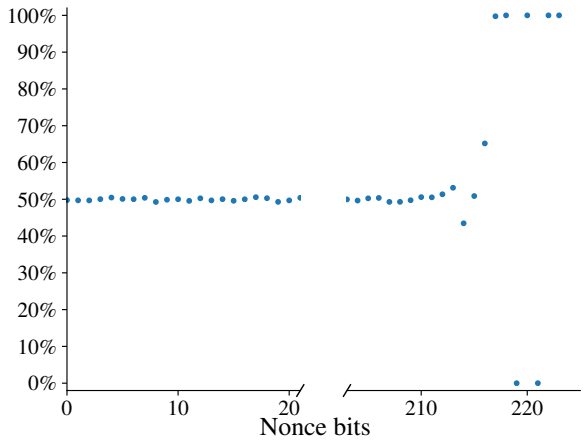


Figure 8: Distribution of nonce bits for brainpoolp224r1 curve in OpenSSL given event ($O_1 = 1, O_2 = 0$). The y-axis shows the percentage of nonces for which the value of the corresponding bit is 1. The 6 most significant bits are 0b110101 for all samples. The 7th bit is 1 for more than 99% of all samples.

was successful, we assume that more than 4 bits are required for brainpoolp384r1, in contrast to NISTP384.

The 3 bit leakage for the brainpoolp320r1 was too small to be exploited with the methods of Albrecht et al. in our experiments.

6.3 Nonce Leakage in OpenSSL

For analyzing OpenSSL, we compile it with the default configuration, which uses truncation with modular reduction to compute the nonce k. From the OpenSSL code and corresponding comments, we could infer that it is envisaged to implement the computation of the estimated quotient in constant-time. However, this feature is not used and during the course of the responsible disclosure we learned that it is not implemented.

Leaking Control Flow Events. We show a simplified version of the procedure used for the division during nonce truncation in Listing 4. It is comparable to the procedure used in wolfSSL, however, contains slightly different observable side-channel events. Event O_1 in line 9 describes how often the estimated quotient is decremented. Additionally, we observe the event O_2 in line 14 which describes whether the remainder of the division overflows during the procedure of decrementing the variable q. We do not consider the if-clause following event O_2. Though it changes the control flow we did not observe any differences in the resulting bit distributions when using it as a differentiator.

Leakage Quantification. In Figure 8, we show that there is a 7 bit leakage for curve brainpoolp224r1. As detailed in Table 2, only 6 of these 7 bits are *full bit* leakages. However, since the error for the 7th bit is small, we can integrate it into the key reconstruction as well. This leakage is only observable in the OpenSSL implementation as a differentiation by the event O_2 is required. For the secp160r1, brainpoolp320r1 and brainpoolp384r1 curve, OpenSSL shows similar leakage as wolfSSL.

Leakage Exploitability. The 7 bit leakage for curve brainpoolp224r1 is exploitable. To reconstruct the key, we use our extended variant of the predicate with sieving technique from Albrecht et al. [2]. We present the result in our single-stepping case study in Section 7.2. The maximum leakage and exploitability for the curves secp160r1, brainpoolp320r1 and brainpoolp384r1 corresponds to the analysis in Section 6.2.

6.4 Leakage Analysis Summary

We observe that the truncation of the secret nonce k leaks a varying number of most significant bits, depending on the order of the ECDSA curve. The order of the curve serves as denominator during nonce truncation. While we observe only small leakages for curves with an order that consist of only 1 valued bits in the MSBs, we see large leakage in the opposite case, i.e., few 1 valued bits in the MSBs of the curve's order.

In contrast to what is reported in previous work [52], we find that the bias introduced through modular reduction during nonce creation is not always negligible, but depends on the order n of the curve. We show that in certain situations, a substantial bias is introduced and observable through side-channels. Additionally, note that FIPS 186.5 [38] states in A3.1 that implementations which use truncation during nonce creation shall use an additional 64 bit of randomness to avoid a bias to k. While wolfSSL is following this advice, k is still biased. We assume that the advice in FIPS 186.5 refers to the overall distribution of k, but does not take into account additional side-channel information.

7 CASE STUDIES

In the following we evaluate both our attack primitives on real-world cryptographic libraries and demonstrate their ability to leak the ECDSA nonce k, allowing us to reconstruct the private key. Our attack targets are the nonce leakages described in the previous

section. We first explain our attack approach in general and then give details for the specific primitives and attacks targets.

7.1 Attack Approach

The general attack approach is the same for both attacks and splits into an online and an offline phase.

Offline Phase. In the offline phase, we build the mapping from the number of observed instructions per trace to the occurrence count of the events. Additionally, to be able to use single-stepping and *StumbleStepping* we need to find page fault trigger points, such that the number of instructions executed between the trigger points allows us to infer the number of times the control flow passes the observed event. To infer when the victim is about to be executed, we generate a page fault based template. For all tasks, we use a semi-automated approach combining static binary analysis and dynamic binary instrumentation.

Online Phase. In the online phase, the attacker first needs to determine the guest physical addresses of certain functions inside the target in order to instantiate the page fault sequence template from the offline phase. Afterwards, they can use the template to start single-stepping or *StumbleStepping* for tracing the TD when the targeted code is about to be executed. This allows us to count the executed instructions between the trigger points. For the evaluation, we streamlined the attack scenario by calling the target libraries from a self-written program, that triggers the signature generation and supplies the attacker with the guest physical address of the target library. As several works [28–30, 35] against AMD's confidential VM solution SEV, as well as confidential VM like systems in general [9], have already demonstrated that an attacker can locate applications in memory by observing access patterns, it is a valid assumption that the attacker can infer the guest physical addresses for the page fault template. We want to stress that the addresses used for the template are only from the target library, not from the calling application. To maximize the performance, we implemented our attack logic inside the Linux KVM hypervisor kernel module.

7.2 Single-Stepping brainpoolp224r1

The first case study shows the reliability and high resolution of our single-stepping primitive. We extract the private ECDSA key from the side-channel leakage in the nonce generation process for the brainpoolp224r1 curve in OpenSSL as described in Section 6. The attack was executed on the same platform as the single-stepping evaluation. The possible event combinations for brainpoolp224r1 in OpenSSL are (O_1, O_2): $(0, *)$, $(1, 0)$, $(1, 1)$. The event $(1, 0)$ corresponds to signatures with the nonce bias required for our attack. This event corresponds to leaving the inner for-loop in Listing 4 before event $O_2 = 0$ in Line 14 but only after executing event $O_1 = 1$ in Line 9 once.

Offline Phase. Due to the code structure, we cannot use page accesses to distinguish the events. Instead, we use page accesses shortly before and after the loop to trigger single-stepping. Using our trigger points, we measure 32 steps for the event $(0, *)$, either 38 for 39 steps for the nonce bias event $(1, 0)$, and 42 or 43 steps for the event $(1, 1)$. The variable amount of steps for the events $(1, 0)$ and $(1, 1)$ is caused by the *or*-condition in Listing 4 before the event O_1 and some code restructuring by the compiler.

Online Phase. As explained in the attack approach, we first obtain the guest physical address for the attacked code sequence in OpenSSL to instantiate the page fault template which we use to single-step only the execution of the nonce truncation. Our attack code requires on average 32.98 ms per signature. Without an ongoing attack, a signature requires on average 0.33 ms.

To recover the key, we need 33 signatures with the nonce bias event $(O_1, O_2) = (1, 0)$, i.e. 38 or 39 counted steps. Given the low probability of the event, we need to observe 170 000 signatures. Collecting all signature traces takes approximately 94 minutes. The reconstruction of the long-term key is conducted as described in Section 6.3 and requires 1.5 seconds on an Intel Xeon E-2286M.

7.3 StumbleStepping secp160r1

In this section, we exploit the nonce leakage in the secp160r1 curve from Section 6 with *StumbleStepping*. We choose wolfSSL as target for the attack and run the experiments on the same platform used for the evaluation of the *StumbleStepping* primitive.

The possible event combinations for secp160r1 are (W_1, W_2): $(1, 1)$, $(2, 1)$, $(2, 2)$. The event W_1 describes the number of times the do-while loop in Listing 3 is executed. The events $(2, 1)$, $(2, 2)$ correspond to signatures with the nonce bias required for our attack.

Offline Phase. Due to the code structure, we cannot use page accesses to distinguish the events. Instead, we use page accesses shortly before and after the outer loop to trigger *StumbleStepping*. Using our trigger points, the events $(1, 1)$, $(2, 1)$, $(2, 2)$ correspond to 178, 230 and 239 executed instructions.

Online Phase. As explained in the attack approach, we first determine the required guest physical address for the attacked code sequence in wolfSSL, instantiate the page fault template and then start the *StumbleStepping* primitive to trace only the execution of the nonce truncation. Our attack code requires on average 4.77 ms per secp160r1 signature. Without an ongoing attack, a signature requires on average about 0.01 ms.

To recover the key, we require 12 signatures with a biased nonce, i.e. two occurrences of the event W_1. Since the probability for two occurrences of event W_1 is very low, we need to observe 700 000 signatures. Collecting all signature traces takes approximately 56 minutes. The measurement results are shown in Figure 9. As expected from the evaluation of the toy examples in Section 5.2, the measurements contain some noise. Still, we are able to distinguish the two relevant event groups, which differ by 52 instructions, without errors. Using the LLL approach described in Section 2.5, the key recovery finishes in 1.7 seconds on an Intel Xeon E-2286M.

8 DISCUSSION

In this section we suggest improvements to the current single-stepping countermeasure design and discuss limitations of our *StumbleStepping* primitive.

8.1 Improved Single-Stepping Detection

We propose to only rely on the progress of the instruction counter to detect single-stepping. As discussed in AEX-Notify [12], single-stepping requires the attacker to artificially slow down the execution of the first instruction, e.g. via a TLB flush. The authors further show that consequently the attacker cannot reliably interrupt the

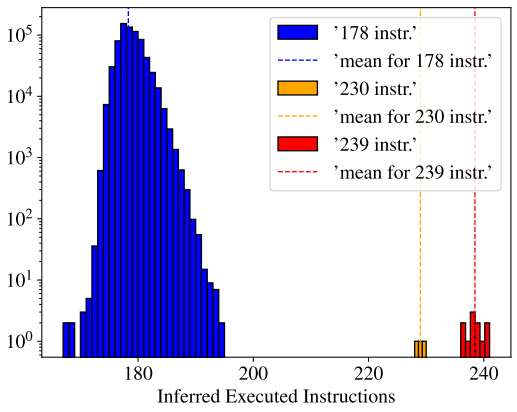


Figure 9: Side-channel data for the *StumbleStepping* attack on the secp160r1 curve in wolfSSL. The legend states the actual number of executed instructions while the x-axis shows the inferred number of instructions. In total, we collected data for 700 000 signatures.

SGX enclave after the second instruction, i.e. the attacker cannot "two-step". Based on these results, changing the heuristic to enforce that at least two instructions have been executed prevents repeatedly interrupting the TD after x instructions. If less than two instructions have been executed the prevention mode gets activated.

8.2 Improved Single-Step Prevention Mode

In this section, we propose changes to the single-stepping prevention mode, to mitigate the instruction count leakage. For our *StumbleStepping* attack in Section 5, we observe memory accesses to the TD's TDVPS management data structure to infer the number of executed instructions. Thus, one could consider adding additional accesses to this data structure from the TDX module to introduce noise to any potential side-channel measurements relying on these accesses. However, the fact that each iteration of the invocation of the TDX single-stepping prevention mode requires a TD entry and exit exposes a large microarchitectural attack surface, potentially allowing for other measurable effects. For example, simply measuring the time between the APIC timer interrupt firing and control being handed back to the hypervisor already reveals coarse grained information about the number of instructions executed by the TD.

Thus, as a more profound solution, we propose to extend the Monitor Trap Flag mechanism currently used when executing the TD in single-stepping prevention mode. Instead of trapping after one instruction, the mechanism could directly support to execute a randomized number of instructions in one burst. As a result, only a single TD entry is required regardless of the number of instructions executed by the single-stepping prevention mode, mitigating the instruction count leakage at its root.

8.3 AEX-Notify based Countermeasure

Orthogonal to the TDX single-stepping countermeasure, that is split into detection and prevention, it should also be possible to port the countermeasure from AEX-Notify[12] to VM-based TEEs like TDX. They execute a special interrupt handler that prefetches the first instruction to undo any artificial slowdowns that would be required for single-stepping. Since VMs are already interrupt aware, it should be possible to simply execute this interrupt handler every time the TD is entered. With the original AEX-Notify design, the security of the TEE depends on the runtime inside the protected enclave to use their interrupt handler. With TDX, this could be improved by moving the prefetching step to the TDX module, instead.

8.4 Limitations of StumbleStepping

Compared to instruction counting attacks that use single-stepping, as e.g. CopyCat [33], the *StumbleStepping* attack from Section 5 provides slightly weaker leakage. Since we cannot pause the TD after every instruction, we cannot distinguish balanced if-else branches that only differ in the relative order of their memory accesses. This is only possible with single-stepping, as it allows removing page access rights after every memory access instruction.

8.5 Attack Overhead

The case studies in Section 7.2 and Section 7.3 show different relative overheads introduced to the signature computation time by the attack code. While the single-stepping attack on OpenSSL in Section 7.2 slows down the execution approximately by a factor 100, the signature creation with a running StumbleStepping attack on wolfSSL in Section 7.3 is roughly 500 times slower.

These differences can be attributed to multiple factors. First, we use different single-stepping mechanisms in both primitives. While StumbleStepping implicitly single-steps the TD by setting the MTF flag, the single-stepping primitive uses the APIC timer. Furthermore, the observed instruction sequences have different lengths and finally, the page fault sequences required to trigger the attack have different lengths.

9 RELATED WORK

We start this section, by reviewing existing security flaws found in TDX before giving a summary of existing attacks on ECDSA.

9.1 TDX

To the best of our knowledge, this is the first academic paper attacking the Intel TDX single-stepping countermeasure. However, Intel commissioned several security reviews [1, 20] to assess and improve the security of TDX.

Single-Stepping. In Intel's security review [20], a straightforward single-stepping attack against an early TDX version was developed. However, Intel states that this is mitigated since TDX module version 1.0. Our evaluation targets use version 1.0 and version 1.5 and thus break Intel's countermeasure. During the disclosure process, Intel stated TDX module versions after 1.5.0.6 will contain additional security measures.

Page Fault Controlled-Channel. The authors of [1] discuss that the memory blocking feature of the TDX API can be used to

implement page fault controlled-channel attacks, recovering the TD's control flow with page granularity.

Cache Attacks. In [1] they also describe how the MKTME KeyID in combination with the cache coherency protocol enables *Flush+Reload*-style cache attacks on TDX. In addition, they state that the *monitor* and *mwait* instruction can be used to implement cache attacks, similar to [60].

9.2 ECDSA Key Recovery

Weiser et al. [52] perform a systematic study of ECDSA nonce leakages. They already discovered the leakage described in this work, but classify it as negligible and do not further investigate it. However, our results show that the leakage depends on the curve order and that it introduces large biases for some curves. CopyCat [33] uses instruction counting with SGX to exploit side-channels in modular inversion and elliptic curve scalar multiplication. In TPM-Fail [34], the authors also exploit the elliptic curve scalar multiplication, however in the context of TPM implementations. In Ladderleak [4], the authors use a timing side-channel in combination with roughly half a billion signatures for a 163 bit curve to exploit nonce leakages smaller than 1 bit, building on Bleichenbacher [8]. Moreover, Ryan [42] investigates leakages introduced through non-constant-time implementations of the modular reduction of $r \cdot d$ and $r \cdot d + h$.

10 CONCLUSION

Intel's most recent TDX TEE comes with a built-in countermeasure against single-stepping attacks. In this work, we have demonstrated the first attacks against this countermeasure. We developed two attack primitives: Single-stepping TDs by outwitting the detection heuristic and counting the TD's instructions with *StumbleStepping*. The former fully breaks the countermeasure by manipulating the CPU frequency to pass a time check in the single-stepping detection heuristic. The second attack exploits the side-channel properties of the single-stepping countermeasure, revealing a systematic flaw in the current design that leaks the number of executed instructions via a cache side-channel. We propose design changes to mitigate both attacks. As a second major contribution, we have performed an extensive analysis of nonce truncation-based leakages in ECDSA signatures, revealing vulnerable implementations in wolfSSL and OpenSSL. We exploit our findings in two attack case studies: one against curve secp160r1 in wolfSSL using *StumbleStepping* and one against curve brainpoolp224r1 in OpenSSL using our single-stepping primitive.

ACKNOWLEDGEMENTS

The authors thank Intel for providing them with access to a TDX enabled machine. Additionally, we thank Anja Rabich for fruitful discussions as well as Anna Pätschke and Jan Wichelmann for proofreading and valuable feedback. This work was supported by the BMBF projects SASVI and AnoMed as well as by Deutsche Forschungsgemeinschaft (DFG) under grant 439797619 (HaSPro).

REFERENCES

[1] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. 2023. Intel Trust Domain Extensions (TDX) Security Review. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf. Accessed on 07.10.2023.

[2] Martin R. Albrecht and Nadia Heninger. 2021. On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem. In *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12696)*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer, 528–558. https://doi.org/10.1007/978-3-030-77870-5_19

[3] AMD. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. 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.

[4] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 225–242. https://doi.org/10.1145/3372297.3417268

[5] ARM. 2023. Introducing Arm Confidential Compute Architecture. https://developer.arm.com/documentation/den0125/latest. Revision 0300-01.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[7] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3 (2015), 8:1–8:26. https://doi.org/10.1145/2799647

[8] Daniel Bleichenbacher. 2000. On the generation of one-time keys in DL signature schemes. In *Presentation at IEEE P1363 working group meeting*. 81.

[9] Robert Buhren, Felicitas Hetzelt, and Niklas Pirnay. 2018. On the Detectability of Control Flow Using Memory Access Patterns. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution* (Toronto, Canada) *(SysTEX '18)*. Association for Computing Machinery, New York, NY, USA, 48–53. https://doi.org/10.1145/3268935.3268941

[10] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 4:1–4:6. https://doi.org/10.1145/3152701.3152706

[11] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 178–195. https://doi.org/10.1145/3243734.3243822

[12] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. 2023. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/constable

[13] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices* 53, 2 (2018), 693–707.

[14] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, Cristiano Giuffrida and Angelos Stavrou (Eds.). ACM, 2:1–2:6. https://doi.org/10.1145/3065913.3065915

[15] Nick Howgrave-Graham and Nigel P. Smart. 2001. Lattice Attacks on Digital Signature Schemes. *Des. Codes Cryptogr.* 23, 3 (2001), 283–290.

[16] IBM. 2022. Introducing IBM Secure Execution for Linux 1.3.0. https://www.ibm.com/docs/en/linuxonibm/pdf/l130se03.pdf. Revision SC34-7721-03.

[17] Intel. 2021. Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification. Revision 348549-001.

[18] Intel. 2022. Intel Architecture Memory Encryption Technologies. Revision 336907-004US.

[19] Intel. 2023. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1 to 4. Revision 325462-080.

[20] Intel. 2023. Intel TDX Documentation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/tdx-security-research-and-assurance.html. Accessed on 30.11.2023.

[21] Intel. 2023. Intel Trust Domain Extensions. https://cdrdv2.intel.com/v1/dl/getContent/690419. Accessed on 07.10.2023.

[22] Intel. 2023. MKTME Side Channel Impact on Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mktme-side-channel-impact-on-intel-tdx.html. Accessed on 07.10.2023.

[23] Intel. 2024. Intel TDX Module - Code for Single-Step Detection and Single-Step Prevention. https://github.com/intel/tdx-module/blob/tdx_1.5/src/td_transitions/td_exit_stepping.. Accessed on 18.04.2024.

[24] Intel. 2024. Software Security Guidance - Best Practices. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/best-practices.html. Accessed on 28.08.2024.

[25] David Kaplan. 2017. Protecting VM Register state with SEV-ES. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf.

[26] David Kaplan, Jeremy Powell, and Wolle. 2021. AMD Memory Encryption. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf.

[27] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 557–574. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[28] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 337–351. https://doi.org/10.1109/SP46214.2022.9833768

[29] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1257–1272. https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan

[30] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 717–732. https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan

[31] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 355–371. https://doi.org/10.1109/SP40001.2021.00063

[32] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10529)*, Wieland Fischer and Naofumi Homma (Eds.). Springer, 69–90. https://doi.org/10.1007/978-3-319-66787-4_4

[33] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 469–486. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat

[34] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. 2020. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2057–2073. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm

[35] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*, Angelos Stavrou and Konrad Rieck (Eds.). ACM, 1:1–1:6. https://doi.org/10.1145/3193111.3193112

[36] Stephan Mueller and Marek Vasut. 2021. Intel Trust Domain CPU Architectural Extensions. https://www.kernel.org/doc/html/latest/crypto/index.html. Revision 343754-002.

[37] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1466–1482. https://doi.org/10.1109/SP40000.2020.00057

[38] National Institute of Standards and Technology. 2023. FIPS 186-5 - Digital Signature Standard (DSS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf.

[39] Phong Q. Nguyen and Igor E. Shparlinski. 2003. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.* 30, 2 (2003), 201–217. https://doi.org/10.1023/A:1025436905711

[40] OpenSSL. 2024. OpenSSL Security Policy. https://www.openssl.org/policies/general/security-policy.html. Accessed on 18.04.2024.

[41] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1852–1867. https://doi.org/10.1109/SP40001.2021.00020

[42] Keegan Ryan. 2019. Return of the Hidden Number Problem. A Widespread and Novel Key Extraction Attack on ECDSA and DSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 1 (2019), 146–168. https://doi.org/10.13154/TCHES.V2019.I1.146-168

[43] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WESEE: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *to appear at 45th IEEE Symposium on Security and Privacy*.

[44] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. HECKLER: Breaking Confidential VMs with Malicious Interrupts. In *to appear at 33rd USENIX Security Symposium (USENIX Security 24)*.

[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 753–768. https://doi.org/10.1145/3319535.3354252

[46] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. 2021. Util: : Lookup: Exploiting Key Decoding in Cryptographic Libraries. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2456–2473. https://doi.org/10.1145/3460120.3484783

[47] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. 2023. TeeJam: Sub-Cache-Line Leakages Strike Back. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 1 (Dec. 2023), 457–500. https://doi.org/10.46586/tches.v2024.i1.457-500

[48] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2020. MicroScope: Enabling Microarchitectural Replay Attacks. *IEEE Micro* 40, 3 (2020), 91–98. https://doi.org/10.1109/MM.2020.2986204

[49] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[50] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[51] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2023. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13959)*, Daniel Gruss, Federico Maggi, Mathias Fischer, and Michele Carminati (Eds.). Springer, 46–66. https://doi.org/10.1007/978-3-031-35504-2_3

[52] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. 2020. Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1767–1784. https://www.usenix.org/conference/usenixsecurity20/presentation/weiser

[53] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 161–173. https://doi.org/10.1145/3274694.3274741

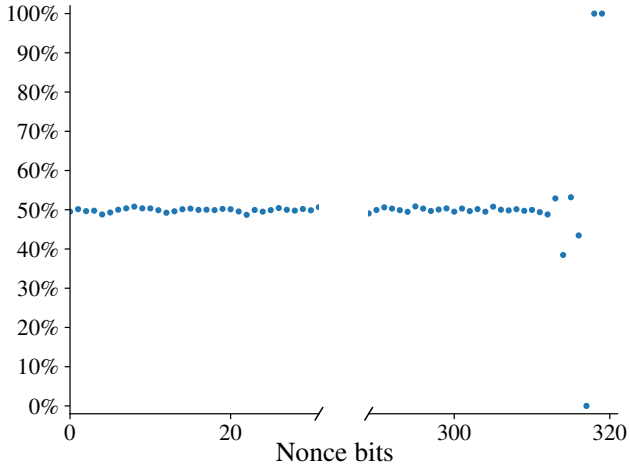
[54] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. 2022. Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 2915–2929. https://doi.org/10.1145/3548606.3560654

[55] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1483–1496. https://doi.org/10.1109/SP40000.2020.00080

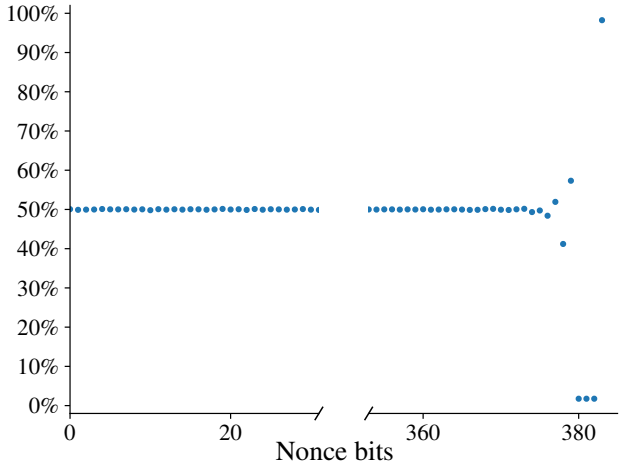
[56] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. 2023. SEV-Step A Single-Stepping Framework for AMD-SEV. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 1 (Dec. 2023), 180–206.

https://doi.org/10.46586/tches.v2024.i1.180-206
[57] wolfSSL. 2023. wolfSSL Manual. https://www.wolfssl.com/documentation/manuals/wolfssl/chapter02.html. Accessed on 30.11.2023.
[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 640–656. https://doi.org/10.1109/SP.2015.45
[59] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *33rd USENIX Security Symposium (USENIX Security 24)*.
[60] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 7267–7284. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-ruiyi

A ECDSA LEAKAGE ANALYSIS



(a) Distribution of the nonce bits for curve brainpoolp320r1 in wolfSSL given the event $W_1 = 3$. The most significant bits are 0b110 in all cases.



(b) Distribution of the nonce bits for curve brainpoolp384r1 in wolfSSL given the event $W_1 = 2$. The most significant bits are 0b1000 in 98% of all cases.

Figure 10: Distribution of nonce bits for different Brainpool curves. Events are specified in the subcaptions. The y-axis shows the percentage of nonces with a 1 in the corresponding bit position.

Table 3: Maximum obtainable leakage in terms of mutual information (MI) and fully leaked bits (FB) for different curves in wolfSSL und OpenSSL. This table complements Table 2. Data collection described in Section 6.1.

wolfSSL			OpenSSL		
Event	MI/FB	Pr[A=a]	Event	MI/FB	Pr[A=a]
(W_1, W_2)	[bit / bit]		(O_1, O_2)	[bit / bit]	
bp160r1					
-	-	-	(0, *)	0.1 / 0	0.81
			(1, 0)	1.3 / 1	0.11
			(1, 1)	1 / 1	0.08
bp192r1					
-	-	-	(0, *)	0.4 / 0	0.64
			(1, 0)	0.7 / 0	0.13
			(1, 1)	<0.1 / 0	0.23
bp256r1					
(1, 1)	0.2 / 0	0.24	(0, *)	0.4 / 0	0.85
(1, 2)	0.6 / 0	0.61	(1, *)	1.1 / 0	0.15
(2, *)	1.1 / 0	0.15			
bp224r1					
(1, 1)	<0.1 / 0	0.35	(0, *)	0.1 / 0	0.92
(1, 2)	0.3 / 0	0.56	(1, 0)	7 / 6	$1.6 \cdot 10^{-4}$
bp320r1					
(1, 1)	<0.1 / 0	0.16	(0, *)	0.3 / 0	0.54
(1, 2)	0.6 / 0	0.38	(1, 0)	0.1 / 0	0.28
(2, 3)	<0.1 / 0	0.38	(2, *)	3 / 3	$1.7 \cdot 10^{-3}$
(2, 4)	<0.1 / 0	0.8			
bp384r1					
(1, 1)	0.6 / 0	0.25	(0, *)	0.7 / 0	0.95
(1, 2)	0.8 / 0	0.70	(1, *)	3.5 / 0	0.05
secp128r1					
(1, 1)	<0.1 / 0	0.30	(0, *)	<0.1 / 0	0.77
(1, 2)	0.3 / 0	0.47	(1, *)	1.3 / 1	0.23
(2, *)	1.3 / 1	0.23			
secp192r1					
(1, *)	0.2 / 0	0.5	(0, *)	0.2 / 0	0.5
(2, *)	0.2 / 0	0.5	(1, *)	0.2 / 0	0.5