# TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters

Fabian Rauscher
Graz University of Technology
Graz, Austria
fabian.rauscher@tugraz.at

Hannes Weissteiner
Graz University of Technology
Graz, Austria
hannes.weissteiner@tugraz.at

Daniel Gruss
Graz University of Technology
Graz, Austria
daniel.gruss@tugraz.at

## Abstract

Trusted execution environments (TEEs) protect applications running inside of them from untrusted host systems. The host can not access or modify the memory of applications protected by a TEE. Intel TDX is a recently introduced TEE that allows for the execution of arbitrary code, including entire operating systems, inside a protected environment. Prior work has attacked AMD SEV-SNP, AMD's counterpart to Intel TDX, using performance counters, by leaking sensitive information through them, e.g., which branches are taken. Intel TDX is thought not to be affected by such attacks, as it disables performance counters when entering the protected guests (TDs) to mitigate these attacks.

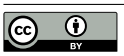
In this paper, we bypass the protections of Intel TDX, allowing us to not only recover secrets, such as private keys, but also expand on what is thought possible by leaking arbitrary memory using performance counters. We analyze available performance counters on the recent Intel Emerald Rapids microarchitecture, finding 8 that track events for the whole physical core and not just the current logical core. We use this to bypass the Intel TDX mitigation against performance counter-based attacks by monitoring TDs from the sibling logical core. One of these counters tracks uOPs executed, allowing an attacker to gain valuable insight into victim TDs. Using this information, we recover an RSA-2048 private key from a TD running MbedTLS with an average edit distance of only 0.92 bits. Furthermore, this performance counter includes speculatively executed uOPs, enabling the use of a large number of previously unusable Spectre compare gadgets in Spectre attacks. With this discovery, we leak TD memory at a rate of 52.6 bit/s and break KASLR in less than 2 s. Finally, we break the recently introduced inter-keystroke timing defense of OpenSSH, allowing us to detect real keystrokes with an $F_1$ score of 99.6 %.

## CCS Concepts

• **Security and privacy → Trusted computing**; **Side-channel analysis and countermeasures**.

## Keywords

Intel TDX, Side Channel, Performance Counter, TEE




*ASIA CCS '26, Bangalore, India*

ACM ISBN 979-8-4007-2356-8/2026/06
https://doi.org/10.1145/3779208.3785379

**ACM Reference Format:**
Fabian Rauscher, Hannes Weissteiner, and Daniel Gruss. 2026. TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 01–05, 2026, Bangalore, India.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779208.3785379

## 1 Introduction

To reduce hardware and maintenance costs, an increasing number of companies are moving their infrastructure to the cloud. This change comes with additional confidentiality concerns, as sensitive data, e.g., customer data, company secrets, and cryptographic keys, are now stored and processed on hardware owned and operated by a third party. With traditional virtualization technologies, the cloud provider has full access to all data in the virtual machine (VM). Thus, a data breach or malicious insider at the cloud provider can have severe consequences for their customers. To mitigate these concerns, trusted execution environments (TEEs) have been introduced by hardware vendors. TEEs provide isolated environments that protect the confidentiality and integrity of applications and data in the TEE from the host system by protecting the memory and register state of the TEE. Traditional TEEs, e.g., Intel SGX or ARM TrustZone, require applications to be written specifically for the specific TEE implementation [6, 16], limiting their applicability and portability. Prior work demonstrated that generic programs can be ported to Intel SGX using a library OS approach [74], but resulting programs are still limited due to the design details of SGX. Recently, vendors introduced TEEs that allow entire VMs to run inside of them, such as AMD SEV-SNP [2] and Intel TDX [32]. These confidential VMs (CVMs) allow users to run unmodified applications on general-purpose operating systems in TEEs.

The TEE threat model includes a malicious machine owner, with control over the host operating system and physical access to the machine [2, 16, 32]. As such, the attack surface of this threat model is significantly larger than the traditional unprivileged user-space attacker model. Many prior works have demonstrated powerful attacks on SGX. Early works demonstrated that the fact that the host and the enclave share the same hardware allows for cache side-channel attacks [8, 26, 54, 67, 79, 80]. Many attacks on TEEs rely on functionality that is only available as a malicious hypervisor, e.g., page faults [82, 85], single-stepping [64, 77, 82, 84], performance counters [22, 51], or firmware modifications [18].

Recently, multiple works were released on performance counter based attacks on AMD SEV-SNP, as SEV did not protect against these type of attacks. Lou et al. [51] were the first to demonstrate a website fingerprinting attack on AMD SEV-SNP using performance counters. Gast et al. [22] demonstrated more fine-grained performance-counter-based attacks on AMD SEV-SNP, including

the recovery of RSA keys, TOTP secrets, and breaking the HQC post-quantum signature scheme. Both works mention that Intel TDX is not affected by their attacks, as Intel TDX disables performance counters when entering the TEE. Weissteiner et al. [81] proposed a method to decorrelate performance counter values between the host and the TEE, mitigating fine-grained performance counter leakage from TEEs. However, they only mitigate leakage from performance counters on the same logical core, without considering potential leakage across hyperthreads.

In this paper, we break Intel's mitigation against performance counter attacks and show how it is still possible to leak fine-grained information through them, allowing us to leak RSA keys and perform Spectre attacks on compare gadgets regardless of the operations they perform. We exploit performance counters that monitor the whole physical core to monitor TDX CVMs, called trust domains (TDs), from a sibling logical core. This is possible as Intel TDX only disables performance counters on the logical core the TD is running on, but not the entire physical core. We analyze available performance counters on a recent Intel processor and find 8 that count events for the whole physical core. One of these performance counters, `UOPS_EXECUTED.CORE`, monitors the precise micro OP (uOP) throughput of the TD, allowing us to gain detailed information on the TD's execution flow. We leverage `UOPS_EXECUTED.CORE` to fingerprint code passages inside the TD, identifying secret-dependent code execution. Using this information, we recover a full RSA-2048 private key from a TD running MbedTLS by only monitoring 400 encryptions happening in the TD. The recovered key has an average Levenshtein distance of only 0.92 bits, meaning on average *less than one bit* has to be changed in the recovered key to get the real key, showing the high accuracy of our attack. Additionally, `UOPS_EXECUTED.CORE` also counts speculatively executed uOPs. Hence, our attack is a new way to leak information for speculative execution attacks on Intel TDX using `UOPS_EXECUTED.CORE` that is universal across Spectre transmission channels.

Our discovery introduces an asymmetry in favor of the attacker when considering the four gadget types in Spectre attacks [10], namely prefetch, compare, index, and execute gadgets. Index gadgets have been explored the most so far, as they are easy to find and use [41, 72]. However, our attack provides a substantial benefit for all other gadget types: More specifically, the attacker can infer the outcome directly from the `UOPS_EXECUTED.CORE` performance counter. Hence, there is no need for any other specific transmission channel, such as a cache side channel, to leak the secret-dependent outcome of the gadget. Thus, we can use previously unusable gadgets for Spectre attacks. We demonstrate this by abusing the Linux kernel's implementation of the `memchr` function to leak arbitrary memory from a TD at a rate of 52.6 bit/s with an error rate of only 0.6 %, breaking the confidentiality guarantees of Intel TDX. We also demonstrate a KASLR break using our generic Spectre attack on TDs, leaking the KASLR offset in less than 2 s. Finally, we discover that the recently introduced inter-keystroke timing defense introduced to OpenSSH [1] is insufficient for a CVM scenario. By performing precise timing measurements and taking advantage of the high amount of control an attacker has in the traditional TEE threat model, we are able to detect real keystrokes with an $F_1$ score of 99.6 % and a temporal standard deviation of only 5.64 ms, despite the presence of a large number of fake keystrokes generated by the mitigation. Furthermore, we show that, even if this timing side channel is mitigated, the real keystrokes can still be determined using `UOPS_EXECUTED.CORE`.

In summary, our work makes the following contributions:

- We systematically analyze available performance counters on a recent Intel CPU for cross-core leakage, identifying 8 counters that count events for the whole physical core and 1 counter that is a particularly high threat to Intel TDX.
- We mount an attack on MbedTLS `3.5.2` running inside of a TD using this dangerous performance counter to bypass Intel's mitigation to recover an RSA-2048 key with an average Levenshtein distance from the real key of **only** 0.92 bits.
- We demonstrate Spectre attacks exploiting the inherent asymmetry of the `UOPS_EXECUTED.CORE` performance counter channel, *i.e.*, any Spectre gadget encoding information can be used; and use it to leak arbitrary memory at a rate of 52.6 bit/s and break KASLR in <2 s.
- We demonstrate that OpenSSH's recent inter-keystroke timing defense is insufficient for CVMs, allowing us to distinguish between real and fake keystrokes with an $F_1$ score of 99.6 %, thus, re-enabling inter-keystroke timing attacks.

*Outline.* We provide background in Section 2. In Section 3, we analyze available performance counters for cross-core leakage. In Section 4, we recover a full RSA-2048 private key from a TD running MbedTLS, using performance-counter leakage. In Section 5, we demonstrate Spectre attacks using the new `UOPS_EXECUTED.CORE` channel for data leakage from speculative execution on Intel TDX. In Section 6, we break openSSH's recent inter-keystroke timing defense. We discuss possible mitigations and related work in Section 7. We conclude in Section 8.

*Responsible Disclosure.* We responsibly disclosed our findings to Intel on August 7, 2025 and to the OpenSSH team on August 25, 2025. Intel recommends developers to follow their security best practices for side-channel resistance. The OpenSSH team does not consider confidential virtual machines as part of their threat model for the inter-keystroke timing attack mitigation and will therefore not mitigate our attack.

## 2 Background

In this section, we first discuss Trusted Execution Environments (TEEs) and, in more detail, Intel TDX. We also provide a brief overview of hardware performance counters (on Intel processors) and how they can be used both for malicious and benign purposes.

### 2.1 Trusted Execution Environments

Trusted Execution Environments (TEEs) are an emerging technology that processor vendors introduce to offer increased confidentiality and integrity guarantees compared to what traditional user-kernel isolation can provide [2, 5, 34, 35]. The first generation of TEEs mainly protected specific application components, e.g., Intel Software Guard Extensions (SGX) [34] and ARM TrustZone [6], focused on a scenario where a trusted application must be protected from a malicious user or compromised system. Since this design limits the potential usage scenarios to small applications handling small amounts of highly sensitive data e.g., fingerprints

or cryptographic material, or for digital rights management (DRM), the scientific community quickly realized the need for TEEs that allow users to run any system and application as a TEE [74]. As a consequence, a new type of TEEs emerged that focuses on the scenario where an entire virtual machine must be protected from a malicious or compromised host. Since similar confidentiality guarantees (but not necessarily integrity guarantees [83]) are provided, these virtual machines are also called confidential virtual machines (CVMs). Both AMD and Intel realized their respective CVM implementation: AMD Secure Encrypted Virtualization (SEV) [3] and Intel Trust Domain Extensions (TDX) [32].

## 2.2 Intel TDX

Intel calls CVMs running on Intel TDX trust domains (TDs) [35]. The open-source TDX module is designed to handle the encryption and management of guest memory and saved guest state within the trust domain virtual processor state area (TDVPS). The TDX module operates in SEAM root execution mode, which is protected from the host, and interfaces between the TEE and the host. Internally, the TDX module uses existing Intel Virtual Machine Extensions (VMX) for virtualization. To protect against malicious modifications, the TDX module is signed by Intel and can only be loaded if the processor can verify the signature. TDX splits the guest's physical memory into a shared section, accessible by both host and guest, and a private, encrypted section, only accessible to the guest and the TDX module. This design allows for fast communication with the host via the shared section, while still protecting the guest's private memory. The TDX module handles page table management for the private guest-physical memory, while the host manages the shared memory's page tables. A dedicated bit in the guest physical address allows the guest to distinguish between shared and private memory, avoiding accidental interactions with shared memory. Additionally, shared memory is not executable to mitigate certain bugs and attack vectors. To encrypt the memory of TDs, TDX employs Intel's Total Memory Encryption - Multi Key (TME-MK). TDX splits TME-MK's key ID (HKID) range into a public and a private part, reserving the private range for the TDX module and TDs. Each 64-bit region is validated against a cryptographic MAC on every memory access to protect against memory corruption, e.g., due to physical interference. However, the CPU is still shared between all TDX guests and the host, allowing for potential information leakage through shared hardware components.

Information leakage from TEEs is a major concern. While first side-channel attacks have been demonstrated, in the context of Intel SGX [8, 21, 26, 30, 43, 54, 67, 79, 80] and Arm TrustZone [48, 66], transient-execution attacks have been found to be an even more powerful information leakage primitive [13, 41, 57, 72, 75, 76, 78]. Thereby, the TEE threat model often allows for a stronger attacker, that can precisely control the execution of the TEE, e.g., through single-stepping [54, 64, 77, 84, 85]. Alternatively, attackers can also use power side channels [49] and fault attacks [14, 56, 60, 61].

Unlike AMD's counterpart to TDX called AMD SEV-SNP, Intel TDX employs active mitigations against some controlled channels, such as single-stepping through interrupts and zero-stepping through page faults, and performance counter-based attacks, that are built directly into the TDX module. The single-stepping defense detects an attack through the number of instructions executed and introduces noise in case an attack is detected. The defense against page fault-based zero-stepping simply checks whether the instruction pointer changes between two consecutive page faults in the guest physical to host physical translations. To mitigate performance counter-based attacks, the TDX module disables performance counters when entering the TDX module through the `GLOBAL_PERF_CNT` MSR and context switches performance counter MSRs in case the TD is allowed to use them for themselves.

## 2.3 Hardware Performance Counters

Modern processors are highly complex and provide numerous ways to debug, profile, and monitor the processor's execution of software. Hardware performance counters are particularly useful when investigating how a piece of code exercises the CPU hardware. Performance counters are provided to the user via registers that count certain hardware events, e.g., cache hits and misses in a specific cache level. Developers can use this information to debug performance bottlenecks and optimize the software [59], or to monitor software execution and detect anomalies [12, 15, 17, 44, 45, 88]. Performance counters are typically only reachable from kernel space, e.g., via Model Specific Registers (MSRs). The potential information leakage through performance counters was already known when Intel SGX was introduced. Intel excluded any SGX activity from all performance counters [36]. AMD did not exclude SEV activity from performance counters until recently in response to performance-counter-based attacks on AMD SEV-SNP [22, 51]. Gast et al. [22] demonstrated several end-to-end attacks, e.g., the recovery of RSA keys from a SEV-SNP CVM. Consequently, performance-counter-based attacks are currently considered mitigated on Intel SGX, Intel TDX, and recent versions of AMD SEV-SNP. However, some performance counters only count global information, which so far has not been considered relevant in terms of leakage from TEEs.

# 3 Performance Counter Analysis

In this section, we analyze performance counters for the core domain and determine possible data leakage through them. We analyze the available core performance counters on our Intel Xeon Silver 4514Y Emerald Rapids CPU and determine which performance counters allow for leakage across logical cores. From the found performance counters, we discuss one promising counter for attacks, `UOPS_EXECUTED.CORE`.

Unlike with AMD SEV-SNP, most performance counters cannot be used to monitor guest activity [34, 35]. This is due to the TDX module context switching the `GLOBAL_PERF_CNT` MSR. The `GLOBAL_PERF_CNT` MSR contains a bit for each hardware performance counter through which they can be enabled or disabled. When entering the TDX module, the `GLOBAL_PERF_CNT` is set to disable all performance counters. If a TD is allowed to use performance counters, a `GLOBAL_PERF_CNT` value, as well as the values for the other performance counter MSRs, are maintained for each virtual core of the TD and loaded by the TDX module. As these values are stored inside of the encrypted TD state, the host can neither read nor manipulate these values. Therefore, all performance counters that are set up by the host on the logical core on which the TD is run on do not count while executing the TD.

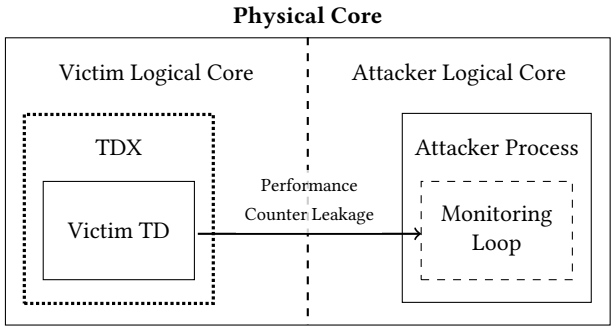



**Figure 1: Attack overview. TDX protects from performance counter leakage on the same logical core. However, the victim TD affects performance counters that count for the whole physical core, which can be monitored by an attacker on the sibling logical core.**

While the context switching of GLOBAL_PERF_CNT mitigates attacks on the same logical core, it does not prevent a malicious host from tracking performance counters that count events for the whole physical core through the use of the target's sibling logical core. In general, Intel differentiates between 3 types of performance counters: core, uncore, and offcore. Core performance counters track events occurring inside of the CPU core, e.g., instructions executed and cycles stalled. Uncore counters track events that are outside of the core, in the uncore (including the LLC and memory controller), e.g., DRAM interactions and data requests to MMIO, and are for the whole CPU. Finally, offcore counters track per-core events related to interactions between the core and the uncore, e.g., prefetches to the L2. In this work, we focus on core domain performance counters, as information that can be gained from uncore and offcore performance counters is limited, e.g., memory requests that go to the L3. Core performance counters, on the other hand, provide very specific information regarding the execution flow of a core, making them interesting tools for leaking data from a TD.

As core performance counters provide much more fine-grained information on the current execution flow, e.g., branches taken, instructions retired, and mispredictions, we want to analyze them to determine whether there are any that can be used to attack TDs, despite the existing defenses against such attacks. Performance counters of particular interest are the ones that count for the whole physical core, as they can be set up on one attacker-controlled logical core, while the sibling logical core is inside a TD, as shown in Figure 1. Intel's documentation regarding what exactly some performance counters cover can be very vague. For example, while some of the documented events explicitly mention whether they count for the current logical core or the whole physical core, most do not [33]. Despite this, a majority of the events only the current logical core that monitors them. This includes events such as BR_INST_RETIRED.*, INST_RETIRED.ANY_P, and UOPS_DISPATCHED.PORT_*. We found that performance counters only target the whole physical core when it is explicitly mentioned in the description of the counter, when it is part of the performance counter name, or when it targets offcore events.

**Table 1: List of performance core counters monitoring the whole core**

| Performance Counter Name |
|---|
| CORE_SNOOP_RESPONSE.* |
| CPU_CLK_UNHALTED.REF_DISTRIBUTED |
| CPU_CLK_UNHALTED.REF_TSC* |
| IDQ_UOPS_NOT_DELIVERED.CORE |
| IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE |
| OFFCORE_REQUESTS.* |
| OFFCORE_REQUESTS_OUTSTANDING.* |
| UOPS_EXECUTED.CORE |

```
size_t last = rdmsr(IA32_PMC0), i = 0:
while (!atomic_read(&done) {
  size_t cur = rdmsr(IA32_PMC0);
  measurements[i++] = cur-last;
  last = cur;
}
```

**Listing 1: Code of the performance counter measurement hot loop.**

We analyzed all the available performance counter events published by Intel for our Emerald Rapids CPU Intel Xeon Silver 4514Y [33] and listed all performance counters that we found that target the whole core in Table 1. From the found performance counters, UOPS_EXECUTED.CORE is a very promising tool for attacks, as the uOP throughput of a core highly depends on the instructions executed. Assuming secret-dependent branches in an application, this can leak sensitive information. Additionally, we show that UOPS_EXECUTED.CORE also counts speculatively executed uOPs that are never committed. This can introduce additional noise to some attacks, but enables the use of UOPS_EXECUTED.CORE to broaden the scope of exploitable Spectre gadgets.

UOPS_EXECUTED.CORE counts uOPs executed by the whole core. When the CPU executes instructions, they are first translated into simpler operations, so-called uOPs, which are in turn executed by different execution units of the core. The number of uOPs an instruction is translated to depends on the instructions and the system it is running on. This information is similar to the information leaked by single-stepping, which is actively mitigated by the TDX module [35], making it a relevant attack vector.

We now confirm whether UOPS_EXECUTED.CORE actually counts the uOPs executed inside of a TD, and that we do not just measure contention or other side effects between the two logical cores. We let the TD execute either IMUL or ADD in a loop and continuously measure the uOPs executed on the sibling logical core through UOPS_EXECUTED.CORE and UOPS_EXECUTED.THREAD. The UOPS_EXECUTED.THREAD performance counter only accounts for the current logical core, in contrast to UOPS_EXECUTED.CORE, which targets the whole physical core. To minimize any unnecessary overhead, we do not track time and only record the value of the performance counters (see Listing 1). As the measurement loop consists

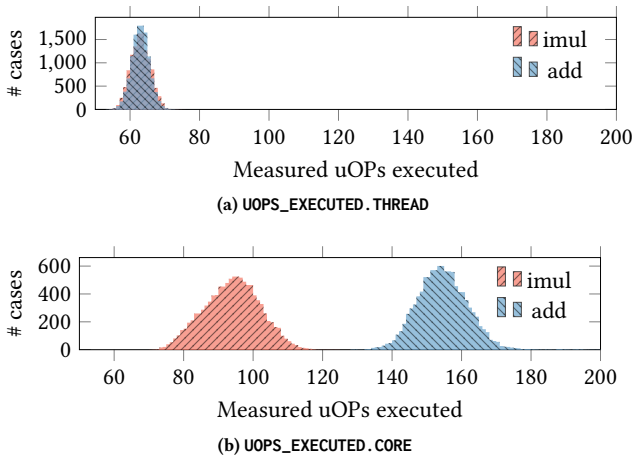



**Figure 2: Measured uOPs executed using `UOPS_EXECUTED.CORE` (Figure 2b) and `UOPS_EXECUTED.THREAD` (Figure 2a) while the other logical core is inside of a TD and executing either `IMUL` or `ADD` instructions in a loop. With `UOPS_EXECUTED.CORE`, the two instructions can be clearly differentiated from each other. With `UOPS_EXECUTED.THREAD`, the two instructions are indistinguishable, showing that the differences observed by `UOPS_EXECUTED.CORE` are from the direct influence of the TD on the counter and not from other side effects, e.g., contention.**

```
1 test rax, rax;
2 je 1f;
3 serialize;
4 .rept 32;
5 add rbx, 1;
6 .endr;
7 serialize;
8 1: nop;
```

**Listing 2: Code to determine whether `UOPS_EXECUTED.CORE` counts specultively executed uOPs.**

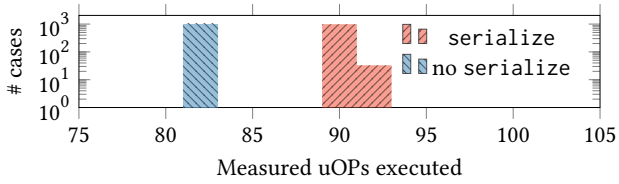



**Figure 3: Measured uOPs executed using `UOPS_EXECUTED.CORE` of the code listed in Listing 2 when the conditional branch always misspeculates, with the `serialize` in line 2 and without it. As both cases commit the same instructions, the difference in measured uOPs executed (without the serialize higher due to more instructions being speculatively executed) confirms that `UOPS_EXECUTED.CORE` also counts only speculatively executed uOPs.**

of a fixed number of instructions that are executed, the uOPs executed should also stay constant with slight variations, e.g., due to misspeculation of the measurement loop. Therefore, if `IMUL` and `ADD` can be clearly distinguished using `UOPS_EXECUTED.CORE`, but not with `UOPS_EXECUTED.THREAD`, we are able to track uOPs executed inside the TD with `UOPS_EXECUTED.CORE`, and we are not observing a different side effect. In case `IMUL` and `ADD` can also be distinguished from each other with just `UOPS_EXECUTED.THREAD`, then we are likely observing a different side effect, e.g., the code in the TD affects the measurement codes branch prediction.

The results of these measurements are provided in Figure 2. With `UOPS_EXECUTED.THREAD` (Figure 2a), we measure ~62 uOPs for the `IMUL` and `ADD` loops, making them indistinguishable from each other. The uOPs executed do not reach 0, in this case, as the measurement still includes the uOPs executed by the measurement code itself. With `UOPS_EXECUTED.CORE` (Figure 2b), we measure ~90 uOPs for `IMUL` and ~150 uOPs for `ADD`, making them clearly distinguishable, showing that we are able to infer information about code executed inside of a TD. Furthermore, these measurements show that uOP throughput can also leak information on the instructions executed.

Intel defines `UOPS_EXECUTED.CORE` as counting all uOPs executed, including speculatively executed instructions [33]. To confirm whether `UOPS_EXECUTED.CORE` actually includes speculatively executed uOPs, we run a short test snippet shown in Listing 2 on our Intel Xeon Silver 4514Y. For each measurement, we first train the branch in line 2 not to be taken and then use `UOPS_EXECUTED.CORE` to determine the uOPs executed when the branch is taken. Due to the training, the branch will misspeculate and execute `serialize` in line 3 only speculatively, which in turn stops the speculation. We then repeat the experiment without the `serialize` in line 3. Without `serialize`, the additions in line 5 should be executed speculatively. While with and without the `serialize` instruction, the exact same number of uOPs are committed, there are more uOPs speculatively executed without `serialize`. Therefore, if the uOPS executed in the two cases differ, `UOPS_EXECUTED.CORE` includes speculatively executed uOPs, even when they are never committed.

The results of our measurements are shown in Figure 3. Without `serialize`, the uOPs executed are 90.1 ($n = 1024$, $\sigma_{\bar{x}} = 0.01$). With `serialize`, the uOPs executed are 82 ($n = 1024$, $\sigma_{\bar{x}} = 0$). This confirms that `UOPS_EXECUTED.CORE` includes speculatively executed uOPS.

## 4 RSA Key Recovery

In this section, we leverage `UOPS_EXECUTED.CORE` to perform a full RSA-2048 private key recovery from a TD running MbedTLS 3.5.2 [4]. The MbedTLS RSA implementation is not constant-time and uses a windowed square-and-multiply approach to perform RSA encryption, enabling side-channel attacks. While both the square- and the multiply-operations by themselves are implemented in constant time, *i.e.*, they always take the same amount of time irrespective of the ciphertext, for each bit in the exponent (private key), the encryption either performs a square- or a square- and a multiply-operation. Therefore, if we can determine which operations were performed for each key bit, we can recover the private key. Similar to prior work [22, 24, 49, 71], we configure MbedTLS

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

**Listing 3: Single loop iteration of the MbetTLS RSA implementation [4]. When the exponent bit is 0 (square), lines 2 to 7 are executed. When the exponent bit is 1 (square and multiply), lines 9 to 26 are executed.**

to use a window size of 1, as a working attack on a window size of 1 can be extended to arbitrary window lengths [50].

## 4.1 Threat Model

We follow the typical TEE threat model of a compromised host [56, 64, 69, 75, 77, 82, 84, 87]. Our host system contains a Intel Xeon Silver 4514Y running TDX-enabled Ubuntu 24.04 [11] with TDX module 1.5.16, the most recent version at the time of writing [37]. Our guests run Ubuntu 24.04, which was created according to the TDX guide published by Canonical and has one virtual core. This setup is in line with previous work [64, 82, 84] and the official TDX threat model of a compromised cloud provider, published by Intel [32]. We run standard MbedTLS 3.5.2 [4] and did not modify it in any form to perform our attack.

## 4.2 Overview

With square-and-multiply, for each bit in the exponent, the value to be encrypted is either squared and multiplied by the initial value if the bit is 1 or only squared if the bit is 0. The code for processing a single bit from the MbedTLS RSA implementation is shown in Listing 3. MbedTLS uses the same constant time multiplication function (`mpi_montmul`) for both the square and the multiply steps, making them indistinguishable from each other. The `mpi_select` is a constant-time conditional copy. While the `mpi_montmul` and

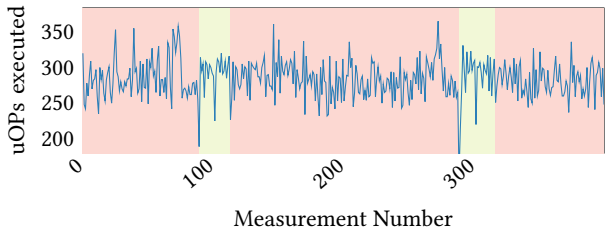



**Figure 4: Part of an RSA encryption using MbedTLS containing two executions of the `mpi_select` and `mpi_montmul` functions. `mpi_select` executions start at measurements ~100 and ~300 highlighted in green with executions of `mpi_montmul` highlighted in red.**

`mpi_select` functions have a relatively constant uOP throughput, the code in between them, where the exponent bits are checked, does not. Lines 2 to 7 perform the square operation when the exponent bit is 0, and lines 9 to 26 are only executed to perform square and multiply when the exponent bit is 1. By constantly monitoring uOPs executed through `UOPS_EXECUTED.CORE` on the sibling logical core of the victim, it is possible to determine when which operation is executed (`mpi_montmul`, `mpi_select`, or the code in between) and which code path in the processing loop was taken.

The trace of uOPs executed during two executions of `mpi_select` and `mpi_montmul` during a regular encryption running inside a TD is shown in Figure 4. To record the trace, we use the same measurement setup as described in Section 3 with a measurement point consisting of the difference of two performance counter reads. Executions of `mpi_select` are highlighted in green between ~100 and ~125 as well as between ~300 and ~325. Almost all of the other execution time is taken up by `mpi_montmul` highlighted in red. At the end of each multiplication, the uOP throughput spikes for a few measurements and then stabilizes, before finally having a 1 to 2 measurement drop (at ~100 and ~300). This short drop is the transition between `mpi_montmul` and `mpi_select`. When executing `mpi_select`, the throughput stabilizes with a short drop in the middle and at the very end when transitioning to the next `mpi_montmul` call. The measurements around the transitions from `mpi_montmul` to the next `mpi_select` hold the conditional code shown in Listing 3. With these measurements extracted, it is possible to determine which parts of the function were executed when, therefore, leaking the key.

As it is not possible for an attacker to know at which exact point in code a measurement point starts and stops, we require multiple traces for a full key recovery. To minimize implementation effort, we take advantage of a controlled channel in addition to our `UOPS_EXECUTED.CORE`. We use the Intel TDX feature to block and unblock TD pages [35] to force a VM exit between the multiplications, allowing us to only measure the target code (Listing 3) together with the `mpi_select` calls, similar to prior work [22, 82]. Intel TDX allows the host to block the access to private pages of TDs as an initial step for hypervisor management functions, such as merging or splitting pages. This functionality is required for the host to perform the TLB invalidation sequence before actual

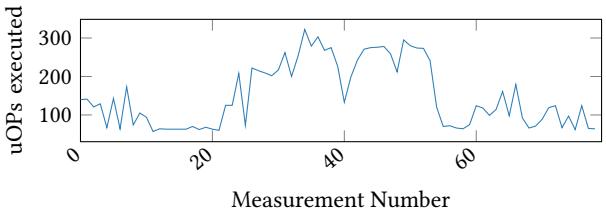



**Figure 5: Trace of a single `mpi_select` together with part of the code from Listing 3 recorded with the help of a controlled channel. The two plateaus with a drop in the middle are the `mpi_select` function, with the very short Listing 3 code being executed directly before it.**

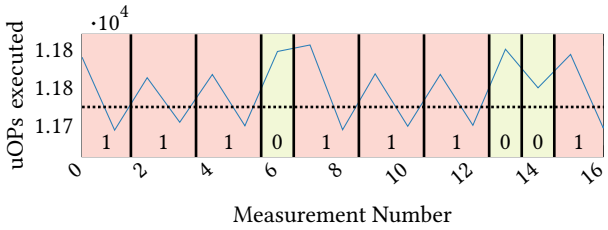



**Figure 6: Trace of uOPs executed. Each datapoint is the sum of uOPs measured of the code between two `mpi_montmul` calls of MbetTLS RSA averaged for 400 measurements. When a data point falls below the threshold (dotted line), the TD executed the short code between a square and a multiply, indicating a 1 (together with the previous measurement point, marked in red). Otherwise, the code executed is between two squares indicating a 0 (marked in green).**

changes to the TD mappings are made. While Intel actively tries to mitigate zero-stepping attacks that are done using this feature, as long as the instruction pointer makes progress between VM exits resulting from blocked pages, the mitigation does not intervene. During our testing, `mpi_montmul` and the MbedTLS RSA function were never on the same page, allowing us to use this controlled channel for synchronization. The guest physical addresses for these functions can be determined by the malicious host through page tracking proposed by Li et al. [46]. Despite this, we confirmed that our attack works even without the use of this controlled channel, by collecting full traces of encryptions, searching for the `mpi_select` executions through pattern matching, and recovering a significant part of the private key using this information. We did not implement the full key recovery using pattern matching, as the `mpi_select` execution is clearly visible (Figure 4). Therefore, implementing reliable pattern matching would only be an engineering challenge, providing no additional scientific value.

To set up the controlled channel, we block access to the page containing `mpi_montmul` and the page containing the rest of the code provided in Listing 3 which can be found by profiling the TDs physical memory. Whenever the TD tries to execute the regular RSA logic it will result in a VM exit, returning the control to the host. In this case we unblock the page and block the access to the `mpi_montmul` page. Whenever the TD gets to the next multiplication, we again receive a VM exit where we unblock the `mpi_montmul` page and block the access to the rest of the RSA implementation. Through this mechanism, we are able to precisely track when the code between two multiplications is executed. The trace of an `mpi_select` and the logic provided in Listing 3, *i.e.*, for the code executed between two `mpi_montmul` calls, recorded with the help of the controlled channel is shown in Figure 5. We use the same measurement approach as in Figure 4. The relatively constant parts of the trace at the beginning and end of the trace are part of the VM entry and VM exit, respectively. The two plateaus with a drop in between them are caused by the `mpi_select` function, which performs a copy operation and takes up most of the execution time between two `mpi_montmul` calls. This same pattern can also be observed in Figure 4. Our target code from Figure 5 is executed right before `mpi_select` and is most likely contained in 1 to 2 measurement points, due to its short length.

Figure 6 shows the average of 400 uOP traces collected from a small part of an MbedTLS RSA encryption with the `mpi_montmul` calls filtered out. Each measurement point in Figure 6 is the sum of all uOPs executed for a call to `mpi_select` and the code in between function calls shown in Listing 3. We collected these traces with the controlled channel by blocking access to the `mpi_montmul` page and the page containing the rest of the RSA implementation, always unblocking the one the TD wants to execute and blocking the other one. Due to this, we are able to synchronize our measurements with the victim without requiring a change in the MbedTLS source code. The lowest points in the trace (below the dotted line) are the executions of the code between line 17 and line 22, shown in Listing 3 (the code between a square and a multiply). This part of the code only contains a call to `mpi_select` and no other logic leading to the low number of uOPs executed. Such a low measurement is a clear indication that the TD just processed a 1, as this code part is only executed in this case. As processing a 1 results in two multiplications, we can group the current and previous point into a single operation. When a measurement of a high number of uOPs executed (above the threshold) is not followed by a low number of uOPs executed (below the threshold), this is an indication of two square operations after each other. Therefore, we can conclude that a 0 has been processed. We set the threshold at "$0.2 \cdot (max - min) + min$" where *min* is the lowest measurement point during the encryption and *max* is the highest. This resulted in the most stable results for our experiments.

## 4.3 Evaluation

We evaluated our attack on an Intel Xeon Silver 4514Y with MbedTLS 3.5.2 and Ubuntu 24.04 running on both the host and the guest. We followed the threat model outlined in Section 4.1. The victim TD is executing RSA-2048 encryptions using MbedTLS, while the host is monitoring the guest on a sibling logical core using `UOPS_EXECUTED.CORE`. Using 400 encryptions for each key extraction, we are able to recover the key with an average Levenshtein distance of 0.92 over 25 key extractions with a standard error of 0.37, meaning on average 0.92 bits of the recovered key need to

be changed to derive the correct key. We report the Levenshtein instead of the hamming distance, as the processing of a 1 has double the measurement points as the processing of a 0. Therefore, when a 1 is misclassified as a 0, a second 0 (the second part of the square and multiply operation) would automatically be detected, consequently shifting the rest of the recovered key by 1 bit. This shift would lead to a high hamming distance that does not properly reflect the correct number of key bits recovered. Additionally, 72 % of our attack runs were able to recover 100 % of the correct private key, while the rest only had a small number of bit errors.

Similarly to our work, Gast et al. [22] performed a single trace RSA key recovery on AMD SEV-SNP using branch-related performance counters on the same logical core. In contrast, our attack does not require single-stepping, which is actively mitigated on Intel TDX, but can be performed on a normally running guest. We use `UOPS_EXECUTED.CORE` for data leakage, which was not highlighted by them as a possible attack target. In contrast, they use branch-related counters, which do not account for the other logical core, making them unusable on TDX. Furthermore, we bypass the TDX mitigation against performance counter-based attacks, which, as Gast et al. [22] acknowledged, mitigates their attacks.

## 5 Any Gadget Spectre Attacks

In this section, we introduce Spectre attacks that have a much broader range of suitable gadgets. We leverage that the performance counter channel using `UOPS_EXECUTED.CORE` turns almost every secret dependent operation, e.g., a conditional branch, into a data leaking gadget when targeting a TD. We first explain how Spectre attacks using the `UOPS_EXECUTED.CORE` channel work and how they drastically increase the possible gadgets available. We then demonstrate a KASLR break by combining Spectre with the `UOPS_EXECUTED.CORE` performance counter channel and show how it can be used in Spectre attacks to leak arbitrary memory from a victim TD. For all attacks, we follow the threat model outlined in Section 4.1.

### 5.1 Overview

Traditional Spectre attacks rely on specific secret-dependent operations to leak information. Kocher et al. [41] showed that memory accesses are an effective means to encode arbitrary data into the cache state during speculative execution. The attacker can then infer the secret information through the cache state after the execution. This greatly limits the number of gadgets available to an attacker. Canella et al. [10] distinguished between 4 gadget types: prefetch, compare, index, and execute gadgets. Most prior works rely on index gadgets that perform two memory accesses: one to load the secret and one to access an array based on the secret value [10]. However, such gadgets can be difficult to find in real-world software [25]. Furthermore, branches are known targets for Spectre attacks, and memory fences are a viable solution against these attacks. While other works explored further covert channels beyond the cache to exfiltrate data from transient execution [7, 70], they still require the attacker to have control over the corresponding covert channel and to find gadgets that encode the data accordingly into this covert channel. Instead, the `UOPS_EXECUTED.CORE` performance counter tracks the number of **any** uOPs executed, even speculatively. Consequently, we can use the `UOPS_EXECUTED.CORE` performance counter as a covert channel receiver that is influenced by any secret-dependent code execution. The change in uOP throughput can be the result of a difference in the microarchitectural state, e.g., if cached memory is accessed in contrast to non-cached memory, or simply different instructions are executed due to a conditional branch, or any other microarchitectural difference.

```
1 if (expression) {
2   //...
3   function_ptr(...);
4   //...
5 }
```

**Listing 4: Example victim code, similar to Göktaş et al. [25]. The attacker can control the function pointer during a speculative execution, but never during actual execution.**

For our Spectre attack, we focus on comparing gadgets, which are abundant in the Linux kernel [10] and not considered in Spectre mitigation efforts so far that focused on index gadgets [31]. Unlike index gadgets that rely on a secret-dependent memory access, compare gadgets only leak a single bit per speculative branch, *i.e.*, branch taken or not taken. While the `UOPS_EXECUTED.CORE` channel is a very generic Spectre receiver, uOPs executed can also be a noisy channel. Still, we believe the significantly broadened scope of possible gadgets makes it an interesting channel for attackers.

For our experiments, we use the threat model described in Section 4.1 and assume the existence of victim code similar to the one shown in Listing 4, in line with prior work [25]. The attacker can control the value of the function pointer that is being called in speculation and two arguments passed to it. However, importantly, the attacker's chosen function pointer is **only** executed in speculation, never during actual execution. Furthermore, we note that this is just one gadget that is realistic based on prior work [25], whereas there is an abundance of further compare gadgets e.g., in the Linux kernel that may also leak information [10].

### 5.2 KASLR Break

KASLR randomizes the virtual addresses of memory mapped in the kernel space. Given that it has virtually no performance overhead, it is widely deployed in modern operating systems. Practically, it is often the first line of defense for an attacker to cross: The attacker often needs to know the address of specific code or data in the kernel, e.g., to leak data from this specific location, modify memory, or reuse code for the attack [28, 39]. Guessing the KASLR offset is often not feasible due to the severe consequences if the attacker guesses wrong [20, 29], e.g., crashing the system.

To circumvent this challenge, prior work used various microarchitectural attacks [9, 28, 39, 42] and Spectre attacks [25, 38, 40, 52]. Similarly, we also use a Spectre attack to break KASLR on Intel TDX, exploiting the `UOPS_EXECUTED.CORE` performance counter channel. We target the kernel code region, which is mapped using 2 MB pages above `0xffffffff80000000` on `x86_64` Linux, with its exact location randomized every boot. To leak the KASLR offset, we exploit that `UOPS_EXECUTED.CORE` counts speculatively executed

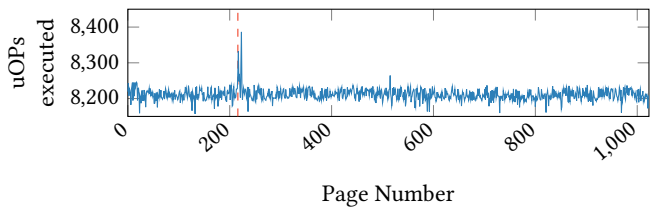



**Figure 7: KASLR break using our Spectre attack with the `UOPS_EXECUTED.CORE` channel. Each point is the median of 30 measurements on a single 2 MB page starting at the start of the Linux kernel code ASLR region (`0xffffffff80000000` or page number 0 in the plot). Kernel code starts at page number 216 (marked by the dashed line), which is clearly visible through a spike in uOPs executed in the plot.**

uOPs. To probe a virtual address of a victim TD, we let the TD speculatively execute it using the gadget shown in Section 5.1. For addresses with executable pages mapped, the processor can execute the code on them speculatively, leading to an increased amount of uOPs executed. Addresses without executable memory mapped lead to a stall until the CPU determines that a misprediction occurred.

We performed measurements every 2 MB (30 times per page) starting at the beginning of the Linux kernel code ASLR range (`0xffffffff80000000`), on our Intel Xeon Silver 4514Y system and provide the results in Figure 7. The baseline is visible at ~8 200 uOPs, which corresponds to no code page being mapped at these locations. The first increase in uOPs is at page number 216, corresponding to virtual address `0xffffffff9b000000`. This aligns exactly with the virtual address provided by `/proc/kallsyms` in the victim TD. We can determine the correct KASLR offset in <2 seconds, requiring 30 measurements per page. This is the same performance range as prior KASLR breaks [9, 25, 28, 38–40, 42, 52] albeit on Intel TDX.

## 5.3 Leaking Arbitrary TD Memory

Being able to leak arbitrary memory provides the host with all secrets located inside the TD, such as encryption keys and other sensitive information, breaking the confidentiality of TDX. To leak memory with a Spectre attack based on the `UOPS_EXECUTED.CORE` channel, we require nothing more than a gadget that varies the number of uOPs executed depending on some input, e.g., any compare and index gadgets. Such gadgets are common in the Linux kernel [10], e.g., every time a flag in memory is checked, memory locations are compared. Which gadgets exactly are exploitable depends on the exact registers the attacker can control. For our attack with the vulnerable code described in Section 5.1, we use a compare gadget in the `memchr` function.

The first few instructions of the `memchr` in our Linux `6.8` kernel are provided in Listing 5. Line 4 compares the byte stored at the memory location held in the `rdi` register and the `sil` register, which is the lowest significant byte of the `rsi` register on x86. This comparison is followed by a conditional branch, which leads to different code being executed depending on whether the branch is taken. To leak specific values, we let the TD trigger our gadget for

```
1 add rdx, rdi
2 jmp 0xe
3 lea rax, [rdi + 1]
4 cmp byte [rdi], sil
5 je  0x13
6 mov rdi, rax
7 cmp rdi, rdx
```

**Listing 5: Beginning of the disassembled Linux kernel `memchr` function. The compare operation in line 4 is a Spectre compare gadget leading to different instructions executed and leaking the result of the comparison.**

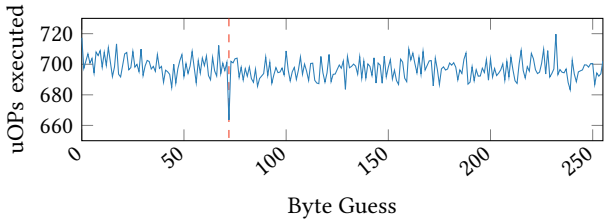



**Figure 8: Example of leaking bytes using `memchr` (Listing 5) as a Spectre gadget. Each data point is the average of 10 executions for a byte guess. When the guessed byte and the target byte are the same at byte 71 (dashed line), the compare is true, leading to a jump and a lower number of uOPs executed due to the different speculative execution path.**

each possible value for `sil`. We repeat this 10 times and compute the mean to eliminate noise.

The results of leaking a byte by using the `memchr` compare gadget are shown in Figure 8. The average uOPs executed hover around 700 cycles for all byte guesses except for byte guess 72. This is the result of the comparison in line 4 returning false for most byte guesses, therefore, not taking the branch in line 5. With byte guess 72, the number of executed uOPs drops to ~660 cycles. As this is the correct value, the branch is taken during speculation, leading to different instructions and, consequently, a different number of uOPs being executed. With our attack, we are able to leak memory at a rate of 52.6 bit/s ($n = 800$, $\sigma_{\bar{x}} = 0.03$), with an error rate of 0.6 % ($n = 800$, $\sigma_{\bar{x}} = 0.02$). This is slower than prior Spectre attacks [25, 41], but still completely breaks the confidentiality of Intel TDX.

# 6 SSH Keystroke Timing Attack

Inter-keystroke timing attacks have been demonstrated from various environments using different techniques [27, 55, 63, 65, 73, 86]. Song et al. [73] showed that attackers can detect keystrokes by observing the encrypted network traffic of an SSH session. All network traffic to a TD is forwarded through the host, making it possible for the host to observe the network traffic. To mitigate against this attack, OpenSSH implements the `ObscureKeystrokeTiming` feature [58], which hides the real keystroke packets by sending additional fake interactive packets in short intervals. The fake packets

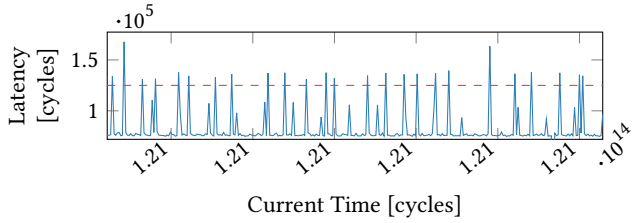



**Figure 9: Latency between forwarding an SSH package to the TD and the network response of real and fake OpenSSH keystroke packages. Real keystroke packages take more than 125 000 cycles (marked by a dashed line) to process on our system, while fake keystrokes are processed significantly faster. This makes keystrokes easily detectable for a malicious host.**

cannot be distinguished from real keystroke packets by observing the network traffic, and are treated as ping packets by the SSH server. However, we can distinguish the fake and real packets by observing the CVM through precise timing measurements.

## 6.1 Attack

Because the attacker controls the host, they can observe the network traffic to and from the TD. Thus, the attacker knows exactly when the TD receives an SSH packet and when it responds to it. To distinguish fake and real packets, we use the precise timing between when the network package is delivered and when the TD sends a response. In OpenSSH `10.0p2`, when `ObscureKeystrokeTiming` is enabled, the client periodically (default: every 20 ms) sends a `SSH2_MSG_PING` packet. To an attacker, the encrypted network packages stemming from real keystrokes and `SSH2_MSG_PING` packets are indistinguishable from each other. When the server receives a `SSH2_MSG_PING` packet, it responds with a `SSH2_MSG_PONG`. This response is sent immediately in the `ssh_packet_read_poll_seqnr` function. In contrast, normal keystroke packets cause the function to return and look up the appropriate handler function in the `*ssh->dispatch` table, which eventually forwards the keystroke to the application. As both execution paths are not identical and require a different amount of work, it should be possible to distinguish them through precise timing measurements.

To determine the processing time of a package, we measure the time between the interrupt for the network packet being injected into the TD and the MMIO TDcall by the guest to send the response packet. In the TDX threat model, interrupts that the TD receives are untrusted, as the host can decide when or even if an interrupt is injected into the guest. We, therefore, do not inject any interrupts into the guest in between the network packet being delivered and the guest responding. As this includes the guest timer interrupts, this essentially disables preemption for the TD, eliminating the threat of unwanted threads being scheduled in the TD.

A precise timing trace of network packet response times of an SSH connection with `ObscureKeystrokeTiming` enabled is shown in Figure 9. Each data point is a single timing measurement done using the host's timestamp counter (TSC). There is a clear baseline visible at ~75 000 cycles or 37.5 µs, which corresponds to `SSH2_MSG_PING` used by the mitigation. A large number of spikes stand out from this

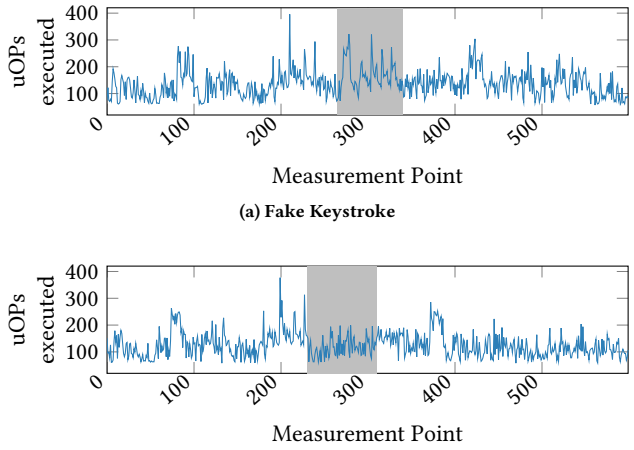



**Figure 10: `UOPS_EXECUTED.CORE` traces of a fake keystroke (Figure 10a) and a real keystroke (Figure 10b) for OpenSSH. The main difference between the two traces is highlighted in gray. The fake keystroke trace has multiple spikes of high uOP throughput, while the real keystroke trace lacks these spikes.**

baseline. The spikes above ~125 000 cycles (62.5 µs), marked with a dotted vertical line, are keystroke packages that are being processed. This small timing difference would be extremely difficult to measure over a regular network, as it is small enough to completely vanish through regular OS operation, such as scheduling. Despite this, these timing differences can be easily and reliably measured under the regular CVM threat model.

*Performance Counter.* While we focus our evaluation on precise timings as they are currently enough to differentiate real keystrokes from fake keystrokes, `UOPS_EXECUTED.CORE` can also be used to distinguish between the two packet types. A part of the uOP traces of the processing for real keystrokes and fake keystrokes generated by OpenSSH are shown in Figure 10. Despite two traces having multiple similar features, such as a plateau between measurement points 80 and 100, as well as a slight increase in uOPs executed around measurement point 200, there are multiple differences, making them reliably distinguishable. One of these differences is highlighted in gray in Figure 10a and Figure 10b. For fake keystrokes, there are large spikes in uOPs executed in this area, which are missing for real keystrokes. As this is a very distinct difference, performing inter-keystroke timing attacks on OpenSSH would be possible through `UOPS_EXECUTED.CORE` if the OpenSSH team mitigates the currently existing timing side channel.

## 6.2 Evaluation

We performed our attack on a TD running OpenSSH version `10.0p2` on Ubuntu 24.04, following the threat model outlined in Section 4.1. We enabled `ObscureKeystrokeTiming` with the default settings, resulting in fake packets being sent approximately every 20 ms. Our experiments were performed with one of the authors typing at their regular speed inside of a text file edited using VIM inside of

the TD through an SSH connection. Overall, 418 keys were typed. Of the 418 keys, all were detected correctly with 3 false positives. This results in an $F_1$ score of 99.6 %, making it an extremely reliable attack. The high $F_1$ stems from the high amount of control the host has over the environment in this threat model, allowing for the elimination of noise sources such as interrupts and scheduling.

While correctly detecting a majority of the keystrokes is important for an inter-keystroke timing attack, a low variation in the detection latency is vital to recover words from the recorded information. For our experiment, we measured an average latency of 60 ms between the key being pressed and the interrupt for the key being injected into the guest. The standard deviation of the latencies is 5.64 ms. While the latency itself seems high, it itself is irrelevant for an inter-keystroke timing attack. As long as all packages have a similar delay, the timings of the key presses relative to each other (which is holding the information) remain the same, making the standard deviation the relevant metric for this kind of attack. Our standard deviation of 5.64 ms is significantly lower than the average inter-keystroke interval of 120 ms for fast typists [19] and similar to existing inter-keystroke timing attacks [62, 68], making it a viable channel for this attack.

## 7 Discussion & Related Work

In this section, we discuss related work, as well as possible mitigations for our attacks. We first discuss the performance counter-based attacks and finish with the attack on the OpenSSH keystroke detection mitigation.

### 7.1 Performance Counter Attacks

Unlike the mitigations against single-stepping and performance-counter-based attacks on the same logical core that Intel employs, our attacks can not be mitigated through a change in the TDX module. The TDX module can not hinder sibling logical cores from using certain performance counters. The best way to mitigate this issue is to not count performance counter events of sibling logical cores across TD boundaries. While we do not believe that this can be done on existing hardware, it is the best long-term solution.

For existing hardware, the best option is to disable hyperthreading. With hyperthreading disabled, there is no sibling logical core available to collect data from performance counters that monitor the whole core. Whether hyperthreading is enabled is attestable, making this enforceable by the TD and very easy to implement. The main disadvantage of this mitigation is the loss in performance.

Alternatively, the TDX module could enforce that all logical cores of a physical core have to be either inside of one TD or in the host. With both logical cores inside of the same TD, the host can not monitor the performance counters. To enforce this, the TDX module could block when entering a TD until both logical cores are ready to enter it. For VM exits, whenever one logical core performs a VM exit, the TDX module could send an IPI to the other logical core, forcing a VM exit. This would lead to performance loss due to unnecessary VM exits and VM entries blocking until both cores are ready, but this loss in performance should be significantly smaller than disabling hyperthreading.

To mitigate the impact of `UOPS_EXECUTED.CORE` on Spectre attacks, compilers can insert fences between compares and their subsequent branch [31]. We note that this is not the same as simply disabling branch prediction. Since this would lead to a significant loss in performance [10], other mitigation options may be more desirable. Additionally, this approach would only mitigate the use of `UOPS_EXECUTED.CORE` in Spectre attacks and not any other attack vectors stemming from the discussed performance counters. Our new side channel is most relevant when using a gadget that, itself, does not lead to a branch prediction, but instead is executed during speculation. It is necessary for our side channel that the branch is correctly evaluated, as it leaks the outcome of the branch.

Closest to our work is CounterSEVeillance by Gast et al. [22]. They use performance counters on the same logical core to attack a TOTP implementation, the MbedTLS RSA implementation, and perform a divide-and-surrender-style attack on a HQC-KEM implementation running in an AMD SEV-SNP CVM. To perform these attacks, they mainly take advantage of performance counters tracking branches. While CounterSEVeillance [22] and our attacks are similar on the surface, there are key differences: First, the attack style of CounterSEVeillance monitors the performance counters on the same logical core and assumes that performance-counter-based attacks are mitigated if performance counters are context switched when entering a TEE, explicitly mentioning that Intel TDX implements this mitigation. Our attacks take advantage of a performance counter that can monitor the victim from the sibling logical core, bypassing Intel's current defense. Second, in addition to traditional attacks with performance counters, we explore Spectre-type attacks using the information gained by performance counters, allowing for the use of a wide range of potential new gadgets. Third, we do not rely on single-stepping for any of our attacks, instead leveraging the highly detailed information `UOPS_EXECUTED.CORE` provides us to leak secret information. Lastly, we show a completely novel attack that bypasses the recently introduced mitigation against inter-keystroke timing attacks in OpenSSH.

Lou et al. [51] also attack AMD SEV-SNP with performance counters and perform website fingerprinting and keystroke detection. The performance counters are monitored on the same logical core as the victim, which is not possible on Intel TDX. Similar to Gast et al. [22], Lou et al. [51] mention that Intel TDX already protects against performance counter-based attacks and recommend a similar mitigation for AMD SEV-SNP. We show that this defense is insufficient and still makes leakage through some performance counters possible.

Cho et al. [15] and Li et al. [45] use performance counters to detect malicious applications, which could also be applied to TEEs, making a case for providing some performance counter information available to the host. Weissteiner et al. [81] try to strike a balance between confidentiality and thread detection by decorrelating reported performance counter values from the actual hardware events in TEE environments, allowing performance counters to be enabled while protecting against fine-grained information leakage. However, this mitigation does not protect against information leakage through shared performance counters, as it only applies the decorrelation when context switching from the host to the TEE. Thus, the performance counters from a sibling logical core are not decorrelated and still leak information.

Mandal et al. [53] use performance counters to monitor the host application managing the TD and detect contention effects through

them. By monitoring performance counters such as instructions executed, L1 dcache misses, and branch misses, they can determine whether the throughput on the sibling logical core of the TD changes due to contention. Their attack does **not** directly leak information from the TD through the performance counters, as all performance counters listed only monitor the current logical core and are context switched whenever entering and exiting a TD. Due to the limited information, Mandal et al. [53] only perform application fingerprinting. In contrast, our work found a small subset of performance counters that allow for direct monitoring of the TD workload, as they capture data for the whole physical core, even when one of the two logical cores is currently running a TD. This allows us to perform much more fine-grained attacks, such as an RSA private key recovery and Spectre-type attacks.

### 7.2 OpenSSH

To mitigate the OpenSSH inter-keystroke timing attack, the most effective approach would be to implement one processing path for real and fake keystroke packages to avoid a deviation in timing. Due to the difference in how real keystrokes have to be processed, e.g., , there might be a response package with new information that is displayed, but this might not be fully possible. An approximation of this would be to delay the fake keystroke responses of the mitigation into a similar timing range as the last real keystrokes being responded to. The fake keystrokes do not have to be able to mimic the timings of real keystrokes perfectly. Inter-keystroke timing attacks rely on precise timing differences between individual keystrokes. Therefore, as long as large parts of the keystrokes can not be differentiated from fake keystrokes, the timing differences are not valuable for the recovery of the typed words. Another possible angle to defend against these types of attacks would be to delay keystrokes from being sent for a random amount of time. If the random delay range is large enough, e.g., in the range of a few 100 ms, the inter-keystroke timings are also no longer useful for recovering words. This approach has the disadvantage that it does not hide the number of characters typed, but only obfuscates the timings between them. Additionally, this random delay can make the SSH connection less responsive, which might discourage some users from activating this mitigation.

Giavridis [23] discovered that the OpenSSH inter-keystroke timing defense in version `9.7` can be easily bypassed by measuring the response times of packages. Keystroke packages take 3 times longer (60 ms) to process than fake keystrokes (20 ms). This attack was mitigated starting OpenSSH `9.8p1`. Despite the mitigation, we show that there is still a significant timing difference between response times for SSH packages, which can easily be exploited in a CVM scenario running the most recent OpenSSH version (`10.0p2`). Lipp et al. [47] leaked keystrokes from sandboxed JavaScript with an identification rate of 81.75 %. Rauscher et al. [62] measure the timings of inter-processor interrupts to detect other interrupts and performed an inter-keystroke timing attack with a standard deviation of 6.15 ms and an $F_1$ score of 97.9 %. Schwarz et al. [68] propose generating random keystrokes to mitigate inter-keystroke timing attacks, similar to the OpenSSH mitigation.

## 8 Conclusion

Intel's recent CVM extension, Intel TDX, actively mitigates performance counter attacks by context switching relevant registers. Despite these mitigations, we uncovered a fatal flaw in this mitigation, as it does not account for leakage through counters that monitor the whole physical core. We analyzed the available performance counters on our recent Intel CPU and found 8 that allow for information leakage from the sibling logical core, even when the other core is inside of a TD. We uncovered one particular counter, `UOPS_EXECUTED.CORE`, that provides information on uOPs executed, which can be exploited to leak sensitive information on the execution flow inside of a TD. With `UOPS_EXECUTED.CORE`, we attack the MbedTLS RSA-2048 implementation running inside of a TD and leak the full private key with an average Levenshtein distance of only 0.92 bits. While this attack already shows how dangerous this exposed information can be, we further abuse that `UOPS_EXECUTED.CORE` does not only count uOPs of instructions that are committed, but also of speculatively executed instructions. This enables the use of a wide range of new gadgets for Spectre attacks by relying on uOPs executed during speculation instead of other side effects, such as memory accesses. We demonstrate how dangerous this can be by using the Linux kernel's `memchr` implementation to leak arbitrary TD memory at a rate of 52.6 bit/s and by breaking KASLR in less than 2 s. In addition to performance counter-based attacks, we discovered that the novel inter-keystroke timing defense of OpenSSH is insufficient for a CVM scenario, allowing us to detect real keystrokes with an $F_1$ score of 99.6 %, despite the active mitigation. We conclude that the current mitigations against performance counter attacks in Intel TDX are incomplete and require further refinement to protect against this threat.

## Ethics Considerations

We responsibly disclosed our findings to Intel on August 7, 2025, and to the OpenSSH team on August 25, 2025. Both Intel and the OpenSSH team have products that are impacted by our attacks. We conducted our research to highlight the existence of the found vulnerabilities and provide the stakeholders with the opportunity to mitigate them before they are published. While the existence of new attacks can be seen as a negative, it is vital that researchers uncover these vulnerabilities and adequately disclose them to the vendors before a malicious third party can find and exploit them. The publication of such attacks is also essential, as it is a reference for future developers not to introduce similar vulnerabilities into their products. We have proposed mitigations for both existing as well as future platforms to minimize any negative side effects of our research. Lastly, all our experiments were done on our own machines with no code from other users running on them.

## Acknowledgements

This research is supported in part by the European Research Council (ERC project FSSec 101076409) and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from Red Hat, and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

## References


[1] 2025. OpenSSH. https://www.openssh.com
[2] 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
[3] AMD. 2024. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/
[4] ARM. 2020. mbed TLS. https:///tls.mbed.org
[5] ARM. 2024. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture
[6] ARM. 2024. TrustZone for Arm Cortex-M Processors. https://www.arm.com/technologies/trustzone-for-cortex-a
[7] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandt ner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*.
[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*.
[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*.
[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*.
[11] Canonical. 2025. Intel® Trust Domain Extensions (TDX) on Ubuntu. https://github.com/canonical/tdx
[12] Stefano Carnà, Serena Ferracci, Francesco Quaglia, and Alessandro Pellegrini. 2023. Fight Hardware with Hardware: Systemwide Detection and Mitigation of Side-channel Attacks Using Performance Counters. *Digital Threats: Research and Practice (DTRAP)* 4, 1 (2023), 1–24.
[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*.
[14] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. 2020. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX Security*.
[15] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. 2020. Real-time detection for cache side channel attack using performance counter monitor. *Applied Sciences* 10, 3 (2020), 984.
[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086* (2016).
[17] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *S&P*.
[18] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *S&P*.
[19] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. 2018. Observations on typing from 136 million keystrokes. In *CHI Conference on Human Factors in Computing Systems*.
[20] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/
[21] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*.
[22] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *NDSS*.
[23] Philippos Maximos Giavridis. 2024. SSH Keystroke Obfuscation Bypass . https://crzphil.github.io/
[24] Lukas Giner, Sudheendra Raghav Neela, and Daniel Gruss. 2025. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In *DIMVA*.
[25] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*.
[26] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *EuroSec*.
[27] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. 2019. Page Cache Attacks. In *CCS*.
[28] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.
[29] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.
[30] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2020. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In *CHES*.
[31] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf
[32] Intel. 2021. Intel Trust Domain Extensions. https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf
[33] Intel. 2024. Intel Performance Monitoring Events. https://perfmon-events.intel.com/
[34] Intel. 2024. Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html
[35] Intel. 2024. Intel Trust Domain Extensions Module Base Architecture Specification. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html
[36] Intel. 2025. Intel(R) Software Guard Extensions Developer Guide. https://cdrdv2-public.intel.com/671334/intel-sgx-developer-guide.pdf
[37] Intel. 2025. TDX Module 1.5.16 Source Code. https://github.com/intel/tdx-module
[38] Hyerean Jang, Taehun Kim, and Youngjoo Shin. 2024. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In *CCS*.
[39] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*.
[40] Yu Jin, Chunlu Wang, Pengfei Qiu, Chang Liu, Yihao Yang, Hongpei Zheng, Yongqiang Lyu, Xiaoyong Li, Gang Qu, and Dongsheng Wang. 2024. Whisper: Timing the Transient Execution to Leak Secrets and Break KASLR. In *DAC*.
[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.
[42] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P*.
[43] 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 *USENIX Security*.
[44] Congmiao Li and Jean-Luc Gaudiot. 2018. Online detection of spectre attacks using microarchitectural traces from performance counters. In *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.
[45] Congmiao Li and Jean-Luc Gaudiot. 2019. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In *COMPSAC*.
[46] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security*.
[47] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS*.
[48] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*.
[49] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*.
[50] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.
[51] Xiaoxuan Lou, Kangjie Chen, Guowen Xu, Han Qiu, Guo Shangwei, and Tianwei Zhang. 2024. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In *DSN*.
[52] Giorgi Maisuradze and Christian Rossow. 2018. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *arXiv:1801.04084* (2018).
[53] Upasana Mandal, Shubhi Shukla, Nimish Mishra, Sarani Bhattacharya, Paritosh Saxena, and Debdeep Mukhopadhyay. 2025. Exploring side-channels in Intel Trust Domain Extensions. *Cryptology ePrint Archive* (2025).
[54] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*.
[55] John Monaco. 2018. SoK: Keylogging Side Channels. In *S&P*.
[56] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*.
[57] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. 2018. Spectre attack against SGX enclave.
[58] OpenBSD Journal. 2023. Keystroke timing obfuscation added to ssh(1). https://undeadly.org/cgi?action=article;sid=20230829051257
[59] Perf Wiki. 2020. Main Page. https://perf.wiki.kernel.org/index.php/Main_Page
[60] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *CCS*.

[61] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *AsianHOST*.
[62] Fabian Rauscher and Daniel Gruss. 2024. Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In *CCS*.
[63] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. 2024. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In *NDSS*.
[64] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. 2025. TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX. In *USENIX Security*.
[65] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*.
[66] Keegan Ryan. 2019. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In *CCS*.
[67] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.
[68] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.
[69] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.
[70] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*.
[71] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2020. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity* 3, 1 (2020), 2.
[72] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. 2021. Speculative Dereferencing of Registers: Reviving Foreshadow. In *FC*.
[73] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. 2001. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security*.
[74] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*.
[75] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.
[76] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*.
[77] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*.
[78] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice.
[79] 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 *CCS*.
[80] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. 2018. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *AsiaCCS*.
[81] Hannes Weissteiner, Fabian Rauscher, Robin Leander Schröder, Jonas Juffinger, Stefan Gast, Jan Wichelmann, Thomas Eisenbarth, and Daniel Gruss. 2025. TEEcorrelate: An Information-Preserving Defense against Performance Counter Attacks on TEEs. In *USENIX Security*.
[82] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. 2024. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *CCS*.
[83] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity–Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *S&P*.
[84] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. 2024. SEV-Step: A Single-Stepping Framework for AMD-SEV. In *CHES*.
[85] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*.
[86] Kehuan Zhang and XiaoFeng Wang. 2009. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security*.
[87] Ruiyi Zhang, CISPA Helmholtz Center, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*.
[88] Hongwei Zhou, Xin Wu, Wenchang Shi, Jinhui Yuan, and Bin Liang. 2014. HDROP: Detecting ROP Attacks Using Performance Monitoring Counters. In *ISPEC*.