Indirect Meltdown: Building Novel Side-Channel Attacks from Transient-Execution Attacks

Daniel Weber, Fabian Thomas, Lukas Gerlach,
Ruiyi Zhang, and Michael Schwarz

CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
<firstname>.<lastname>@cispa.de

Abstract. The transient-execution attack Meltdown leaks sensitive information by transiently accessing inaccessible data during out-of-order execution. Although Meltdown is fixed in hardware for recent CPU generations, most currently-deployed CPUs have to rely on software mitigations, such as KPTI. Still, Meltdown is considered non-exploitable on current systems.

In this paper, we show that adding another layer of indirection to Meltdown transforms a transient-execution attack into a side-channel attack, leaking metadata instead of data. We show that despite software mitigations, attackers can still leak metadata from other security domains by observing the success rate of Meltdown on non-secret data. With LeakIDT, we present the first cache-line granular monitoring of kernel addresses. LeakIDT allows an attacker to obtain cycle-accurate timestamps for attacker-chosen interrupts.

We use our attack to get accurate inter-keystroke timings and fingerprint visited websites. While we propose a low-overhead software mitigation to prevent the exploitation of LeakIDT, we emphasize that the side-channel aspect of transient-execution attacks should not be underestimated.

1 Introduction

Microarchitectural side-channel attacks have been known for several years [27]. These attacks exploit the side effects of CPU implementations to infer metadata about processed data. Well-known examples of microarchitectural side-channel attacks include cache attacks, e.g., Flush+Reload [64] or Prime+Probe [40], which have been used to leak cryptographic secrets [2,64] or violate the privacy of users, e.g., by spying on user input [39,17,48]. The discovery of transient-execution attacks, such as Meltdown [35] and Spectre [26], was a game changer for microarchitectural attacks, as these directly leak data instead of metadata. Hence, even best practices for side-channel-resistant software [23,11] do not protect secrets anymore. In Meltdown attacks, architecturally inaccessible data is accessed during out-of-order execution and encoded into a microarchitectural element, e.g., the cache, protected from the pipeline flush [35,44]. A subsequent side-channel attack, e.g., Flush+Reload, converts the microarchitectural into an architectural state, revealing the data.

As only new CPU generations contain hardware fixes for Meltdown-type attacks, short- and mid-term mitigations rely on software workarounds. These workarounds ensure that no confidential data is stored in affected buffers when untrusted code is executed [57,49,21] or that the victim data is not addressable [14,54]. For Meltdown-US-L1 [35], i.e., the original Meltdown attack, the OS unmaps the majority of its address space while running in user space, making sensitive data non-addressable [15]. The remaining mapped pages are not considered confidential, such that Meltdown-US-L1 is considered not exploitable. On Linux, this technique is implemented as kernel page-table isolation (KPTI) [12].

In this paper, we show that even with state-of-the-art mitigations, Meltdown can be transformed from a transient-execution attack into a side-channel attack. The main idea is based on two properties. First, while KPTI unmaps most kernel pages, several kernel pages with non-secret content are necessary on x86 CPUs and cannot be unmapped in user space. Second, Meltdown [35] can only leak data if it is cached in the L1D cache, making it usable as a cache-state oracle. Combining these two properties leaks the meta information on whether (non-confidential) kernel data was accessed. Hence, Meltdown can be used as a high-resolution cache attack with cache-line granularity on the kernel. This side channel is superior to state-of-the-art cache attacks on the kernel, which only achieve page [32] or cache-set granularity [48].

We gain an interesting insight from this attack:

While a layer of indirection is necessary for Meltdown to leak data, another layer of indirection transforms the attack to leak metadata of architecturally inaccessible data.

In other words, exploiting a modified version of the Meltdown attack enables the leakage of metadata that cannot be leaked in this granularity with a traditional side-channel attack.

Based on this, we present LeakIDT, a side-channel attack able to spy on interrupts. We exploit that the interrupt descriptor table (IDT) must always be mapped on x86 [19,15]. Hence, despite software mitigations such as KPTI, an attacker can use the side channel to monitor interrupt activity. In contrast to previous works that exploit interrupts as a side channel [33,48,56], LeakIDT can target specific interrupts, e.g., network or keyboard interrupts, instead of just observing that *any* interrupt occurred and works for unprivileged attackers. We identify which website a user visits from the Alexa top 15 and top 100 websites with a precision of 80 % and 55 %, respectively. Furthermore, we reliably observe keystroke timings with an average F-score of 0.89. We propose to mitigate LeakIDT by marking the IDT uncachable, preventing any entry from being cached. This mitigation is practical, with an average performance overhead of less than 0.5 % in 5 different benchmarks simulating real-world workloads.

Our attacks show that while mitigating data leakage is essential, the side-channel aspect of such fixes can be overlooked. We show that adding additional layers of indirection to existing attacks can change their properties. As a result, we create a new side-channel attack from a CPU vulnerability commonly considered unexploitable when applying state-of-the-art software mitigations. Hence,

we argue that future software workarounds should consider the side-channel aspect to prevent such attack vectors. Thus, we encourage researchers to look at other mitigations for hardware vulnerabilities to determine whether they can be circumvented to repurpose the underlying vulnerability as a side channel. For this purpose and to ease reproducibility, we open-source the code of our findings on GitHub[1].

To summarize, we make the following contributions:

1. We show that adding another layer of indirection to Meltdown transforms Meltdown into a side channel that infers the cache state of non-sensitive kernel pages with cache line granularity, leaking details about, e.g., interrupts.
2. We use our side channel to detect the visited websites of a user and spy on their keystroke timings.
3. We present a practical mitigation that stops our attack, while introducing an average overhead of less than 0.5 % for real-world workloads.

Responsible Disclosure. We disclosed our findings to Intel on February 15, 2023 and AMD on February 16, 2023. Despite both vendors acknowledging our findings, they informed us that they do not plan to roll out further mitigations.

2 Background

In this section, we provide the background for this paper. We introduce side channels, transient-execution attacks, and the interrupt descriptor table.

2.1 Side Channels

Side channels leak metadata of (secret) information. In a side-channel attack, an attacker infers secrets from this metadata. For leaking metadata, secret-dependent observable differences must exist, e.g., response time or power consumption that depends on the bits of a cryptographic key. Previous research showed that side channels can be practical tools in an attacker's repertoire [27,38], especially for attacking cryptographic implementation [27,38]. In recent years, researchers have shown various side-channel attacks exploiting microarchitectural components [38,39,61,60,41]. These components include CPU caches [64,40], branch predictors [4,3], execution units [13,61], DRAM components [41], and power usage [63]. The fundamental property that enables such microarchitectural side-channel attacks is that different processes share many hardware components. Hence, the resource usage of one process affects the possible resource usage of another process, leaking meta information between the processes.

2.2 Transient-Execution Attacks

Two important performance optimizations in modern CPUs are out-of-order execution and speculative execution. Out-of-order execution allows the CPU to

[1] `https://github.com/cispa/indirect-meltdown`

reorder or parallelize the execution of instructions in the instruction stream. Speculative execution predicts the outcome of branch and memory load instructions, reducing pipeline stalls. Executed instructions that never commit their state changes to the architecture due to a misspeculation or preceding fault are called transient instructions [8,24]. Transient-execution attacks [8] exploit transient instructions to read otherwise inaccessible memory [35,26]. While transient instructions do not have an architectural effect, they can influence microarchitectural states, such as cache states. These traces can be converted to architectural states using a microarchitectural side channel, e.g., Flush+Reload. In recent years, researchers and CPU vendors discovered a variety of transient-execution attacks [35,26,6,49,57,37,55,36,28,46,45,44,8].

One category of transient-execution attacks are Meltdown-type attacks [8]. The first discovered Meltdown-type attack, later referred to as Meltdown-US-L1 [8], allows unprivileged attackers to leak cached kernel memory. After a faulting load to a kernel address, the value is transiently available and can be encoded in the microarchitecture, e.g., in the cache. The attacker decodes the encoded value using a side channel, e.g., using Flush+Reload. Meltdown-type attacks, and especially Meltdown-US-L1, affect a variety of modern CPUs [22].

2.3 Interrupt Descriptor Table (IDT)

Devices, such as network interface controllers or keyboards, use interrupts to notify the OS of events, e.g., incoming network packets or key presses. On an interrupt, the CPU switches to ring 0 and looks up the corresponding interrupt service routine (ISR) for the specific interrupt in the interrupt descriptor table (IDT). The CPU interrupts the current execution and jumps to the ISR to handle the interrupt. After handling the interrupt, the CPU continues executing the previous instruction stream. We only consider the 64-bit x86 IDT. Each core can have its own IDT containing 256 interrupt vectors [20, Chapter 6.10 & 6.14]. Each interrupt vector is 16 bytes in size and represents one device [20, Chapter 6.10 & 6.14]. Hence, the IDT has a total size of 4 kB, i.e., one memory page, and is stored in the main memory. Each of these interrupt vectors essentially consists of a 64-bit (8-byte) pointer to its ISR in the kernel. The remaining 8 bytes store additional meta-information about the interrupt, such as the type and the privilege level of the interrupt [20, Chapter 6.14]. The base pointer to the IDT is stored in a CPU-internal register, which can be read with the `sidt` instruction. On modern Linux systems, the IDT is hard-coded to `0xfffffe0000000000` [31].

3 Meltdown as a Side Channel

In this section, we introduce the concept of transforming the transient-execution attack Meltdown into a side channel. The main idea is that the success rate of Meltdown-US-L1 reveals the cache state of the target memory address. We discuss which kernel memory ranges are still mapped despite the KPTI mitigation and how Meltdown-US-L1 can be used to leak metadata about these memory

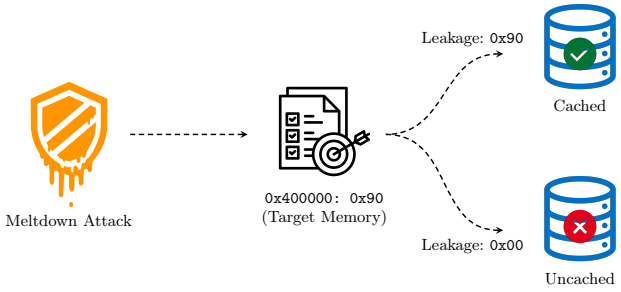

Fig. 1: Meltdown as a side channel. The Meltdown attack only leaks data if the target address is in the L1D cache. Otherwise, the value `0x00` is leaked.

```
1 ; rax = kernel address, rcx/rbx= probe page 1/2,
2 cmp     [rax], 0x0
3 cmovne   rcx, rbx
4 mov      rax, [rcx]
```

Listing 1: Using Meltdown-US-L1 as a side channel. If the target kernel address is cached, the user address stored in `RBX` is cached. Otherwise, the user address stored in `RCX` is cached.

pages. For a list of CPUs affected by Meltdown-US-L1 and thus affected by our attack, we refer the reader to the Intel's list of vulnerable CPUs [22].

While Lipp et al. [35] discussed that Meltdown-US-L1 works best if the target address is stored in the L1D cache, Xiao et al. [62] and Schwarzl et al. [50] show that Meltdown-US-L1 is limited to the L1D cache. Leakage from other cache levels is only caused by prefetching the data into the L1 cache, e.g., via speculative execution. We exploit this requirement to use Meltdown as a side channel: If data is leaked via Meltdown-US-L1, it is in the L1D cache. An illustration of this concept is given in Figure 1.

By detecting whether the target memory address can be leaked, we learn whether it was previously accessed. If the cache-line content can be leaked, the cache line is cached in the L1D, which is only the case if the cache line was recently accessed. As this attack can be applied to any mapped memory address, we can also use it on kernel memory pages that are mapped while in userspace. This converts Meltdown-US-L1 into an Evict+Reload-style side channel for kernel memory.

Attack Details. Listing 1 shows the implementation of the encoding step when using Meltdown-US-L1 as a side channel. We compare the content of the kernel address to zero (Line 2) and, based on the result, select (Line 3) and access (Line 4) one out of two different pages. This works as the access transiently results in a zero if no value can be leaked by Meltdown-US-L1. Otherwise, the result is non-zero if the targeted memory address is non-zero. This code sequence

is simpler than the Meltdown-US-L1 code [35] that transiently loads the value at the kernel address into a register and accesses one out of 256 pages based on the loaded value, since we only need to consider two cases, i.e., cached and non-cached. This means that instead of monitoring 256 cache lines, our attack only has to monitor a single cache line. In line with the Meltdown-US-L1 attack, this code snippet raises an exception that has to be handled, e.g., with fault handling, TSX, or fault suppression via speculation [35]. For the decoding, i.e., transferring the information encoded in the microarchitecture to an architectural state, any side channel can be used. For simplicity and in line with related work [35,54,49,57,6], we rely on Flush+Reload to recover the encoded information. In case of a recent access by the victim, the target address is stored in the L1D cache. Thus, to monitor further cache accesses, we need to remove the target address from the L1D cache. As the target memory address cannot be accessed, we need to rely on eviction. However, as the L1D cache is virtually-indexed, evicting from it is straightforward and can be achieved by accessing virtual memory addresses falling into the same cache line as the target address. Note that we only need to evict the target address when an access occurred, as the Meltdown attack itself does not cache the target address.

Attack Surface. We investigate the attack surface of using Meltdown-US-L1 as a side channel by analyzing which kernel pages are mapped in user space when KPTI is active. As Meltdown-US-L1 cannot be fixed via microcode on affected hardware, KPTI [14] is used as a software workaround on Meltdown-US-L1-affected CPUs. KPTI ensures that while an application runs in user space, no kernel page containing sensitive information is mapped into the address space. For this, KPTI relies on a second set of page tables [15]. However, while this works theoretically, x86 always requires some kernel pages to be mapped, even when running in user space. Luckily, the content of these pages, e.g., the IDT, can be chosen such that they do not contain secrets.

We investigate which pages are still mapped in userspace by iterating through the user page tables using the kernel module PTEditor [47]. For the user-page-table root, we set bit 11 of the physical address stored in the kernel `CR3` register [15]. We iterate through the mappings in the upper half of the address space for kernel addresses mapped in user space. We discover between 198 and 394 4 kB kernel pages mapped in user space, depending on the CPU. However, these pages can be classified into only 3 distinct ranges. The first range is the kernel entry. This range has been exploited for microarchitectural KASLR breaks [46,7,61]. The second range is used for descriptor tables, such as the interrupt-descriptor table or the global-descriptor table. Finally, the third range is within the range of the direct physical map [31], mapping 4 physical pages. One of these mappings is to the task state segment, which is also mapped directly. We cannot explain the reason for these remaining mappings, as the target is already mapped in user space. Still, this does at least not increase the attack surface. The most interesting target for using Meltdown-US-L1 as a side channel is the IDT (cf. Section 4).

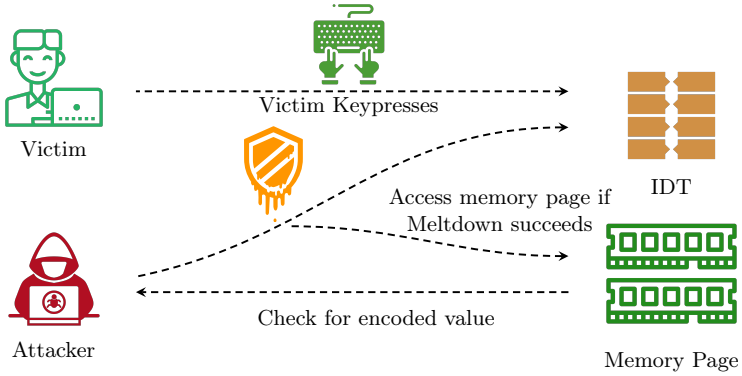

Fig. 2: Using LeakIDT to leak interrupts, such as keystrokes.

4 LeakIDT

In this section, we introduce LeakIDT, a side-channel attack that precisely detects when an attacker-chosen interrupt occurs. LeakIDT achieves that by observing the cache state of the IDT entries of the targeted interrupts.

Linux uses one IDT per core that always resides at the same location (cf. Section 2.3). This IDT is mapped in all processes, even with KPTI. Hence, our attack can target the IDT despite applied software-based Meltdown-US-L1 mitigations. Note that a different operating system could randomize the location of the IDT upon booting and thus harden the system against our attack.

Attack Overview. Figure 2 shows an overview of LeakIDT. We use Meltdown-US-L1 to read a specific IDT entry corresponding to a targeted interrupt. IDT entries are accessed—and thus cached in L1D—if the CPU core handles an interrupt. Hence, if the leakage of the entry is successful, we infer that the interrupt was triggered; otherwise, it was not. Consequently, with LeakIDT we know the timestamp when the interrupt occurred. Note that due to the CPU's hardware prefetchers the actual accuracy of our attack is reduced to blocks of 8 adjacent IDT entries. Further details on this are discussed later in this section. When detecting an interrupt, LeakIDT uses eviction to remove the targeted IDT entry from the L1D cache again. This is crucial for the attack as after every observed interrupt, the attacker must ensure that the IDT entry is removed from the cache as quickly as possible. Otherwise, subsequent accesses to that memory address, i.e., subsequent interrupts of the same type, cannot be detected.

Threat Model. Our attack requires a victim application that leaks information by having secret or data-dependent interrupts. Such a victim can, e.g., receive keystrokes [17,33], issue secret-dependant legacy syscalls [65], or communicate over the network [66]. Besides this, we assume a bug-free software containing no logical vulnerabilities. We further consider the attacker and victim both executing unprivileged native code on the same Meltdown-US-L1-affected

CPU. The attack does not assume any disabled mitigations, i.e., it works with state-of-the-art software-based Meltdown mitigations.

Implementation. For inferring the cache state of the IDT entry, we use the code from Listing 1. Note that each IDT entry is 16 bytes in size. Thus, there are 4 IDT entries per cache line that are all cached when an interrupt occurs. The exact offset of the IDT entry we are targeting with LeakIDT is irrelevant, as every interrupt corresponding to that entry caches the entire cache line. One should note that the granularity of our attack in a normal environment is restricted to blocks of 8 IDT entries. The reason is that upon receiving an interrupt, the CPU's adjacent cache-line prefetcher puts two adjacent cache lines, i.e., 8 adjacent IDT entries, into the L1D cache at once.

To detect the correct entry in the IDT, we template the IDT entries. First, we record the number of interrupts for every IDT entry over a fixed time window, e.g., 100 ms. Second, we repeat this recording step while inducing the interrupt in parallel. Depending on the type of interrupt, this can be done in soft- or hardware. Some interrupts can be triggered the same way the victim triggers the interrupt, e.g., sending a network packet for network interrupts. If this is not possible, e.g., for keyboard interrupts, an attacker can induce the same interrupt as a software interrupt, using the `int` instruction. If the difference in the number of interrupts correlates with the induced interrupts, the correct IDT entry is identified. As we do not require fine-grained measurements for this step, we can take the information exposed by the Linux interface, i.e., the file `/proc/interrupts`.

As discussed in Section 3, to ensure that LeakIDT can detect more than the first interrupt, the IDT entry has to be evicted again from the L1D cache. The cache replacement policy on our machines is Tree-PLRU [1], and the cache is virtually indexed using bits 6 to 11. Thus, we access memory addresses falling into the same L1D cache set by accessing pages at the same offset as IDT entry offset, which performs well enough for the attacks.

5 Evaluation

In this section, we evaluate the performance and reliability of LeakIDT. All experiments are executed on an Intel Core i7-6600U running Ubuntu 20.04 with Linux kernel 5.4.0. On a general level, LeakIDT allows observing the cache state of an inaccessible but mapped memory page. More precisely, we can distinguish between a memory address that is cached in the L1D cache and a memory address that is not cached in the L1D.

First, we evaluate how precisely we can distinguish between such two memory addresses. We mount our exploit on two memory addresses, one being cached in the L1D cache and one not being cached. Note that distinguishing between an address cached in L1D and not cached at all is enough for an attacker to mount side-channel attacks. Our tests show that for a memory address cached in L1D, we have a successful leak in 99.6 % of cases and no leakage in 100 % of cases for uncached memory addresses. We observe that for the uncached target byte, we only see the byte 0x0 encoded in our lookup array. This observation is in line

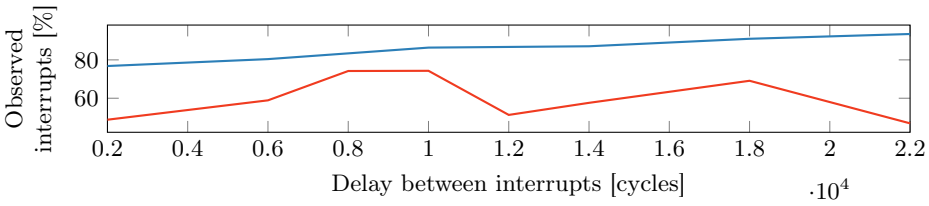

Fig. 3: Delay between interrupts and number of interrupts missed by LeakIDT (upper line) and Prime+Probe (lower line).

with previous work [35,62]. These results show that an attacker can reliably infer the cache state of the target kernel memory address by observing whether the Meltdown-US-L1 leakage exists.

Figure 3 shows how different delays between interrupts interfere with the observation rate of our attack, i.e., the number of interrupts successfully detected. More precisely, we trigger 10 000 interrupts with an artificial busy wait of n cycles between them. This allows us to measure the success rate of our attack when the victim triggers interrupts at a high frequency. We observe that if the interrupts are more closely spaced than 25 000 cycles, our detection rate decreases. We further observe that for interrupts happening at a slower rate, we have success rates of up to 99.5 %. Thus, attackers can exploit LeakIDT to reliably leak interrupts up until this frequency.

Comparison to Related Kernel Attacks. To the best of our knowledge, LeakIDT is the first cache-line-granular side-channel attack on the kernel. LeakIDT does not require read- or writable shared memory, which is typical for cache attacks [64,16,34], preventing their use on kernel memory. While there are also cache attacks not requiring shared memory [5,43,10,40], LeakIDT yields a better granularity as it allows targeting specific cache lines of the kernel. Additionally, cache attacks without shared memory often require knowledge of physical addresses to construct reliable and efficient eviction sets [53]. As we do not assume that knowledge in our threat model, we compare LeakIDT with Prime+Probe on the L1D, as this attack has the same threat model.

Not only does LeakIDT have a finer granularity, but it also outperforms Prime+Probe in terms of reliability. Figure 3 shows the number of interrupts missed by our Prime+Probe implementation. Note that our implementation only counts an interrupt if two probe steps show higher access timing. While this may not be optimal, it significantly reduces the number of false positives and shows the best performance during our evaluation. We suspect that the reason for this is that the probes execute fast enough to measure the activity on the IDT entry multiple times during the interrupt handling. To further compare the two side channels, we compare their performance in a more artifical scenario. We take 100 000 measurements for each attack while the victim accesses the targeted cache line 50 000 times per attack. Finally, we compare the results of our side-channel attacks to the ground truth of victim accesses. For LeakIDT, we get a

recall of 0.999 and a precision of 1.0, yielding an F-score of 0.999. For Prime+Probe on the L1D, we measure a recall of 1.0 and a precision of 0.834, yielding an F-score of 0.91.

6 Case Studies

In this section, we introduce 2 case studies demonstrating LeakIDT. Leveraging LeakIDT, we show that an attacker can spy on websites visited by a victim on the same system (cf. Section 6.1). Furthermore, we show that fine-grained timing measurements of interrupts leak information about the keystrokes entered by a user (cf. Section 6.2).

6.1 Website Fingerprinting

In this section, we use LeakIDT to detect which website a user opens by monitoring network interrupts. For this purpose, we perform the website fingerprinting attack on an Intel Xeon E3-1505M v5, with Ubuntu 20.04 and Linux kernel 5.4.0.

Threat Model. In line with previous work [65,25,52,29,18], we assume an unprivileged attacker with native code execution on the victim system. In contrast to these works, we do not rely on OS interfaces, as they are nowadays only available to privileged users. We assume the attacker application runs on the physical core that handles the network interrupts, which the unprivileged `pthread_setaffinity_np` Linux API can achieve.

Attack Overview. We do not assume prior knowledge of the IDT entry that the attacker needs to probe. Thus, the first step of the attack is to find the specific IDT entry that handles the network interrupts. To do that, we use LeakIDT on all IDT entries while introducing additional network traffic. For each entry, we record the number of accesses during a short fixed period, e.g., 1 second. Next, we repeat the measurement without generating additional network interrupts. As the network interrupts bring the specific IDT entries into the cache, entries with the most significant differences in the number of accesses are likely related to the network interrupts.

In line with previous work [66], we rely on a coarse-grained timer, e.g., `clock_gettime` or `setitimer`, to record the number of interrupts per 5 ms interval when a user opens a website. We then train a random forest classifier to fingerprint the opened website.

Results. We collect 100 interrupt traces for each of the Alexa 100 most-visited websites. Each trace collects the number of interrupts in a 5 ms interval 400 times (2 s in total). The dataset is split into a training set of 7000 and a test set of 3000 examples, and the `n_estimators` for the random forest classification are set to the default of 100. For the top 15 websites, we achieve a precision of 80 % and a recall of 81 %, as illustrated in the confusion matrix in Figure 4. For the top 100 websites, we achieve an overall precision of 55 % and a recall of 56 %. Note that a more precise timer, i.e., with a better accuracy than 5 ms would likely improve these results.

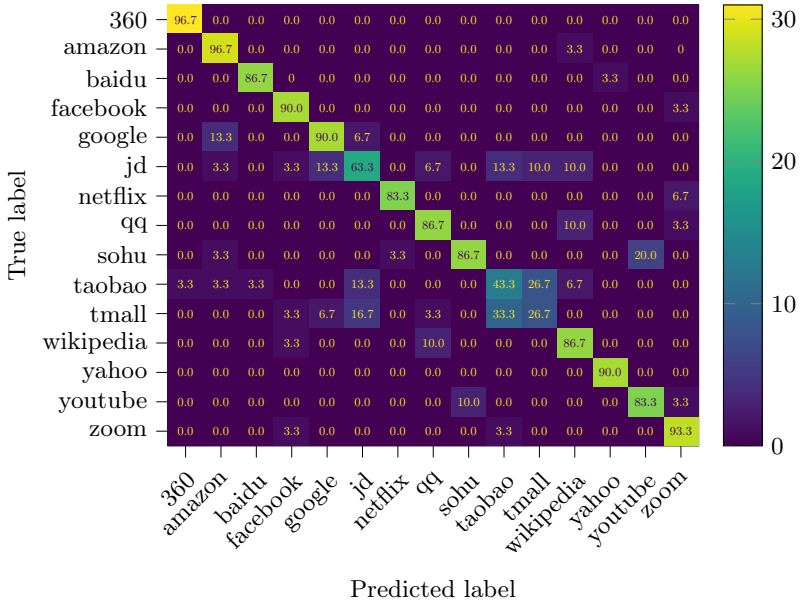

Fig. 4: The confusion matrix for the website classification. Given the Alexa top 15 websites, the trace is classified correctly with an overall probability of 80 %.

Comparison to Related Work. While Spreitzer et al. [52] report 89 % accuracy on 100 sites on Android, the attack requires the unprivileged interface for sampling data-usage statistics. Zhang et al. [66] report 71 % accuracy on 100 sites on Intel, relying on the new `umwait` instructions only available on the latest Intel microarchitectures. The interrupt attack by Lipp et al. [33] correctly classifies a website in 81.75 % of cases inside the browser when only looking at 10 websites. Lee et al. [29] exploit GPU vulnerabilities and report 69.4 % and 60.9 % with two different techniques on 100 sites randomly chosen from Alexa Top 1000.

6.2 Keystroke Timings via LeakIDT

In this section, we show that LeakIDT can be used for keystroke-timing attacks, as first discussed by Song et al. [51]. We show that LeakIDT reliably recovers keystroke timings on USB keyboards on an Intel Xeon E3-1505M v5, with Ubuntu 20.04 and Linux kernel 5.4.0.

Threat Model. We assume an unprivileged attacker with native code execution on a system vulnerable to LeakIDT. We further assume that the attacker application can be pinned to specific physical cores by unprivileged APIs.

Experiment Setup. In line with the first case study, we do not assume knowledge of the IDT entry. Thus, an attacker trying to locate the core responsible for handling keyboard interrupts can probe all interrupts on all cores for

Table 1: Results for the inter-keystroke timing attack.

Run	Noise	Recall	Precision	F-score	Delay (std dev.)
1	no	0.93	0.89	0.91	−323 µs (35.66 µs)
2	no	0.91	0.95	0.93	−334.5 µs (29.71 µs)
3	no	0.90	0.90	0.90	−324 µs (34.11 µs)
1	yes	0.89	0.87	0.88	−573 µs (64.25 µs)
2	yes	0.88	0.88	0.88	−568 µs (49.46 µs)
3	yes	0.86	0.86	0.86	−551 µs (56.28 µs)

a short and fixed time interval. Afterward, when the attacker knows that the victim is likely pressing keys, e.g., by checking for interactive applications in the list of running processes, the attacker can probe these interrupts again and check for significant differences. To optimize the measurements for this case study, the attacker pins the spy process on the sibling of the previously identified core.

We perform our experiments in two settings. In the first setting, a lab environment, the eXtensible Host Controller Interface (xHCI) interrupts are handled by an isolated core. In the second setting, a realistic environment, we boot the system without any preparations and simulate heavy system load with the stress utility (`stress -m 2 -c 2`). The kernel distributes the interrupts over the available 4 cores. xHCI interrupts share their core only with peripheral network interrupts in our experiments. These interrupts occur every 2 s.

We spawn two processes. The first one reads characters from `stdin` and logs microsecond timestamps of the keystrokes. This process can be spawned on any core and provides ground-truth data. The second process is pinned to the physical core handling xHCI interrupts. This process logs microsecond timestamps of leaked interrupts via LeakIDT.

We perform 3 runs of typing 200 random keys on the keyboard for both setups. In our case study, all inputs are entered by a single person. We record the timestamp traces of both processes. We then match every recorded interrupt timestamp to the nearest ground truth timestamp. Since xHCIs generate two interrupts for USB keyboards, i.e., key down and key up, we assume two captured interrupts per actual timestamp. Even though the difference between key down and key up events can improve the results of keystroke attacks [42], we choose to ignore their impact in this case study to focus on the concept. Any missing timestamp from the expected 2 interrupts for each actual timestamp is counted as a false negative. Any detected interrupt matching with more than one uniquely identifiable key-up and key-down event is counted as a false positive.

Results. Table 1 shows the results for the 3 runs for both setups. We calculate recall, precision, and F-score with the data acquired from matching recorded interrupts to ground truth timestamps. We measure the median and the standard deviation of the delay when we detect the interrupts, showing that we detect keystroke interrupts around half a microsecond before they can be read from `stdin` in the victim application. As expected, LeakIDT performs slightly

worse in the realistic setup compared to the isolated lab setup. In the isolated setup, we observe an F-score of 0.91, and for the realistic setup an F-score of 0.87. In comparison, the Android-based keystroke timing attacks from Schwarz et al. [48] achieve an F-score of 0.94 and 0.81. Similar attacks from Vila et al. [58] and Wang et al. [59] achieve a recall of 0.98 and 0.57, respectively. Thus, our results are comparable to previous work. Note that depending on the goal of an attacker, further steps are required for an end-to-end attack, such as machine-learning-based password recovery or user classification.

7 Mitigations

In this section, we propose a mitigation against LeakIDT. We evaluate the mitigation and show that it only introduces a minimal performance overhead.

Although the root cause of LeakIDT cannot be mitigated in software, we propose a software mitigation to prevent exploitation. The main idea is to ensure that the cache state of an IDT entry cannot be inferred by marking the IDT as uncachable, ensuring that the cache state is always the same.

Implementation. Linux uses a shared IDT across all CPU cores. This single IDT is allocated once by the OS and keeps its physical location until reboot. We rely on memory-type range registers (MTRRs) to mark the physical range of the IDT as uncachable. While the number of MTRRs is limited [20, Chapter 11.11], we only require a single MTRR due to the shared IDT. MTRRs have the advantage that the memory type defined by them cannot be overwritten.

Alternatively, if no MTRR can be spared, the IDT mapping can be marked as uncachable via the memory type in the corresponding page-table entry. Care has to be taken that this is done in every single user-space process, as well as in the kernel. This requires more changes to the kernel and introduces a startup overhead for every application. Thus, we opted for the MTRR-based approach, requiring only a minimal overhead at boot for the configuration and allowing the implementation as a kernel module.

Evaluation. We evaluate the security and performance of our approach. All evaluations are run on an Intel Xeon E3-1505M v5, with Ubuntu 20.04.1 and kernel 5.4.0. For the security evaluation, we mount LeakIDT with our active mitigation. As expected, we do not see any leakage. With the uncachable IDT, LeakIDT can never leak an entry of the IDT, preventing LeakIDT.

To evaluate the overhead of our mitigation, we execute benchmarks generating both high CPU loads and a large number of interrupts. We execute SPEC CPU 2017, which resembles generic real-world workloads, and additionally, Kraken and JetStream, two JavaScript benchmarks, to see the impact on web services. For the baseline, we run the benchmarks on the unmodified system. As marking the IDT as uncachable is implemented as a kernel module, we can run the benchmark on precisely the same kernel without even rebooting. Hence, with this setup, there should not be any other factors influencing the benchmark results. Table 2 shows the results of the SPEC CPU benchmark. The details of the JavaScript benchmarks can be found in Appendix A. On av-

Table 2: Performance of uncachable IDT on the SPEC CPU 2017 benchmark.

Benchmark	**SPEC Score**		**Overhead**
	Baseline	Uncachable	[%]
600.perlbench_s	1.88	1.88	0.00 %
602.gcc_s	1.11	1.11	0.00 %
605.mcf_s	1.91	1.91	0.00 %
620.omnetpp_s	1.50	1.52	+1.33 %
623.xalancbmk_s	1.55	1.58	+1.94 %
625.x264_s	1.54	1.54	0.00 %
631.deepsjeng_s	1.33	1.33	0.00 %
641.leela_s	1.20	1.20	0.00 %
648.exchange2_s	3.50	3.52	+0.57 %
657.xz_s	0.91	0.91	0.00 %
Average			+0.65 %

erage, we only measure a minimal performance overhead of 0.65 % with SPEC CPU 2017, 0.57 % with Kraken (cf. Table 3), and 0.32 % with JetStream. We further test the impact on two interrupt-heavy benchmarks. We execute the YCSB benchmark [9] to evaluate the overhead for databases. We test against a MongoDB instance and configure YCSB for 4 500 000 operations. We observe an increase in interrupts of 2326.29 %, i.e., 52 853.62 interrupts (on average over the 8 cores of the system), compared to the system idling for the same amount of time, i.e., 2326.29 interrupts. As these benchmarks have a shorter execution time than the previous ones, we repeat this measurement 10 times on the baseline system and 10 times on the same system with the applied mitigation, thus ensuring a stable result. We observe a median runtime of 155 155 ms with a standard deviation of 243.01 for the baseline system and a median runtime of 155 181.5 ms with a standard deviation of 286.99, i.e., an overhead of 0.02 %. To test the performance of a network-based key-value store, we evaluate the impact on a Memcached instance using the benchmarking framework mutilate [30]. We configure mutilate to execute 16 connections spanned over 8 threads. Table 4 shows the results. Hereby, we observe an increase in interrupts of 2630.30 %, i.e., 69 892.38 interrupts (on average over the 8 cores of the system), compared to the system idling for the same amount of time, i.e., 2559.88 interrupts. We execute this benchmark 10 times with and without the mitigation applied. We observe a slowdown of the receive rate, the transmission rate, and the QPS of 0.14 % each.

8 Discussion

In this section, we discuss Meltdown mitigations, their remaining leakage, and their applicability to other Meltdown variants, OS, and architectures.

Meltdown Mitigations. Gruss et al. [15] showed that unmapping the kernel when possible mitigates several side-channel attacks on it. This has be-

Table 3: Kraken benchmark results.

Test Case	Baseline	Uncacheable IDT	Overhead
ai	164.8 ms (+/- 6.0 %)	169.6 ms (+/- 6.0 %)	+2.91 %
astar	164.8 ms (+/- 6.0 %)	169.6 ms (+/- 6.0 %)	+2.91 %
audio	532.8 ms (+/- 2.6 %)	540.8 ms (+/- 2.1 %)	+1.50 %
beat-detection	141.2 ms (+/- 3.6 %)	141.7 ms (+/- 3.0 %)	+0.28 %
dft	115.8 ms (+/- 3.7 %)	117.9 ms (+/- 5.2 %)	+1.81 %
fft	125.2 ms (+/- 2.9 %)	125.8 ms (+/- 3.3 %)	+0.48 %
oscillator	150.6 ms (+/- 6.6 %)	155.5 ms (+/- 5.3 %)	+3.25 %
imaging	406.2 ms (+/- 2.1 %)	400.3 ms (+/- 2.4 %)	−1.45 %
gaussian-blur	158.3 ms (+/- 4.0 %)	154.0 ms (+/- 2.3 %)	−2.72 %
darkroom	80.1 ms (+/- 0.8 %)	79.6 ms (+/- 0.9 %)	−0.62 %
desaturate	167.8 ms (+/- 4.4 %)	166.7 ms (+/- 5.9 %)	−0.66 %
json	73.9 ms (+/- 7.0 %)	76.7 ms (+/- 5.0 %)	+3.79 %
parse-financial	37.0 ms (+/- 13.7 %)	37.7 ms (+/- 9.6 %)	+1.89 %
stringify-tinderbox	36.9 ms (+/- 2.7 %)	39.0 ms (+/- 4.5 %)	+5.69 %
stanford	288.3 ms (+/- 2.3 %)	287.0 ms (+/- 1.6 %)	−0.45 %
crypto-aes	73.9 ms (+/- 3.2 %)	74.1 ms (+/- 3.2 %)	+0.27 %
crypto-ccm	65.6 ms (+/- 4.1 %)	63.6 ms (+/- 2.4 %)	−3.05 %
crypto-pbkdf2	100.0 ms (+/- 1.9 %)	101.1 ms (+/- 1.7 %)	+1.10 %
crypto-sha256-iterative	48.8 ms (+/- 5.6 %)	48.2 ms (+/- 3.0 %)	−1.23 %
Total	1466.0 ms (+/- 0.9 %)	1474.4 ms (+/- 0.7 %)	+0.57 %

Table 4: Mutilate benchmark results.

Attribute	Baseline score	UC IDT score	Slowdown
QPS	176 238.35 (std: 1376.41)	175 987.5 (std: 1407.86)	0.14 %
RX	7 759 664 293 B (std: 60 596 022.33)	7 748 550 743.5 B (std: 62 033 777.92)	0.14 %
TX	1 208 593 212 B (std: 9 439 659.48)	1 206 927 213 B (std: 9 614 344.61)	0.14 %

come the state-of-the-art mitigation against Meltdown-US-L1 [35,14]. However, a limitation of the x86 architecture is that specific kernel structures, such as the IDT, must always be mapped. While related work used these mappings to break KASLR [46,7,61], such attacks can be prevented by using a different randomization offset for the pages that remain mapped. However, this would not prevent LeakIDT. The reason is that LeakIDT exploits the metadata of the data stored on kernel pages and not the content [35] or the location [46,7,61].

We show that uncachable memory eliminates the remaining leakage of KPTI. Restricting the uncachable memory to the IDT ensures that the performance impact is minimal. Hence, combining two incomplete mitigations for orthogonal problems hardens a system against side-channel attacks.

Ideally, vulnerabilities are mitigated in the hardware. Still, despite hardware fixes, Canella et al. [7] showed that they leak metadata about the mapping of a

virtual address. While the leakage is much more limited than in our attack, it also shows that side-channel leakage can be overlooked when designing mitigations.

Applicability to other Meltdown-Type Attacks. Our attack is not limited to attacking the kernel. While we convert Meltdown-US-L1 into a side channel, the same technique can also be applied to other Meltdown variants. For example, on CPUs affected by Foreshadow [54], our technique could be used to implement an Evict+Reload-style attack on Intel SGX enclaves. For this, only the Meltdown attack has to be replaced with the related Foreshadow attack. However, in contrast to the Meltdown-US-L1 mitigations in the OS, the Foreshadow mitigations for SGX entirely prevent Foreshadow. Hence, an enclave that can be attacked with Foreshadow as a side channel could also be attacked directly with Foreshadow. We leave it to future work to investigate whether other Meltdown-type attacks, such as RIDL [57], ZombieLoad [49], or Fallout [6], could also be transformed into practical side-channel attacks.

Other OS and Architectures. The underlying effects exploited in this paper are OS-agnostic. While this paper targets Linux, we do not require any Linux-specific functionality. For example, while the interrupt numbers differ on Windows, the mechanism is still the same. The IDT is also mapped, as this is required by the x86 architecture, enabling LeakIDT.

As LeakIDT fundamentally relies on the Meltdown-US-L1 CPU vulnerability, it does not apply to Meltdown-unaffected CPUs. Hence, AMD and most Arm CPUs are not affected [35]. While there are Arm CPUs affected by Meltdown-US-L1 [35], the interrupt handling is different, which would require adapting LeakIDT to work with the IDT-equivalent, the Interrupt Vector Table (IVT).

9 Conclusion

We showed that Meltdown cannot only act as a transient-execution attack but can also be exploited as a side-channel attack by adding another layer of indirection, despite active software mitigations. We presented LeakIDT, a side-channel attack that allows an attacker to monitor mapped kernel pages with cache-line granularity, enabling attackers to spy on chosen interrupts. We showed that attackers can exploit this primitive to spy on websites visited by a user. We analyzed that this fine-granular information leakage also reveals valuable insights into the typing behavior of a user by allowing to spy on their keystroke timings. Hence, we conclude that even though Meltdown-US-L1 is considered no longer exploitable, it still threatens the security of modern systems.

Acknowledgment

We want to thank our anonymous reviewers for their comments and suggestions. We also want to thank Leon Trampert and Niklas Flentje for providing their help with running the experiments. This work was partly supported by the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS).

References

1. A. Abel and J. Reineke, "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures," in *ASPLOS*, 2019.
2. O. Acıiçmez and W. Schindler, "A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL," in *CT-RSA 2008*, 2008.
3. O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.
4. S. Bhattacharya and D. Mukhopadhyay, "Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms," *Cryptology ePrint Archive, Report 2015/621*, 2015.
5. S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks," in *USENIX Security Symposium*, 2020.
6. C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *CCS*, 2019.
7. C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *AsiaCCS*, 2020.
8. C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019, extended classification tree and PoCs at https://transient.fail/.
9. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM symposium on Cloud computing*, 2010.
10. C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *USENIX Security Symposium*, 2017.
11. Federal Office for Information Security, "Minimum requirements of evaluating side-channel attack resistance of rsa, dsa, and diffie-hellman key exchange implementations," 2013. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_BSI_guidelines_SCA_RSA_V1_0_e_pdf.pdf
12. T. Gleixner, "x86/kpti: Kernel Page Table Isolation (was KAISER)," 2017. [Online]. Available: https://lkml.org/lkml/2017/12/4/709
13. B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS*, 2020.
14. D. Gruss, D. Hansen, and B. Gregg, "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer," *USENIX ;login*, 2018.
15. D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.
16. D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.
17. D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.
18. B. Gulmezoglu, A. Zankl, T. Eisenbarth, and B. Sunar, "PerfWeb: How to violate web privacy with hardware performance events," in *European Symposium on Research in Computer Security*, 2017.

19. Intel, "Intel 64 and IA-32 Architectures Software Developer′s Manual, Volume 1: Basic Architecture," vol. 253665, 2016.
20. Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2019.
21. Intel, "Intel-SA-00233 Microarchitectural Data Sampling Advisory," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html
22. Intel, "Affected Processors: Transient Execution Attacks," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html
23. Intel Corporation, "Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html
24. ——, "Refined Speculative Execution Terminology," 2020. [Online]. Available: https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology
25. S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *S&P'12*, 2012.
26. P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.
27. P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.
28. E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.
29. S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting gpu vulnerabilities," in *S&P*, 2014.
30. J. Leverich, "Mutilate: high-performance memcached load generator," 2014. [Online]. Available: https://github.com/leverich/mutilate
31. Linux, "Complete virtual memory map with 4-level page tables," 2019. [Online]. Available: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
32. M. Lipp, D. Gruss, and M. Schwarz, "AMD Prefetch Attacks through Power and Time," in *USENIX Security*, 2022.
33. M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C.-m.-t.-n. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *ESORICS*, 2017.
34. M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.
35. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.
36. G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.
37. D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis," in *USENIX Security Symposium*, 2020.
38. J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.

39. Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS*, 2015.
40. C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.
41. P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.
42. S. Pinet, J. C. Ziegler, and F.-X. Alario, "Typing is writing: Linguistic properties modulate typing execution," *Psychon Bull Rev*, vol. 23, no. 6, pp. 1898–1906, April 2016.
43. A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.
44. H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *USENIX Security*, 2021.
45. H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, 2021.
46. M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," *arXiv:1905.05725*, 2019.
47. M. Schwarz, M. Lipp, and C. Canella, "misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8," 2018. [Online]. Available: https://github.com/misc0110/PTEditor
48. M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks," in *NDSS*, 2018.
49. M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.
50. M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss, "Speculative Dereferencing of Registers: Reviving Foreshadow," in *FC*, 2021.
51. D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.
52. R. Spreitzer, S. Griesmayr, T. Korak, and S. Mangard, "Exploiting data-usage statistics for website fingerprinting attacks on android," in *WiSec*, 2016.
53. E. Tromer, D. A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, July 2010.
54. J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.
55. J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *S&P*, 2020.
56. J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *CCS*, 2018.
57. S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, 2019.
58. P. Vila and B. Köpf, "Loophole: Timing Attacks on Shared Event Loops in Chrome," in *USENIX Security Symposium*, 2017.
59. H. Wang, T. T.-T. Lai, and R. Roy Choudhury, "Mole: Motion leaks through smartwatch sensors," in *Proceedings of the international conference on mobile computing and networking*, 2015.

60. Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86," in *USENIX Security Symposium*, 2022.
61. D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated Discovery of Microarchitectural Side Channels," in *USENIX Security*, 2021.
62. Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities," in *NDSS*, 2020.
63. L. Yan, Y. Guo, X. Chen, and H. Mei, "A Study on Power Side Channels on Mobile Devices," in *Symposium on Internetware*, 2015.
64. Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.
65. K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, 2009.
66. R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security*, 2023.

A JavaScript Benchmark Results

Table 5 shows the impact of our mitigations measured using the JetStream JavaScript benchmark. The total overhead is 0.23 %.

Table 5: JetStream benchmark results.

Test Case	Baseline	UC IDT	Overhead
3d-cube-SP	142.229	142.101	−0.09 %
3d-raytrace-SP	120.536	130.642	+8.38 %
acorn-wtb	14.267	13.714	−3.88 %
ai-astar	335.52	330.153	−1.60 %
Air	186.75	195.861	+4.88 %
async-fs	73.995	68.635	−7.24 %
Babylon	180.118	169.655	−5.81 %
babylon-wtb	14.468	16.173	+11.78 %
base64-SP	218.656	236.194	+8.02 %
Basic	195.428	168.823	−13.61 %
bomb-workers	20.826	17.721	−14.91 %
Box2D	120.487	144.29	+19.76 %
cdjs	32.632	28.819	−11.68 %
chai-wtb	42.27	42.068	−0.48 %
coffeescript-wtb	21.964	18.937	−13.78 %
crypto	279.665	341.319	+22.05 %
crypto-aes-SP	179.095	146.226	−18.35 %
crypto-md5-SP	113.45	106.569	−6.07 %
delta-blue	226.869	176.9	−22.03 %
earley-boyer	199.003	201.622	+1.32 %
espree-wtb	15.742	14.141	−10.17 %
first-inspector-code-load	102.469	99.755	−2.65 %
FlightPlanner	176.477	176.196	−0.16 %
float-mm.c	7.474	7.497	+0.31 %
gaussian-blur	225.208	230.45	+2.33 %
gbemu	62.156	57.95	−6.77 %
gcc-loops-wasm	21.568	22.449	+4.08 %
hash-map	94.658	124.756	+31.80 %
HashSet-wasm	27.422	29.125	+6.21 %
jshint-wtb	21.484	20.658	−3.84 %
json-parse-inspector	111.78	108.445	−2.98 %
json-stringify-inspector	122.419	120.712	−1.39 %
lebab-wtb	24.557	24.599	+0.17 %
mandreel	32.562	32.546	−0.05 %
ML	13.853	13.274	−4.18 %
multi-inspector-code-load	109.236	92.512	−15.31 %
n-body-SP	466.978	459.006	−1.71 %
navier-stokes	400.438	409.595	+2.29 %
octane-code-load	502.996	460.544	−8.44 %
octane-zlib	14.938	15.063	+0.84 %
OfflineAssembler	36.527	33.54	−8.18 %
pdfjs	75.934	78.712	+3.66 %
prepack-wtb	20.974	20.541	−2.06 %
quicksort-wasm	215.166	217.597	+1.13 %
raytrace	202.931	222.117	+9.45 %
regex-dna-SP	255.332	249.183	−2.41 %
regexp	281.028	279.361	−0.59 %
richards	196.298	189.976	−3.22 %
richards-wasm	37.949	33.478	−11.78 %
segmentation	11.835	12.609	+6.54 %
splay	88.279	85.402	−3.26 %
stanford-crypto-aes	173.872	188.774	+8.57 %
stanford-crypto-pbkdf2	213.472	258.864	+21.26 %
stanford-crypto-sha256	322.22	317.002	−1.62 %
string-unpack-code-SP	168.69	141.399	−16.18 %
tagcloud-SP	77.685	99.776	+28.44 %
tsf-wasm	42.163	67.481	+60.05 %
typescript	6.67	6.598	−1.08 %
uglify-js-wtb	12.796	13.636	+6.56 %
UniPoker	196.529	195.398	−0.58 %
WSL	0.411	0.405	−1.46 %
Total	7909.404	7927.544	+0.23 %