Reviving Meltdown 3a

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. Since the initial discovery of Meltdown and Spectre in 2017, different variants of these attacks have been discovered. One often overlooked variant is Meltdown 3a, also known as Meltdown-CPL-REG. Even though Meltdown-CPL-REG was initially discovered in 2018, the available information regarding the vulnerability is still sparse.
In this paper, we analyze Meltdown-CPL-REG on 19 different CPUs from different vendors using an automated tool. We observe that the impact is more diverse than documented and differs from CPU to CPU. Surprisingly, while the newest Intel CPUs do not seem affected by Meltdown-CPL-REG, the newest available AMD CPUs (Zen3+) are still affected by the vulnerability. Furthermore, given our attack primitive CounterLeak, we show that besides up-to-date patches, Meltdown-CPL-REG can still be exploited as we reenable performance-counter-based attacks on cryptographic algorithms, break KASLR, and mount Spectre attacks. Although Meltdown-CPL-REG is not as powerful as other transient-execution attacks, its attack surface should not be underestimated.

1 Introduction

Microarchitectural side-channel attacks have been known for several decades [31]. These attacks exploit the side effects of CPU implementations to infer metadata about actual data being processed by the CPU. Well-known examples of microarchitectural side-channel attacks include cache attacks, e.g., Flush+Reload [67] or Prime+Probe [47], which have been used to leak cryptographic secrets [2, 67, 37] or violate the privacy of users, e.g., by spying on user input [44, 19, 34, 55]. Another example of side-channel attacks are attacks based on the CPUs performance counters [58, 12, 8]. However, these attacks are considered mitigated as access to performance counters is restricted on modern CPUs [12].

In 2017, transient execution attacks were first discovered in the form of Meltdown [36] and Spectre [30]. Shortly afterward, a variety of transient execution attacks were discovered [10, 56, 61, 65, 40, 32, 51, 50]. One attack that is often considered less powerful than other variations, and thus easily overshadowed by the discovery of other variants, is Meltdown 3a [10, 6], later on, referred to as Meltdown-CPL-REG in the extended transient-execution attack classification by Canella et al. [10]. Meltdown-CPL-REG allows an unprivileged attacker to leak

the content of system registers restricted to privileged access. After the discovery of the attack, CPU vendors reacted with microcode updates to fix the vulnerabilities [25, 6]. More precisely, CPU vendors fixed the vulnerability for system registers containing confidential information, such as model-specific registers.

In this paper, we show that Meltdown-CPL-REG exposes a more complex attack surface than originally thought, which allows an attacker to exploit it, even 5 years after the initial discovery of the attack. Although the Meltdown variant itself is known, there is no systematic analysis yet. Thus, we introduce RegCheck, an automated tool to test x86 CPUs for various Meltdown-CPL-REG variants. Our analysis using RegCheck reveals two main insights. First, CPUs that are vulnerable to Meltdown-CPL-REG do not show the same leakage for all system registers. Instead, the analysis shows that different CPUs expose leakage of different system registers. Hence, the category Meltdown-CPL-REG is too coarse-grained to determine if a CPU is affected. The official tables published by Intel [25] comment only on the leakage of the `rdmsr` instruction. Nevertheless, RegCheck shows that for some of these CPUs, there is at least one system register that can be leaked. Second, the fact that a CPU is unaffected by the original Meltdown attack, i.e., Meltdown-US-L1 [36, 10], does not imply that the CPU is also unaffected by Meltdown-CPL-REG as we observe leakage until the newest tested AMD CPUs. Our analysis shows that while Meltdown-CPL-REG was mitigated using microcode updates for system registers containing confidential data, Meltdown-CPL-REG is still possible on modern CPUs for those privileged registers that are not considered confidential, including registers containing only metadata about a program, such as performance counters.

Based on these observations, we introduce the attack primitive CounterLeak. CounterLeak allows unprivileged attackers to read performance counters, thereby leaking performance monitoring metadata about applications running on a system. This shows that the state-of-the-art Meltdown-CPL-REG mitigations are insufficient for protecting against side-channel leakage. In our proof-of-concept attack, we read the performance counters to leak meta information about applications. We encode transiently-read data in the form of a Spectre attack with 66.7 bit/s, but with a generic encoding gadget. We also break the security mitigation Kernel Address Space Layout Randomization (KASLR) by leaking meta information of the page-table walker when accessing potential kernel pages. Furthermore, CounterLeak re-enables attacks that rely on performance counters [7, 4]. These attacks are considered mitigated because the required performance interface was made privileged. Our attack extracts an RSA key from a square-and-multiply implementation based on MbedTLS. We demonstrate a full key recovery of a 2048-bit key within 15 min. We also show that CounterLeak can be used to break the Zigzagger branch-shadowing mitigation [33]. While all these attacks require that the underlying system has performance counters enabled, this is the case for various performance-counter-based defenses that were proposed [71, 29, 46, 69, 11, 43, 42, 63, 64, 72]. Thus, we stress that when designing defense tools, it is crucial to evaluate the additional attack surface introduced by these tools.

To summarize, we make the following contributions:

1. We analyze 19 CPUs of different vendors using an automated tool, showing that Meltdown-CPL-REG was never fully mitigated and can still be exploited. The analysis tool is open-source and can be found on GitHub[1].
2. We use our side channel for a novel Spectre attack using performance counters and to bypass KASLR based on the performance characteristics of the page-table walker.
3. We re-enable attacks on cryptographic libraries.

Outline. Section 2 provides background. Section 3 discusses our analysis of Meltdown-CPL-REG across different Intel and AMD CPUs. Section 4 presents the CounterLeak primitive, and Section 5 evaluates the primitive. Section 6 shows 4 case studies based on the attack primitive. Section 7 discusses mitigations to prevent the exploitation of Meltdown-CPL-REG and CounterLeak. Section 8 discusses related work and the generalization of our insights. Section 9 concludes.

Responsible Disclosure. We disclosed our findings to Intel on February 15, 2023 and AMD on February 16, 2023. While both vendors got back to us, neither plan to roll out mitigations for the new findings.

2 Background

In this section, we provide the background for this paper. We introduce performance counters as we attack this interface in the remainder of the paper. We introduce side channels and transient-execution attacks, as these concepts are crucial for the understanding of our attack implementation.

2.1 Performance Counters

Modern CPUs expose performance counters to help developers analyze and benchmark their programs. Performance counters keep track of different microarchitectural events, such as the number of issued micro-operations or the number of evicted cache lines from the L1D cache. Performance counters are programmed to record a specific event. The current count of the event can be read using the x86 instruction `rdpmc`. The privilege level needed to execute `rdpmc` can be configured by the operating system. For example, Linux exposes this configuration via the file `/sys/devices/cpu/rdpmc`. In the past, unprivileged access to performance counters was exploited to mount side-channel attacks and break KASLR [12, 58]. Thus, modern operating systems, such as Debian 11, Ubuntu 20.04, or Fedora 35, disallow the access to the performance monitoring interface.

2.2 Side Channels

The term side channel refers to a meta information leaking from a system that can be used to reason about the actual inaccessible data being processed by

[1] `https://github.com/cispa/regcheck`

the system. In CPU microarchitectures, this meta information occurs in various forms, including power usage [35], access timings [47, 18], and contention [5, 16, 45]. An attack exploiting observable meta information is referred to as a side-channel attack. Microarchitectural software-based side-channel attacks (in the remainder of this paper just referred to as "side-channel attacks") have been demonstrated against cryptographic algorithms and libraries [47, 38, 67, 8], to spy on users [55], and to break security boundaries [13, 17, 12]. Over the last few decades, researchers have demonstrated side-channel attacks based on several microarchitectural components, such as the CPU caches [47, 67, 18, 48], the execution units [5, 52], or the component's power consumption [35].

2.3 Transient-Execution Attacks

Transient-execution attacks exploit performance optimizations of the microarchitecture. They are split into two major categories, namely Meltdown-type and Spectre-type attacks, based on the type of performance optimization they exploit [10, 27]. While Spectre-type attacks exploit branch predictors, Meltdown-type attacks exploit faulting instructions for which the processor continues to execute depending instructions. These instructions can compute with the values of the faulting instructions until the fault is recognized by the CPU and the instruction stream is rolled back to before the faulting instruction. These instructions that were executed but never architecturally visible because of the roll-back, are called transient instructions [10, 27]. One Meltdown-type attack that is typically considered less critical, is Meltdown-CPL-REG (initially called Meltdown 3a) [6, 23, 10]. Meltdown-CPL-REG allows an unprivileged attacker to leak the content of privileged system registers. Hereby, the attacker reads the system registers via a designated instruction such as `rdmsr` and encodes the content into a microarchitectural element before the roll-back occurs. Afterward, the attacker can decode this information using a side-channel attack, thus leaking the system register's content. To mitigate the impact of Meltdown-CPL-REG, CPU vendors provide microcode updates for affected systems [25, 6].

3 Analysis of Meltdown-CPL-REG

For Meltdown-CPL-REG, microcode prevents the leakage of system registers containing sensitive values. However, other registers containing meta-data about applications can still be leaked, enabling another source of side-channel leakage. We present the first systematic analysis of Meltdown-CPL-REG [6, 23, 10] to analyze the remaining attack surface after applying state-of-the-art microcode patches. To systematically analyze CPUs, we design RegCheck to test a CPU for different Meltdown-CPL-REG variants automatically. Our analysis of 19 systems leads to two main insights. First, if a system is vulnerable to Meltdown-CPL-REG, this does not mean that *all* system registers are affected. Second, even fully patched recent CPUs unaffected by the original Meltdown attack (Meltdown-US-L1) [36] can be vulnerable to Meltdown-CPL-REG.

Design and Implementation. Our prototype of RegCheck is developed for Intel and AMD CPUs running Linux. Note that the same approach can be ported to other architectures, e.g., to support Arm CPUs, as this is purely an engineering task. RegCheck tests a list of different system registers that are either only accessible for privileged users or can be configured to only allow privileged access. The list is based on Intel's list of affected registers [23]. We provide a complete list of analyzed system registers in Table 1. The inner workings of RegCheck can be broken down into two steps:

First, RegCheck changes the kernel parameters to a consistent state for the measurements. More precisely, one CPU core is isolated using the `isolcpus` kernel parameter, and unprivileged access to `rdfsbase` and `rdgsbase` is disabled using the `nofsgsbase` kernel parameter. Similarly, the access to further system registers which are not permanently restricted to privileged access, e.g., performance counters (cf. Section 2.1), is configured to prevent unprivileged access to these registers before testing. After applying these settings, RegCheck executes on the isolated CPU core to reduce the system noise for its measurements. Next, for each system register, RegCheck tries to reason about its exploitability. To do so, RegCheck tries to exploit Meltdown-CPL-REG and encode 8 bits of the system register into a lookup array. The encoding is done by transiently accessing the corresponding index of the array, e.g., if the leaked bits form the value 7, then an access to `array[7 * N]` is performed. The resulting fault can either be suppressed or handled. For RegCheck we choose to handle the fault using a signal handler as this approach is portable to all modern CPUs. Our implementation varies N from 1024 to 4096 bytes to find a good tradeoff between the size of memory pages needed to encode the values while still preventing different accessing from either directly going into the same cache line or prefetching other array entries. Note that we choose to encode 8 bits instead of only 1 bit to distinguish actual leakage from system noise better. After encoding these bits, the tool checks whether a transient access to any index has taken place by iterating over the array and performing Flush+Reload, i.e., timing the memory access to each array index. If RegCheck succeeds at leaking the target system register multiple times, it flags it as vulnerable. We test our tool on Intel and AMD CPUs from different generations. All tests use the latest microcode available in the Ubuntu repositories. For further details on the specific microcode version used we refer the reader to Table 2.

Affected Registers. The main insight from our analysis is that not all privileged registers are affected in the same way by Meltdown-CPL-REG. This is especially interesting because Intel's list of CPUs affected by certain vulnerabilities [25] (accessed May 2023) only lists CPUs where the `rdmsr` instruction can be exploited by Meltdown-CPL-REG. However, our results in Table 2 show that some CPUs that Intel flags as unaffected by the Meltdown-CPL-REG `rdmsr` leakage can still be exploited to leak the contents of other system registers, such as the performance counters using `rdpmc`. This, for example, is the case for the Intel Celeron J4005 and the Intel Celeron N3350. The results in Table 2 show that the instruction `rdfsbase` leaks on 8 out of 14 CPUs affected by Meltdown-

Table 1: System registers and their access instructions tested by RegCheck.

Access Instruction	Details
`rdpmc`	Reads the specified Performance counter
`rdtsc`	Reads the CPU timestamp counter
`rdtscp`	Reads the CPU timestamp counter
`mov CRx`	Loads the Control registers 0 - 8
`mov DRx`	Loads the Debug registers 0 - 7
`rdfsbase`	Retrieves segment selector of the FS segment base register
`rdgsbase`	Retrieves segment selector of the GS segment base register
`rdmsr`	Model Specific Registers
`str`	Loads the segment selector of the Task register
`sldt`	Loads the segment selector from the Local Descriptor Table register
`sidt`	Loads the segment selector from the Interrupt Descriptor Table register
`sgdt`	Loads the segment selector from the Global Descriptor Table register
`smsw`	Loads the Machine status word

CPL-REG. The CPU timestamp counter accessed via `rdtsc` or `rdtscp` leaks on 2 out of 14 affected CPUs. Performance counter leak on 3 of the affected CPUs via `rdpmc`. A possible explanation for these different leakage rates could be that for executing `rdpmc`, the CPU has to decode an argument of the instruction, i.e., the index of the access performance counter stored in `RCX`, while for `rdtsc`, `rdtscp`, and `rdfsbase` all required information to fetch the requested data is available, leading to a potentially simpler execution path. Nevertheless, the CPUs where `rdpmc` is vulnerable do not show a superset of the vulnerable instructions compared to the other systems. Even though these systems show vulnerable `rdpmc` implementations, we could not verify further leakage.

Affected CPUs. Our second insight is that the fact that CPUs are vulnerable to Meltdown-US-L1 is not related to whether a CPU is also vulnerable to Meltdown-CPL-REG, as shown in Table 2. In other words, we can leak from system registers of CPUs that are affected by Meltdown-US-L1 and of CPUs not affected by Meltdown-US-L1. This is especially surprising for recently released CPUs, such as the Ryzen 9 6900HX. We observe that the tested Intel CPUs from Alder Lake onward do not show leakage, while newer AMD CPUs do.

RegCheck Limitations. The current proof-of-concept implementation of our tool RegCheck comes with different limitations. We do not check for the leakage of `swapgs` as previous work has already analyzed this instruction and its leakage potential [39]. We neither check the `xgetbv` instruction. The reason for the latter is that to prevent unprivileged access to `xgetbv`, RegCheck needs to set the `OSXSAVE` bit of `CR4`, which crashes the tested OS. A detailed list of the analyzed system registers is shown in Table 1.

Table 2 flags `rdtsc` and `rdtscp` for certain instances with an "U" (short for "unverified"). On these systems, we observed leakage from the system registers, but could not verify that the leakage stems from the CPU timestamp counter. The reason for this is that RegCheck uses a counting thread as a timer for analyzing the instructions `rdtsc` and `rdtscp`. However, this timer does not work

Table 2: CPUs tested by RegCheck for Meltdown-CPL-REG. "U" means we could not verify if an actual timestamp is leaked. "ZF" means that only the value 0 is returned transiently. Additionally, we annotate machines that are vulnerable to the original Meltdown attack.

CPU	μcode	μarch	Release	MD-US	Leaking Instructions
Intel Core i5-2520M	0x2f	Sandy Bridge	2011	Yes	`rdtsc`, `rdtscp`
Intel Core i5-3230M	0x21	Ivy Bridge	2013	Yes	`rdtsc`, `rdtscp`, `sldt`
Intel Core i3-4160T	0x28	Haswell	2014	Yes	`rdfsbase`, `rdgsbase`
Intel Core i3-5010U	0x2f	Broadwell	2015	Yes	`rdfsbase`, `rdgsbase`, `rdtsc` (U), `rdtscp` (U)
Intel Atom x5-Z8350	0x411	Cherry Trail	2016	Yes	`rdpmc`
Intel Celeron N3550	0x28	Apollo Lake	2016	No	`rdpmc`
Intel Celeron J4005	0x3c	Gemini Lake	2017	Yes	`rdpmc`
Intel Core i3-7100T	0xf0	Kaby Lake	2017	Yes	`rdfsbase`, `rdgsbase`
Intel Core i3-1005G1	0xb2	Ice Lake	2019	No	–
Intel Core i7-10510U	0xf0	Comet Lake	2019	No	`rdfsbase`, `rdgsbase`
Intel Core i7-1185G7	0xa4	Tiger Lake	2020	No	–
Intel Celeron N4500	0x240000023	Jasper Lake	2021	No	`rdfsbase` (ZF), `rdgsbase` (ZF), `sldt` (ZF)
Intel Core i9-12900K	0x22	Alder Lake	2021	No	–
Intel Atom x6425E	0x17	Elkhart Lake	2021	No	–
AMD GX-415GA	0x700010f	Jaguar	2013	No	–
AMD Ryzen 5 2500U	0x810100b	Zen	2017	No	`rdfsbase`, `rdgsbase`
AMD Ryzen 5 3550H	0x8108102	Zen+	2019	No	`rdfsbase`, `rdgsbase`
AMD Epyc 7252	0x8301055	Rome	2019	No	`rdfsbase`, `rdgsbase`, `str` (ZF)
AMD Ryzen 9 6900HX	0xa404102	Zen 3+	2022	No	`rdfsbase`, `rdgsbase`

reliably on CPUs not supporting hyperthreading, as the counting and attacker thread yield a more accurate timer when both threads execute on co-located hyperthreads. Table 2 also has system registers flagged with "ZF" (short for "zero forwarding"). For these registers, an access always returns the value 0 instead of the actual value. While such behavior intuitively sounds invulnerable, instructions forwarding zero values already led to microarchitectural attacks [60, 9].

4 Attack Primitive

In this section, we introduce our attack primitive CounterLeak. CounterLeak exploits Meltdown-CPL-REG to leak performance-counter values using `rdpmc` to infer side-channel information about program executions.

4.1 Threat Model

We assume an unprivileged attacker with native code execution. We further assume bug-free victim software, e.g., the absence of memory corruption or logical vulnerabilities. However, our attacker model relies on side-channel vulnerabilities, i.e., we assume secret-dependent control or data flow in the victim application. Even though our attacks are, in theory, mountable from inside virtual machines, we did not explicitly test this, and attackers could only target victims inside their own virtual machine and not the hypervisor or other virtual machines. While this weakens the attack surface, intra-VM attacks are still a realistic scenario, e.g., in container-based environments. We target only Intel and

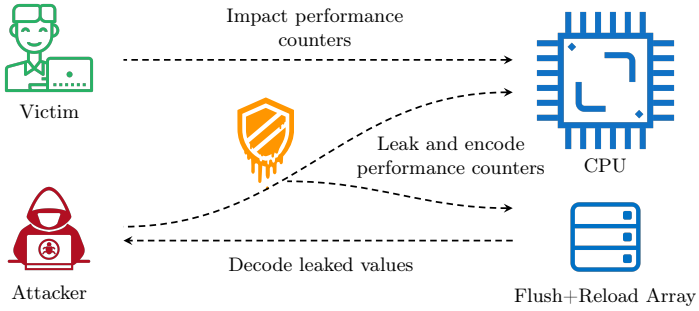

Fig. 1: Meltdown-CPL-REG leaking system registers, such as performance counters.

AMD CPUs in this work. Note that Meltdown-CPL-REG is also exploitable on Arm [6] but we consider further architectures out of scope for the experiments conducted in this paper and only discuss them in Section 8.

4.2 CounterLeak

The CounterLeak attack primitive relies on Meltdown-CPL-REG. We use Meltdown-CPL-REG to infer side-channel information about a victim program. Based on our systematic analysis using RegCheck, and the publicly-available information regarding Meltdown-CPL-REG by Intel [23], we build our attack primitive on top of `rdpmc`. `rdpmc` provides a generic but privileged interface to performance counters. Access to these performance counters leaks information about the program execution that can be exploited for side-channel attacks [58, 8, 12].

Attack Overview. CounterLeak relies on Meltdown-CPL-REG to leak the content of a performance counter. We assume that the system already has a performance counter programmed. This is the case if the system uses performance counters for attack detection, as suggested by previous work [20, 46, 28, 69]. For example, Cloudflare relies on performance counters to detect Spectre attacks [62]. An attacker leaks the performance-counter values by encoding the transiently-read return value of the `rdpmc` instruction into the microarchitecture and recovers it using a side channel.

Implementation. In line with previous Meltdown-type attacks [36, 30, 56, 59, 61, 41, 10], we use the CPU cache to encode the transiently-leaked values and Flush+Reload as the covert channel to make the values architecturally visible. We support leakage of 1 to 4 bytes per `rdpmc` invocation by encoding each byte into the cache state of an array consisting of 256 pages. The more data is encoded into the microarchitecture, the better the resolution of the underlying performance counter value. However, this also leads to a slower decoding phase, as more Flush+Reload attacks are required. For leaking a single byte, at most 256 Flush+Reload attacks are necessary, while for leaking 4 bytes, at most 1024 Flush+Reload attacks are necessary. We evaluate this trade-off in Section 5.

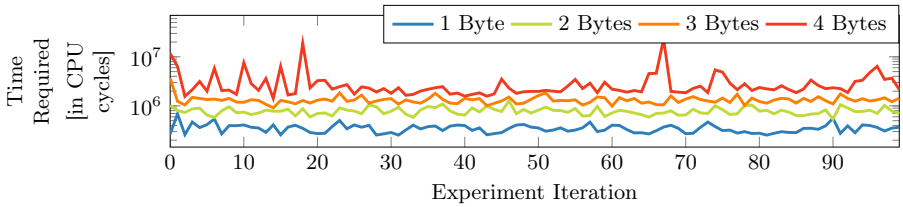

Fig. 2: CounterLeak: CPU cycles needed to leak, i.e., access, encode, and decode, n bytes of a performance counter by attacking `rdpmc`. The y-axis shows the CPU cycles required for each repetition of the experiment (x-axis).

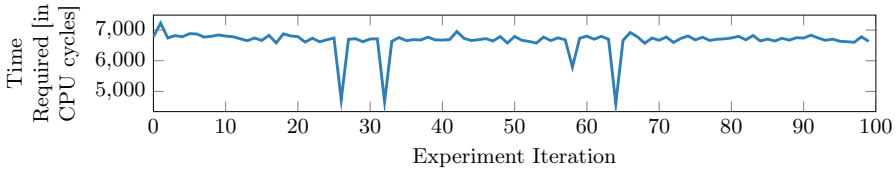

Fig. 3: CounterLeak: CPU cycles needed to transiently encode 4 bytes of the CPU timestamp counter. The y-axis shows the CPU cycles required for each repetition of the experiment (x-axis).

5 Evaluation

In this section, we evaluate the attack primitive CounterLeak which is based on Meltdown-CPL-REG. All evaluations use our proof-of-concept implementation on an Intel Celeron J4005 running Ubuntu 20.04 with Linux kernel 5.4.0.

The most important property in our evaluation is the temporal resolution of CounterLeak, i.e., the time between two measurements. This property reflects how fine-grained the information can be leaked by the exploit. We evaluate the time it takes to leak n bytes of a system register. This measurement directly gives us the temporal resolution of the attack. We observe that the implementation leaks 1 byte of a system register in, on average, 348 257 cycles ($n = 100$). Figure 2 summarizes the time an attacker needs to leak the content of a performance counter when leaking n bytes within one transient window. We emphasize that this is a good indication of the theoretical performance of this attack, as an attacker can likely mount exploits by only leaking parts of the system register. We also require only partial leakage for our attacks discussed in Section 6. Note that the temporal resolution mostly affects the execution time of an attack but does not prevent an attack. An attacker can often compensate for a lower temporal resolution by averaging over repeated measurements [35].

Still, whereas our complete attack primitive takes millions of CPU cycles for one iteration (cf. Figure 2), the actual time spent encoding multiple bytes of a system register is significantly shorter. While the time needed to leak n bytes of a performance counter, i.e., the attack's temporal resolution, is important for repeated measurements, another critical metric is the time that an attacker

needs to encode a value in the CPU cache. This metric is especially important for event-driven attack scenario, i.e., whenever the attacker wants to take a measurement after a certain event has happened. To evaluate the time it takes to encode a value, we record the time needed to encode the value of the timestamp register over 100 runs. Figure 3 shows the results. We observe that the average time between the faulting access and the first subsequent attacker-controlled instruction when encoding 4 bytes simultaneously is 6655 cycles. Whereas the effective blindspot of our attack is higher, this time yields the offset between an event triggering a measurement in the attacker code and the measurement itself.

6 Case Studies

In this section, we introduce 4 case studies demonstrating CounterLeak. We demonstrate a Spectre proof-of-concept (PoC) (Section 6.1) and break KASLR by monitoring the behavior of page walks (Section 6.2). To demonstrate that our side channels re-enable mitigated attacks, we leak a 2048-bit RSA private key from a square-and-multiply implementation found in MbedTLS using CounterLeak (Section 6.3). Lastly, we show that we can break the branch-shadowing mitigation proposed by Lee et al. [33] using CounterLeak (Section 6.4).

6.1 Spectre with CounterLeak

In this case study, we demonstrate a Spectre-type attack [30, 10] with our CounterLeak primitive to leak otherwise inaccessible data. We build a Spectre-PHT [30, 10] PoC with a performance counter as covert channel.

Target Performance Counter. We target a performance counter that tracks speculative events [49], such as `CYCLES_DIV_BUSY.ALL` and assume that it is either activated or can be enabled by the attacker. Note that depending on the victim's code, the discussed attack can also be mounted with a different performance counter. The only requirement is that the accessed secret can be encoded in branches that can be distinguished based on any performance counter.

Attack Overview. We attack a Spectre gadget of the form

```
1 if (i >= 0 && i < array_size) {
2     int tmp = (array[i] >> offset);
3     if ((tmp & 1))  x / y;
4 }
```

The attacker controls the variables `i` and `offset`. Note that even though the inner `if` branch is only doing an operation that should not result in any state change, it still affects related performance counters and hence suffices to enable our attack. The attacker starts by mistraining the outer `if` branch such that its subsequent execution is misspeculated to be taken. The simplest way to achieve this is by in-place mistraining [10], i.e., executing the branch multiple times with `i` being a valid offset for the array. As a baseline, the attacker leaks the value of the performance counter `CYCLES_DIV_BUSY.ALL` using CounterLeak.

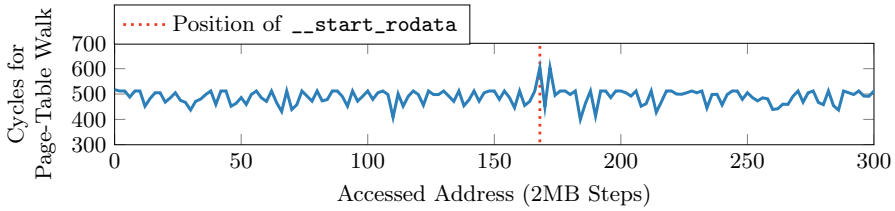

Fig. 4: The leaked values of `DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M` when iterating over the potential locations where the Linux kernel could be mapped. The page-table walk needs longer when the address is the actual start of the kernel, i.e., the position of the kernel symbol `__start_rodata`.

This performance counter keeps track of the number of cycles the CPU's divider units are used. The attacker executes the victim function with an index `i` that is outside the bounds of the array and corresponds to the targeted memory address. Afterward, the attacker again leaks the performance counter of `CYCLES_DIV_BUSY.ALL` using CounterLeak and subtracts the previously leaked value. As the divider is only used when the inner `if` branch is (speculatively) taken, the delta is slightly higher if the transiently-accessed bit is '1'.

Results. We measure each bit 50 times and set a threshold on the median to distinguish between '1' and '0' bits based on the value of the performance counter. Our PoC achieves a leakage rate of 66.7 bit/s with an accuracy of 99.6 %. While not the fastest covert channel, we argue that it is still fast enough to pose a threat when such an attack is mounted.

Comparison to Similar Attacks. Our attack only relies on a control flow that is distinguishable by observing performance counters. Common covert channels used in Spectre-type attacks require cache accesses to encode data from transient execution [30, 32, 21, 40]. Finding such code paths that can be exploited by Spectre-type attacks, also referred to as Spectre gadgets, is a challenging task. While our attack is limited to a CPU vulnerable to CounterLeak and providing a usable performance counter, it can use both traditional Spectre gadgets and novel types of gadgets. Hence, with the combination of Spectre and CounterLeak, the number of potential gadgets increases.

6.2 Breaking KASLR with CounterLeak

We demonstrate that unprivileged access to performance counters breaks Kernel Space Address Layout Randomization (KASLR). KASLR randomizes the base address of the operating system kernel upon booting. As precise knowledge of the memory layout is a requirement for many attacks, KASLR adds an additional barrier that attackers have to overcome for a successful kernel exploit. We show that we can derandomize the location of the Linux kernel on an Intel Celeron N3350 running Ubuntu 22.04 with Linux kernel 5.15.0 and thus bypass KASLR.

Target Performance Counter. We target a performance counter influenced by page-table walks, such as `DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M`

and assume that is already programmed or can be programmed by the attacker. A scenario in which this is the case is if the system is protected using the approach of Wang et al. [63].

Attack Overview. For derandomizing the kernel location, we rely on the property that non-present pages are not stored in the TLB [9]. Thus, a memory load request to a non-present page always leads to a page-table walk, whereas a memory load request to a present page leads to a TLB hit, resulting in no page-table walk if the page was recently accessed. The attack iterates over each potential location of the kernel and accesses it. The resulting fault caused by the access is suppressed using speculative execution, TSX transactions, or fault handling. For each memory access, the attacker leaks the performance counter `DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M`, or an alternative one correlating to the number of or the cycles spent for page-table walks, using CounterLeak. Based on the leaked value, the attacker can observe whether a memory page is present and was recently accessed. The first page of the kernel's `.rodata` section is frequently accessed. Thus, the first address showing an abnormal timing difference is the location of the kernel symbol `__start_rodata`. Note that a more advanced version of this attack can also be used to actively monitor the access to kernel memory pages, similar to the work of Schwarz et al. [53].

Results. Figure 4 shows the cycle difference iterating over the kernel address space. The kernel location is easily distinguishable from non-present pages due to the change in cycles spent for page table walks. We tested our KASLR break on an Intel Celeron J4005 running Ubuntu 20.04 with Linux kernel 5.4.0 observing a success rate of 98 % (n=100) and a median execution time of 4.7 s.

6.3 Attacking RSA with CounterLeak

In this case study, we attack the RSA implementation based on the MbedTLS version 1.3.10 running on an Intel Celeron J4005 with Ubuntu 20.04 and Linux kernel 5.4.0. This MbedTLS version implements RSA by using a window-based square-and-multiply algorithm. We configure the window size to 1. Previous work [37] showed that all window sizes are vulnerable if window size 1 is vulnerable. While such square-and-multiply implementations are known to be vulnerable to side-channel attacks, we choose this target as it is a common target for related attacks [54, 22, 14, 33, 66]. Hence, we ease comparison with other side-channel attacks.

Target Performance Counter. We target the performance counter `BR_INST_RETIRED.NEAR_TAKEN` and assume that it is either already programmed or can be programmed by the attacker. This performance counter keeps track of the number of taken near-branch instructions. An example for a realistic scenario in which this performance counter would be programmed is a system protected by the rootkit detection of Singh et al. [57].

Attack Overview. The victim application consists of a branch only taken when the currently-processed secret bit is '1'. Thus, the secret bit correlates with the number of branches taken. The attacker gains oracle access to the signing routine of the application to sign arbitrary messages. We assume that the

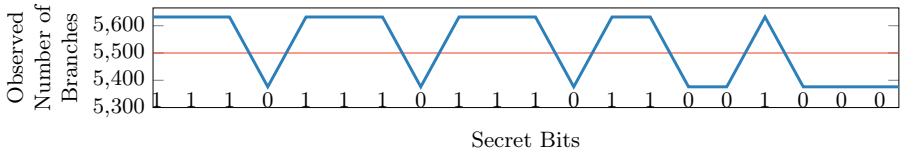

Fig. 5: The leaked value of the performance counter `BR_INST_RETIRED.NEAR_TAKEN` and its correlation to the secret bits of the exponent.

attacker and victim are synchronized, i.e., the attacker either knows when the victim processes each iteration of the exponentiation loop, or the attacker can influence this by, e.g., interrupting the victim. During the execution of the victim, the attacker repeatedly leaks the value of the performance counter and, thereby, the number of branches taken. The attacker leaks the performance counter once per key bit. Afterward, the attacker stores the delta of two consecutive performance counter leaks, i.e., the approximation of the victim's taken branches for the processing of a specific secret bit. The attacker repeats this procedure for the decryption of 10 000 different messages, averaging out the noise of branches taken by CounterLeak itself and the unrelated branches of the victim application.

Results. By averaging over 10 000 traces, we extract a clear indication of the secret bits. Figure 5 visualizes the correlation between the number of branches taken and the secret bits. Using a simple threshold, we recover 99.9 % of the 2048-bit RSA keys ($n = 10$) in around 15 min. Compared to previous work, there are both faster attacks requiring fewer encryptions [3, 37] and attacks requiring a similar number of decryptions or more time to execute [68, 70]. We conclude that CounterLeak yields a strong primitive for leaking secrets from, for example, cryptographic implementations.

6.4 Breaking Zigzagger with CounterLeak

In this case study, we explore how CounterLeak breaks the Zigzagger branch-shadowing mitigation. Branch-shadowing attacks exploit the shared branch history between processes, allowing attackers to reason about the direction of a branch. For example, Lee et al. [33] demonstrate that a branch-shadowing attack can leak confidential data from Intel SGX enclaves. To prevent branch-shadowing attacks, Lee et al. [33] proposed a software mitigation called Zigzagger. Zigzagger replaces a set of branches with a single indirect branch. Thus, the attacker can only infer whether the branch was executed but cannot infer the branch direction anymore. To compute the address of the indirect jump, additional conditional-move instructions are used. In line with Gerlach et al. [15], we exploit the number of retired instructions to break the Zigzagger mitigation. While Gerlach et al. used an architectural interface to this information, we show that we can recover the same information using CounterLeak. This information allows an attacker again to distinguish the branches taken by the victim.

Target Performance Counter. We target the `INSTR_RETIRED` performance counter that is either already programmed or can be programmed by the attacker. A realistic scenario for this would be if the defense approach of Wang et al. [63] is in use on the system.

Attack Overview. The victim process contains secret-dependent branches and is hardened against branch-shadowing attacks using Zigzagger [33]. The attacker leaks the `INSTR_RETIRED` performance counter before and after the Zigzagger-hardened victim executes. The delta between these measurements yields the number of retired instructions. The attacker correlates this number with a baseline measurement for all branches.

Results. For the case study, we use an Intel Celeron J4005 running Ubuntu 20.04 with Linux kernel 5.4.0. For each of the 3 different possible arguments of the sample function, there is a unique number of retired instructions after the Zigzagger modification was applied. Hence, by observing the number of retired instructions, an attacker can directly infer the arguments. We observe a success rate of 100 % using 10 000 recorded measurements.

7 Countermeasures

In this section, we discuss countermeasures against CounterLeak and Meltdown-CPL-REG. The fundamental problem is that an unprivileged attacker can transiently access the metadata of an application in the form of performance counters. The exploited vulnerability is rooted deep inside the CPU. As the information stem from a CPU register, no software is involved. Nevertheless, operating systems can still defend against the impact of the attack whereas the victim application itself can be hardened against the attack.

Firmware. Several CPUs received microcode updates to prevent the leakage of system registers [1]. While CPU vendors do not disclose internals of these updates, it is likely that a similar patch can also mitigate the remaining leakage. Thus, the most efficient and effective mitigation is likely via microcode updates.

Kernel. CounterLeak fundamentally relies on performance counters that are either already programmed or that can be programmed by an attacker-accessible API. A common scenario for this are performance-counter-based detection approaches [71, 29, 46, 69, 11, 43, 42, 63, 64, 72]. As the absence of programmed or programmable performance counter prevents CounterLeak, a carefully designed system that does not use performance counters at all or only in the absense of untrusted parties and code can also prevent the exploitation of CounterLeak. As performance counters and their programming requires kernel privileges, the kernel could, in theory, completely prevent the programming of performance counters. However, this decision comes with the drawback that it would break existing software like the performance-counter-based detection approaches or monitoring utilities such as `perf`. In contrast, an operating system can prevent attacks on KASLR without breaking existing software. Canella et al. [9] proposed mapping dummy pages in the kernel such that all kernel addresses are mapped. Consequently, an attacker cannot infer the real location of the kernel.

Userspace Software. As CounterLeak is a side-channel attack, it is fundamentally limited to leaking data from an application with secret-dependent branches or data-flow edges. However, an application can generally be implemented without any secret-dependent accesses [26]. Applications implemented in such a way are not susceptible to CounterLeak. Especially for cryptographic algorithms, such implementations are state-of-the-art.

8 Discussion

In this section, we discuss related work. Furthermore, we show how the presented attack primitive behaves on different operating systems and architectures. As the building blocks of CounterLeak are OS-agnostic and also exist on other architectures, we assume that similar attacks are also possible there.

8.1 Related Work

In 2018, Intel and Arm disclosed the vulnerability and assigned it CVE-2018-3640 [24, 6]. While Intel released a security advisory and added a new category to their list of CPUs affected by vulnerabilities [24, 25], Arm added a section about the vulnerability in their Cache-Speculation Side-Channel whitepaper [6]. Our work builds on this initial disclosure by analyzing the leakage of different system registers on 19 CPUs with applied vendor mitigations. We further demonstrate that it is still possible to exploit Meltdown-CPL-REG in different scenarios.

While we focus our work on Meltdown-CPL-REG, Canella et al. [10] analyzed the landscape of transient-execution attacks with a broader focus. Furthermore, they first introduced the split into Meltdown- and Spectre-type attacks. In contrast, our work focuses on the specific variant Meltdown-CPL-REG and analyzes further details about it, including how widespread the issue itself is.

Attacks exploiting performance counters have been shown when the interface was accessible to unprivileged users. In 2008, Uhsadel et al. [58] first exploited performance counters to leak information about the CPU caches. With information similar to a cache attack, they showed that the information can be exploited to recover confidential values from a victim program. They also demonstrated their attack on an OpenSSL AES implementation. Bhattacharya et al. [8] further demonstrated that performance counters expose even more information than just the cache state and thus allow reasoning about the branch-predictor state. Their work discusses an exploit on a square-and-multiply implementation of RSA using the Montgomery-ladder algorithm. Since then, the access to performance counters is privileged by default, preventing these attacks on modern systems [12]. Dixon et al. [12] further stresses the importance of disabling unprivileged access to performance counters by showing that it allows derandomizing the kernel location. Gerlach et al. [15] exploit the unprivileged access to performance counters on RISC-V CPUs to break KASLR, leak the presence of inaccessible files, and detect interrupts. Our work mainly differs from these previous ones by demonstrating these and similar attacks on modern systems where performance-counter access is restricted to privileged users.

8.2 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. CounterLeak interacts with the hardware directly without requiring any OS support. If any application legitimately enables performance counters, they can be leaked.

CounterLeak requires systems that are vulnerable to Meltdown-CPL-REG. While Meltdown-CPL-REG was also shown on Arm CPUs [6], we leave it for future work to systematically analyze Arm CPUs for their Meltdown-CPL-REG attack surface. Nevertheless, as all strict requirements for CounterLeak are also given on Arm CPUs, we suspect that the issue also affects these systems.

9 Conclusion

In this paper, we analyzed the attack surface of Meltdown-CPL-REG. For this, we developed an automated approach using RegCheck (open-sourced on GitHub) to analyze 19 Intel and AMD CPUs based on different microarchitectures. In our analysis, we observe that the privileged system registers that can be leaked by Meltdown-CPL-REG differ from CPU to CPU. Furthermore, we observe that the FS and GS segment base registers can be leaked even on recent AMD CPUs (Zen 3+). We further show that our attack primitive CounterLeak can exploit side-channel information by leaking the values of performance counters using Meltdown-CPL-REG. We demonstrated CounterLeak in 4 different case studies. We showed that the primitive allows us to break KASLR by monitoring the page-table walker and can break the Zigzagger branch-shadowing mitigation [33]. Additionally, we demonstrated the applicability of CounterLeak as a flexible covert channel for Spectre attacks and leaked a 2048 bit RSA key from a square-and-multiply implementation in MbedTLS, verifying that our primitive reenables previously mitigated attacks. In conclusion, our work shows that Meltdown-CPL-REG should not be underestimated and still poses a threat to modern and fully patched 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. “Rogue system register read,” 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/rogue-system-register-read.html

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, "Yet Another MicroArchitecutral Attack: Exploiting I-cache," in *AS-PLOS*, 2007.
4. O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.
5. A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port Contention for Fun and Profit," in *S&P*, 2018.
6. ARM, "Cache Speculation Side-channels," 2020, version 2.5.
7. S. Bhattacharya, C.-m.-t.-n. Maurice, S. Bhasin, and D. Mukhopadhyay, "Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls," *Cryptology ePrint Archive, Report 2017/968*, 2017.
8. 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.
9. C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *AsiaCCS*, 2020.
10. 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/.
11. M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," ePrint 2015/1034, 2015.
12. L. Dixon, "Breaking KASLR with perf," 2017. [Online]. Available: https://blog.lizzie.io/kaslr-and-perf.html
13. U. Frisk, "Windows 10 KASLR Recovery with TSX," 2016. [Online]. Available: http://blog.frizk.net/2016/11/windows-10-kaslr-recovery-with-tsx.html
14. C. P. García, S. Ul Hassan, N. Tuveri, I. Gridin, A. C. Aldaya, and B. B. Brumley, "Certified side channels," in *USENIX Security Symposium*, 2020.
15. L. Gerlach, D. Weber, R. Zhang, and M. Schwarz, "A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs," in *S&P*, 2023.
16. B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS*, 2020.
17. D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.
18. D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.
19. D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.
20. N. Herath and A. Fogh, "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security," in *Black Hat Briefings*, 2015.
21. L. Hetterich and M. Schwarz, "Branch Different - Spectre Attacks on Apple Silicon," in *DIMVA*, 2022.
22. T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX," in *CHES*, 2020.

23. Intel, "Instructions affected by rogue system register read," 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/instructions-affected-rogue-system-register-read.html
24. ——, "Intel-SA-00115 Q2 2018 Speculative Execution Side Channel Update," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html
25. 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
26. 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
27. ——, "Refined Speculative Execution Terminology," 2020. [Online]. Available: https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology
28. G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Stopping microarchitectural attacks before execution," ePrint 2016/1196, 2017.
29. ——, "Mascat: Preventing microarchitectural attacks before distribution," in *CODASPY*, 2018.
30. 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.
31. P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.
32. E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.
33. S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security Symposium*, 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, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2020.
36. 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.
37. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.
38. X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM CSUR*, 2021.
39. A. Lutas and D. Lutas, "Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction," in *BlackHat Europe*, 2019.
40. G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.
41. D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis," in *USENIX Security Symposium*, 2020.

42. M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters," in *HASP*, 2018.
43. M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "WHISPER: A Tool for Run-time Detection of Side-Channel Attacks," *IEEE Access*, 2020.
44. 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.
45. R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical," in *USENIX Security Symposium*, 2021.
46. M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS*, 2016.
47. C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.
48. A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.
49. P. Qiu, Y. Lyu, H. Wang, D. Wang, C. Liu, Q. Gao, C. Wang, R. Sun, and G. Qu, "Pmuspill: The counters in performance monitor unit that leak sgx-protected secrets," *arXiv:2207.11689*, 2022.
50. 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.
51. H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, 2021.
52. T. Rokicki, C. Maurice, and M. Schwarz, "CPU Port Contention Without SMT," in *ESORICS*, 2022.
53. M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," *arXiv:1905.05725*, 2019.
54. M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.
55. 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.
56. 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.
57. B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *AsiaCCS*, 2017.
58. L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'08)*., 2008.
59. 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.
60. 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.
61. 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.
62. K. Varda, "Dynamic process isolation: Research by cloudflare and tu graz," 2021. [Online]. Available: https://blog.cloudflare.com/spectre-research-with-tu-graz/

63. H. Wang, H. Sayadi, A. Sasan, S. Rafatirad, and H. Homayoun, "Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks," in *ICCAD*, 2020.
64. H. Wang, H. Sayadi, A. Sasan, S. Rafatirad, T. Mohsenin, and H. Homayoun, "Comprehensive Evaluation of Machine Learning Countermeasures for Detecting Microarchitectural Side-Channel Attacks," in *GLSVLSI*, 2020.
65. O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," 2018. [Online]. Available: https://foreshadowattack.eu/foreshadow-NG.pdf
66. Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially Analyzing Side-channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves," in *CCS*, 2017.
67. Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.
68. Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *JCEN*, 2017.
69. T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.
70. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *CCS*, 2012.
71. Y. Zhang and M. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *CCS*, 2013.
72. Z. Zhang, X. Zhang, Q. Li, K. Sun, Y. Zhang, S. Liu, Y. Liu, and X. Li, "See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer," in *AsiaCCS*, 2021.