(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels

Ruiyi Zhang
CISPA Helmholtz Center for Information Security

Taehyun Kim
Independent

Daniel Weber
CISPA Helmholtz Center for Information Security

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

In the last years, there has been a rapid increase in microarchitectural attacks, exploiting side effects of various parts of the CPU. Most of them have in common that they rely on timing differences, requiring an architectural high-resolution timer to make microarchitectural states visible to an attacker.

In this paper, we present a new primitive that converts microarchitectural states into architectural states without relying on time measurements. We exploit the unprivileged idle-loop optimization instructions `umonitor` and `umwait` introduced with the new Intel microarchitectures (Tremont and Alder Lake). Although not documented, these instructions provide architectural feedback about the transient usage of a specified memory region. In three case studies, we show the versatility of our primitive. First, with Spectral, we present a way of enabling transient-execution attacks to leak bits architecturally with up to 200 kbit/s without requiring any architectural timer. Second, we show traditional side-channel attacks without relying on an architectural timer. Finally, we demonstrate that when augmented with a coarse-grained timer, we can also mount interrupt-timing attacks, allowing us to, e.g., detect which website a user opens. Our case studies highlight that the boundary between architecture and microarchitecture becomes more and more blurry, leading to new attack variants and complicating effective countermeasures.

1 Introduction

Microarchitectural attacks are a serious threat to the security of modern systems. These attacks exploit properties of the implementation of CPUs. Attacks leaking metadata have been shown on data and instruction caches [21], but also on other microarchitectural elements [1, 9, 18, 21, 49, 71, 98]. Transient-execution attacks, such as Spectre [45] or Meltdown [53], even leak data instead of only metadata.

The majority of these attacks use subtle timing differences to infer the state of a microarchitectural element [98]. Hence, they require an architectural high-resolution timer to measure these differences. A native-code attacker typically has access to such a high-resolution timer, e.g., the CPU's time stamp counter. However, on ARM, timers are often privileged [51], and on AMD, the resolution is limited on many CPUs [52]. Hypervisors and operating systems can also trap (and possibly emulate) this instruction [26, 42, 57, 95]. As the time-stamp counter is also used in malware, detection approaches can rely on that to detect malicious software [67].

Previous work showed that even in the absence of high-resolution timers, timers can be built by an attacker [22, 81]. Such timers, implemented as counting threads, require concurrent, uninterrupted execution, and shared memory. Still, while they have been used for microarchitectural attacks [17, 22, 45, 52, 77], they are typically less accurate and noisier than native high-resolution timers [17, 94].

A second property many side channels have in common is the *blind spot*. If the event to observe happens between a measurement and the end of a potential reset sequence, it cannot be detected [92]. The length of these blind spots is often in the range of multiple hundred to thousand CPU cycles [72]. Hence, we ask the following research questions:

Can we replace timing measurements with an architecturally-defined interface to leak side-channel information? Can such an interface also reduce or eliminate the blind spot of existing side channels?

In this paper, we introduce a novel primitive to enable side-channel attacks without architectural timers. With the Tremont and subsequently also the Alder Lake microarchitecture, Intel introduced a new unprivileged instruction pair for optimizing idle loops: `umonitor` and `umwait` [35]. These instructions let the CPU enter a sleep state from which it wakes up again if a pre-defined memory range is modified. They are similar to AMD's `monitorx` and `mwaitx` instructions as well as the privileged variants `monitor` and `mwait`. We exploit undocumented properties of these instructions to transform microarchitectural states into architectural states without measuring time explicitly. Specifically, we show that these instructions can be triggered by memory writes, and also via other instructions, such as cache-maintenance instructions

or transient writes. Moreover, although the feature set of the unprivileged `umwait` and the privileged `mwait` is documented to be different, we show that they behave nearly identical, by reverse engineering an undocumented timeout function of the `mwait` instruction.

Based on our reverse engineering and analysis of these instructions, we demonstrate cross-core covert channel without an architectural timer. In contrast to previous work [16,18], we do not rely on disabled ISA extensions [60] or time windows for the transmission. We only use the architectural output of the instructions combined with a self-clocking encoding.

While previous work showed that Spectre attacks can be mounted with coarse-grained or remote timers [31, 83, 87], they all require an architecturally accessible timer. Based on our covert channel, we demonstrate that Spectre attacks [45] can be mounted on Intel's newest CPUs without requiring any timing measurement. We refer to Spectre attacks based on our conversion method from microarchitectural to architectural states as Spectral (Spectre with architectural leakage) attacks. Furthermore, we show that Spectre Attacks are theoretically also possible with Prime+Abort [16]. However, Prime+Abort does no longer work on recent Intel CPUs, and we discuss several advantages of Spectral over Prime+Abort, such as its robustness in high-noise environments and its accuracy in regards to false-positive measurements. We show that Spectral reliably leaks data with a simple bit-wise Spectre gadget [10, 31,55,82] and without an architectural timer. Our unoptimized proof-of-concept implementation leaks up to 200 kbit/s in a lab environment, and on average 56.5 kbit/s on a default Linux installation. In general, this technique applies to all transient-execution attacks, providing a deterministic and fast primitive to convert microarchitectural to architectural states.

To show the versatility of our techniques, we demonstrate a primitive to distinguish cache hits from cache misses without an architectural timer. By automatically calibrating a padding sequence to the internal timeout of the `mwait` instruction, we build an architectural classifier that distinguishes short-running from long-running events. With that, we can, e.g., detect if memory loads hit or miss the cache. To demonstrate the stability of this attack, we use the well-known AES T-table attack on OpenSSL 1.0.1e, a de-facto benchmark for such attacks [21, 23, 51, 72]. We show that in contrast to other cache attacks [16, 70, 72], our primitive is extremely resistant to false positives caused by memory pressure on the system.

Finally, we show that the susceptibility to interrupts of the `mwait` instructions can also be exploited. We show that while ARM does not have a full-fledged `mwait` instruction, the `wfi` instruction shares this property with the `mwait` instruction, i.e., it can also observe interrupts. This property allows monitoring interrupts to, e.g., infer inter-keystroke timings [15, 50, 79]. In contrast to previous work, we do not require OS interfaces [15, 101] or architecturally accessible high-resolution timers [79]. Moreover, as our primitive immediately reports an interrupt, it reduces the blind spot, providing a high resolution. As a case study, we use interrupt monitoring for network interrupts, detecting opened websites.

Our results show that the line between architecture and microarchitecture becomes fuzzier with the introduction of new ISA extensions. Instructions such as the unprivileged `umonitor` and `umwait` can be abused to bridge the gap between microarchitectural and architectural states. This property is also dangerous for defenses. Assumptions that certain attacks, such as Spectre attacks, require an architectural timer no longer hold [59]. Moreover, we show that our primitives are not affected by typical system noise, and cannot be detected using any known dynamic detection method. We stress that cloud providers should be aware of the risks posed by new ISA extensions before deploying them in their environments.

To summarize, we make the following contributions:

1. We reverse engineer undocumented properties of the `monitor`- and `mwait`-instruction family that help convert microarchitectural into architectural states.
2. With Spectral, we show that fast Spectre attacks are possible without *any* architectural timing primitive, leaking up to 200 kbit/s.
3. We introduce a threshold measurement primitive to distinguish short-running from long-running events, e.g., to distinguish cache hits from misses without an architecturally accessible timer.
4. We show a precise interrupt-monitoring attack, e.g., to detect the website a victim opens.

Outline. Section 2 provides background. Section 3 presents the reverse engineering of the memory-monitoring functions on Intel and AMD. Section 4 introduces the attack primitives built on top of the undocumented features of these instructions. Section 5 shows a timing-less covert channel, and Section 6 demonstrates three case studies using our attack primitives. Section 7 proposes countermeasures. Section 8 discusses limitations and future work. Section 9 concludes.

Responsible Disclosure. We responsibly disclosed our findings to Intel, AMD, and ARM. All vendors acknowledged our side-channel attacks.

2 Background

2.1 Caches and Cache Attacks

For modern CPUs, memory accesses are the bottleneck when executing instructions. Hence, to reduce the latency when accessing recently used data, a CPU stores a copy of this data in a small but fast buffer, a so-called cache. Cache attacks exploit the different execution times for accessing data stored in the cache or main memory. There are multiple variants of cache attacks [76] but the best-known attacks are Flush+Reload [100] and Prime+Probe [70]. Flush+Reload [100] relies on shared memory between the attacker and the victim, as well as on the cache-maintenance instruction `clflush`. Prime+Probe [70] does not require shared memory and observes eviction of at-

tacker data caused by cache activity of the victim. Flush+Reload and Prime+Probe have been used for attacks on cryptographic algorithms [4, 8, 20, 28, 37, 43, 51, 70, 99]. A well-known attack is the cache attack on the AES T-table implementation [39], which is often used to compare the efficiency of attacks [16, 23, 72, 93] or defenses [19, 24]. In addition to attacks on cryptographic implementations, Flush+Reload has also been used to spy on user behavior [25, 51, 96, 103] and as the covert channel in transient-execution attacks [11, 45, 53, 80, 90, 91, 94].

Except for Prime+Abort [16], all cache attacks have in common that they require a method to measure time differences with a high resolution. However, Prime+Abort relies on Intel TSX, which only existed on selected Intel CPUs from the 6th to the 9th generation, and is now disabled on all currently available Intel CPUs with the newest microcode updates [60]. Thus, Prime+Abort does not apply to modern CPUs anymore.

2.2 Transient Execution Attacks

On x86, complex instructions are split into smaller micro-operations (μop), allowing superscalar optimizations. Instructions are decoded and committed to the architectural state in program order, with μops not necessarily executed in this order. If the CPU encounters a conditional branch, the CPU reduces pipeline stalls by *speculatively executing* the predicted code path. On a correct prediction, the speculatively-executed μops are committed to the architectural state, otherwise, the CPU discards the result of the μops. While non-committed instructions do not modify the architectural state, they can still affect the microarchitectural state, e.g., the cache. Such instructions are called *transient instructions* [12, 45, 53].

Transient-execution attacks are a new class of microarchitectural attacks relying on transient instructions to leak inaccessible data [10–12, 45, 47, 53, 65, 80, 82, 90, 91, 94]. These attacks are classified into two classes, Meltdown-type [53] and Spectre-type [45] attacks [12]. Meltdown-type attacks exploit that some CPUs defer permission checks for memory accesses during transient execution. This deferred permission check allows transient instructions to access normally inaccessible data from different security domains [11, 53, 62, 74, 80, 86, 90, 91, 94]. Spectre-type attacks [10, 32, 45, 47, 56, 65, 74] exploit transient instructions resulting from misspeculation. For both classes, the accessed data is encoded into a microarchitectural element and converted to the architectural domain using a microarchitectural covert channel, e.g., Flush+Reload, AVX2 wake-up times [82], port contention [10], or the TLB [49, 55]. While new CPUs mitigate Meltdown-type attacks in silicon, Spectre-type attacks are inherent to speculative execution [45, 53].

Spectre-BTB [45] can be mitigated automatically via software workarounds [89]. Mitigating Spectre-PHT [45] is not possible in an automated way without non-negligible performance penalties [41]. Manual mitigation requires developers to mitigate individual code sequences by inserting memory barriers [44] or using special hardened indexing functions as used in the Linux kernel. Despite automated mitigation attempts in specific environments [64], recent works showed that Spectre-PHT is still a threat by presenting Spectre-PHT attacks in JavaScript [83, 87] and even in the hardened Linux kernel [41]. Spectre-PHT [45] exploits the misprediction of conditional direct branches. If such a branch is used for an array-size check, the CPU can be tricked to speculatively execute an array access with an out-of-bounds value. The out-of-bounds value is encoded using a so-called *Spectre gadget* [45], and then transmitted to the architectural state using a covert channel, often Flush+Reload. In addition to Flush+Reload, different microarchitectural covert channels have also been used for Spectre-PHT attacks [10, 49, 52, 82, 88]. However, all these covert channels have in common that they require an architectural timer [83].

2.3 Idle-Loop Optimization

In addition to performance, power efficiency plays a vital role in modern CPUs. Reducing the consumed power leads to less heat and longer battery runtimes for mobile devices. As idle periods with low or nearly no computational workload are very common in systems, the CPU supports multiple performance and idle states. In these states, the CPU turns off or throttles unused functions to save power. A typical scenario for idle times is when no application is scheduled on a CPU core. In this scenario, the operating system (OS) spends time in an idle loop that puts the CPU to sleep until an interrupt arrives. The `monitor` and `mwait` instructions optimize such idle loops. Instead of requiring an interrupt to wake up, these instructions put the CPU into an optimized power state from which it wakes up when a specified memory location is modified [35]. For this, `monitor` arms the monitoring hardware with a virtual address to be monitored, and `mwait` hints the CPU to enter such an optimized state. In addition to this privileged instruction pair, there is an AMD-specific unprivileged instruction pair `monitorx` and `mwaitx` that can also be used in user-space applications [2]. These instructions additionally provide a timeout after which the CPU wakes up, regardless of whether there was an interrupt or a write to the monitored address. Intel also introduced an unprivileged version with the `umonitor` and `umwait` instructions. In the remainder of the paper, we refer to all of these instructions as `monitor` and `mwait` if we do not specify a specific variant.

3 Memory Monitoring

In this section, we analyze the wake-up behavior of the `mwait` instructions on Intel and AMD CPUs, the `umwait` instruction on Intel CPUs, and `mwaitx` on AMD CPUs. We show undocumented behavior when using these instructions in combination with transient execution and cache-maintenance functions.

Table 1: Analysis of which wake-up triggers, i.e., actions on the monitored cache line, wake up which `mwait` variant.

Access	Trigger	UMONITOR	MONITORX	MONITOR
architectural	Write	✓	✓	✓
	Flush	✗	✓	✓
	`clzero`	N/A	✓	✓
	`clwb`	N/A	†	†
	`prefetchw`	✓	†	†
transient	Speculative write	✓	†	†
	Write after exception	✓	†	†

† only on Zen 3, not on Zen or Zen+.

Setup. We test on different microarchitectures, namely Jasper Lake (Intel Celeron N4500; Tremont-based; microcode 0x24000014), Alder Lake (Intel Core i9-12900K; microcode 0xf), Whiskey Lake (Intel Core i7-8565U; microcode 0xec), Comet Lake (Intel Core i7-10710U; microcode 0xe8), Zen (AMD Ryzen 5 2500U; microcode 0x810100b), Zen+ (AMD Ryzen 5 3550H; microcode 0x8108102), and Zen 3 (AMD Ryzen 9 5900HX; microcode 0xa50000c). All systems are running Ubuntu 20.04 LTS (Linux kernel 5.4).

3.1 Wake-up Trigger

When entering an optimized power state using `mwait`, the official documentation states that the CPU wakes up either when a store matches the address range specified with `monitor`, an interrupt occurred, or a potential timeout is reached [2, 34]. Except for `umwait`, there is no indication for the wake-up reason. However, in addition to the documented wake-up events, our experiments show that not only writing directly to the monitored virtual address trigger the wake-up.

To evaluate if modifications to aliased addresses wake up the CPU as well, we map two different virtual addresses to the same physical address. We do not observe any difference between writing to these aliased addresses and writing to the monitored address. Moreover, a write to any offset within the monitored cache line leads to a wakeup, whereas writes outside the cache line do not trigger a wakeup. This indicates that the implementation is based on the physical cache line and not by matching the virtual address. This behavior is the same for all `mwait` variants on both Intel and AMD CPUs.

Second, cache-maintenance functions, such as flushes, also wake up the `mwait` instruction both on Intel and AMD. However, this is only the case for `mwait` and `mwaitx`, while the `umwait` instruction does not wake up from a flush of the monitored address range. While normal software prefetches (`prefetchnt*`, `prefetcht*`) do not cause a wakeup, prefetches for write (`prefetchw`) also cause a wakeup on Intel CPUs and AMD's Zen 3 microarchitecture. This is in line with concurrent work showing that `prefetchw` changes the cache-coherence state of the target address to *modified* [29]. For completeness, we also show that cache-maintenance functions with implicit writes, i.e., `clzero` and `clwb`, also wake up the `mwait` instructions on AMD. While `clwb` only triggers a wakeup on Zen 3, `clzero` also triggers a wakeup on Zen and Zen+. Transient writes wake up the `umwait` instruction. For the `mwaitx` and `mwait` instructions, transient writes only act as wake-up triggers on AMD Zen 3.

Table 2: Wake-up triggers on different memory types.

Memory Type	UMONITOR	MONITORX	MONITOR
Write-back	✓	✓	✓
Write-through	✓	✗	✗
Uncachable	✓	✗	✗
Invalid	✢	✢	✢

✢ CPU does not enter a sleep state.

Based on these observations, we hypothesize that the `monitor` and `mwait` instructions are implemented using the cache-coherence protocol. If the cache-coherence state for a cache line changes to the *modified* state, the `mwait` instruction wakes up, independent of the actual modification of the *cache line*. We also verified this by observing that non-architectural writes to read-only pages behave the same, which is in line with the prefetch observations by Guo et al. [29]. Table 1 summarizes the results for different triggers.

3.2 Memory Types

As suggested by the description of the `monitor` instruction in the Intel and AMD manuals [2, 35], the monitored address range should be write-back memory. Monitoring a non-write-back memory is not guaranteed to trigger a wakeup. Our experiments confirm that writes to a write-back memory range wake up the CPU from the power-optimized state for the `umwait`, `mwaitx`, and `mwait` instructions. To evaluate the behavior with undocumented memory types, we rely on PTEditor [78] to manipulate the page-table bits of the target address. Table 2 summarizes the overall wake-up-trigger results for different memory types. We verify that writing to uncachable or write-through memory does not trigger the `mwaitx` and `mwait` instructions to wake up on any microarchitectures. However, if the `umwait` instruction monitors such a memory type, a modification still triggers the wakeup. We confirm that the CPU does not enter a power-optimized state if the `monitor` instruction suffers an exception, e.g., a page fault. Moreover, speculative execution of `monitor` does not arm the monitoring hardware either.

3.3 Timeouts

On the Tremont microarchitecture, the `umwait` instruction features two timeouts. First, the instruction takes a user-provided timestamp at which the instruction continues if no memory write was observed until that point. Second, there is a maximum waiting time that the OS can define. The `umwait` instruction uses the 32-bit value stored in the `IA32_UMWAIT_CONTROL` MSR (`0xe1`), as the maximum number of cycles it waits before it continues [34]. Hence, on

a CPU with 2 GHz, the maximum waiting time is approximately 2 s, or unlimited if the value is zero. On Linux (kernel 5.11), the OS timeout of `umwait` is set to 100 000 cycles while there is no maximum time set on Windows 10 (version 20H2) by default.[1] If a user-defined timestamp and an OS-defined maximum waiting time are provided, the CPU always uses the timeout that is reached first. Intel also supports an undocumented timed `mwait` feature [34]. We experimentally confirmed that the feature can be enabled by setting bit 31 in MSR (`0xe2`) [46]. We further reverse engineered the feature and found that bit 1 of the `ECX` register of the `mwait` instruction indicates that the timeout feature is used. The maximum waiting time is an implicit 64-bit timestamp-counter value stored in the `EDX:EBX` register pair. The timed `mwait` wakes up when the timestamp counter reaches or exceeds the specified 64-bit value. We verified that this undocumented timeout feature works since the Skylake microarchitecture.

On AMD, the `mwaitx` instruction features a user-defined timeout. This feature is enabled by setting bit 1 of the `ECX` register when executing `mwaitx`. In contrast to the `umwait` and `mwait` instruction, where the timeout value is set as an absolute 64-bit value, `mwaitx` uses a 32-bit relative timeout in CPU cycles via the `EBX` register.

3.4 Wake-up Latency

We also analyzed the wake-up latency of `umwait`, `mwaitx`, and the undocumented timed-`mwait` instructions with their timeout features. We observed that the number of cycles spent waiting after the timeout differs slightly across these three instructions. On Jasper Lake, `umwait` wakes up about 6 to 32 cycles after the timestamp counter exceeds the timer's value provided by the user. When the timeout is triggered by the OS-defined maximum waiting time, `umwait` wakes up about 32 cycles after the OS-defined time has elapsed, which is in line with the user-defined timeout feature. Note that such a short wake-up latency can provide a fast synchronization primitive. Compared to the `umwait` instruction, the timed `mwait` and `mwaitx` instructions have a higher wake-up latency. Specifically, when the provided 64-bit timestamp has expired, the `mwait` instruction wakes up after about 696 cycles on an Intel Core i7-8565U (Whiskey Lake). The latency of the `mwaitx` instruction is about 932 cycles on AMD Ryzen 5 2500U (Zen) and about 1654 cycles on AMD Ryzen 9 5900HX (Zen 3).

3.5 Virtualization

To analyze the behavior of the `umwait` instructions, we performed the wake-up trigger experiments (cf. Section 3.1) inside virtual machines (VMs). We used the newest version of QEMU (6.1.0) with KVM that supports the `umwait` instruction. KVM does not trap the `umonitor` instruction. Hence, it also works inside VMs. The wake-up behavior is the same as when executed natively (cf. Table 1). In contrast, `monitorx` inside a VM on AMD behaves like a `nop`, as it is trapped by current versions of KVM and emulated as a `nop`.

[1] The value of the MSR is set to 0 on Windows.

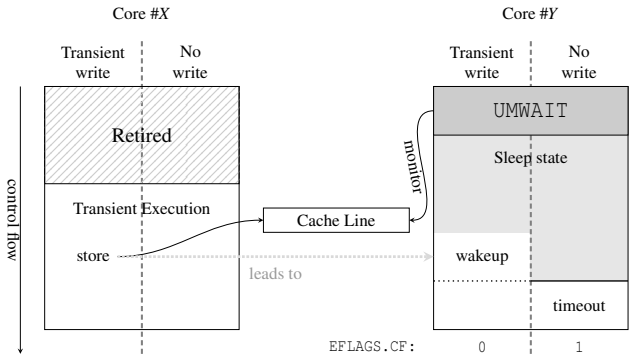


Figure 1: TWM: Transient stores wake up the `umwait` instruction. On such a wakeup, the carry flag (`CF`) is '0', while it is '1' if the instruction wakes up due to a timeout. This property provides an architectural information about a transient event.

4 Attack Primitives

In this section, we describe the attack primitives that we use for our case studies. First, we show how to use the `monitor` and `mwait` instructions to directly convert microarchitectural state changes into architectural state changes (Section 4.1). Second, we show how to use the precise timeout mechanism of the `umwait` instruction to measure timing differences without access to an architectural timer (Section 4.2).

4.1 TWM: Architectural Monitoring of Transient Writes

Our analysis of the `monitor` and `mwait` instruction family (cf. Section 3) shows that, although not documented, transient writes wake up the `umwait` instruction. Moreover, the `umwait` instruction provides the architectural information whether it woke up due to the OS timeout, or due to some trigger, e.g., modification of the monitored address range. By combining these two properties, we build a primitive that reports a microarchitectural event (transient write) via an architectural interface (carry flag). We refer to this primitive as TWM (transient-write monitor).

Figure 1 illustrates the basic idea of TWM. The setup consists of two CPU cores, *Core #X* and *Core #Y*, and a shared cache line. It does not matter whether the two cores are physical cores or logical cores. Core #*Y* first arms the monitoring hardware by executing the `umonitor` instruction with the cache line as the target. This is the initial step, generally also referred to as *prime* or *reset* step in other side-channel attacks [72, 98]. With the armed monitoring hardware, Core

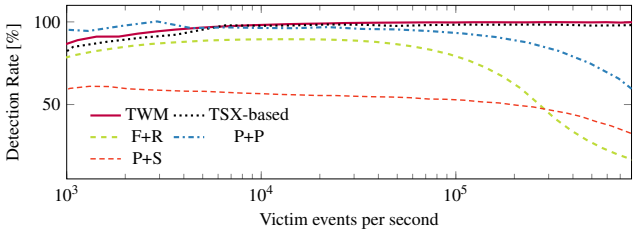


Figure 2: Detection rate for asynchronous events using TWM and compared paradigms. The chosen window size for Flush+Reload and the TSX-based primitive allows for a blind spot rate of less than 5 %.

#Y can now enter the *probe* or *measure* loop. By executing the `umwait` instruction, the core enters a light-weight sleep mode, typically C0.1 or C0.2 [34]. There are now two cases to distinguish: Core *#X* transiently writing or not writing to the monitored cache line. If Core *#X* transiently writes to the monitored shared cache line, Core *#Y* wakes up and continues execution with a cleared carry flag (`CF` = 0). If Core *#X* does not write to the monitored shared cache line, Core *#Y* sleeps until the maximum sleep time defined by the OS is reached (cf. Section 3.3). In this case, the carry flag is set (`CF` = 1) when Core *#Y* wakes up. Hence, with the carry flag, an attacker has the *architectural* information whether there was a *microarchitectural* event, i.e., a transient write.

Evaluation. To comprehensively evaluate TWM, we construct a standard benchmark for detecting fully asynchronous events with TWM and other conventional side-channel attacks for reference [16, 70, 72, 100]. The victim event is either a transient access or a transient write to a targeted cache line pinned to a different physical core than the attacker. Each asynchronous event is triggered after the process relinquishes the CPU and waits for a delay loop. The iteration number is randomly selected below a decreasing threshold. To simulate the frequency of the asynchronous events, we reduce the threshold per 1000 s. As no microarchitecture supports both TSX and `umonitor`, we run the experiments on an Intel Jasper Lake (Celeron N4500; microcode 0x24000014) that supports `umonitor` and an Intel Skylake (Xeon E3-1505M; microcode 0xd6) that supports TSX. While Prime+Scope cannot find an efficient prime pattern on our Jasper Lake machine, we test TWM and Flush+Reload on Jasper Lake and test the TSX-based primitive, Prime+Probe, and Prime+Scope on Skylake. Figure 2 shows TWM has a perfect recall of the victim event at frequencies from thousands to a million events per second. We identify that TWM has three additional advantages over conventional side-channel attacks.

First, TWM never misses an event during the monitoring phase, i.e., while `umwait` waits. Once TWM detects an event or the timeout of `umwait` is reached, `umwait` wakes up and the attacker re-arms address monitoring hardware. This is similar to the attacker waiting for an abort in TSX-based attacks. In contrast, Flush+Reload constantly resets the observed state and measures it again after waiting for a period. Suppose multiple events occur during the waiting and measurement periods. In that case, only one event can be detected, and the rest are missed. Attacks such as Prime+Probe and Prime+Scope can observe the cache state continuously and only reset the state after detecting an event. While Prime+Scope offers a rapid measurement for the eviction of a certain cache line, the probe step of Prime+Probe needs to access the entire eviction set, which costs hundreds to thousands of cycles. This effect explains why the performance of Prime+Probe declines at high frequencies as more events fall in the same probe period.

Table 3: Comparison of error rate and blind spot for detecting asynchronous events using TWM and compared paradigms.

Paradigm	WL	Event	Blind Spot	FP(/s)	FP under `stress`(/s)
Flush+Reload	✗	Transient Access	448 cycles	0	0
TWM	✓	Transient Write	175 cycles	263	297
TSX-based	✓	Transient Write	108 cycles	1609	63 063
Flush+Reload	✗	Transient Access	635 cycles	0	0
Prime+Probe	✓	Transient Access	4702 cycles	736	173 790
Prime+Scope	✓	Transient Access	3100 cycles	618	51 682

WL denotes windowless.

Second, the blind spot of TWM is relatively low compared to conventional side-channel attacks. Like all prior attacks, TWM also has a reset period to re-monitor the target address, namely the time between a timeout of `umwait` and its subsequent execution. In this reset period (*blind spot*), the side channel cannot detect the monitored event and thus misses it. Our experiment shows the average blind spot is 175 cycles on Jasper Lake, including a wake-up latency of `umwait` and the time of re-arming the monitoring hardware. With a monitoring period of 100 000 cycles, the blind spot is only 0.175 % of one iteration. This is significantly less than for other cache attacks [72]. For example, the blind spot for Flush+Reload on Jasper Lake is around 400 to 500 cycles, and the average blind spot for TSX-based attack is 108 cycles on our Skylake microarchitectures. Note that the blind spot rate of TSX-based attack could be decreased by waiting more time on each transaction, which in turn introduces more false positives. Furthermore, Prime+Scope behaves similarly to Prime+Probe but has more tension between the prime state and the victim event in a fully asynchronous attack. Specifically, accessing the cache line of the victim could lead to a failed prime stage, resulting in a higher number of missing events. To overcome this issue, the attacker has to repeat the prime stage in a chosen period.[2]

Third, we demonstrate that TWM is low-noise. The `umwait` instruction does not miss any writes, regardless of whether the writes are architectural writes or transient writes. Hence, we do not observe any false negatives, i.e., it never happens that a write is missed. We only observe false positives if an interrupt occurs while `umwait` is sleeping. However, interrupts are a common source of errors for traditional timing-based

[2] In our instance, the attacker repeats the prime stage after every five victim events for Prime+Scope.

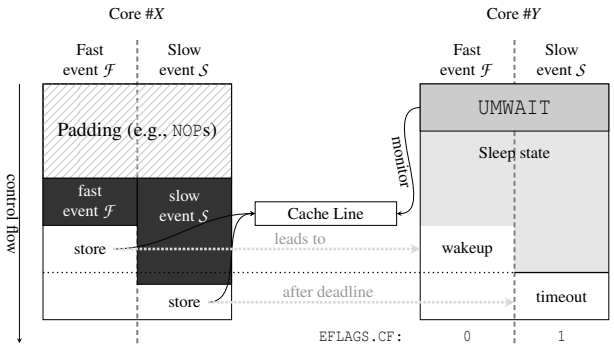


Figure 3: TLT: A padding is used to ensure that a store to the monitored cache line executes before or after the OS timeout of `umwait`, depending on the runtime of the event between padding and store. Calibrating the length of the padding once allows distinguishing fast and slow events, such as cache hits and misses, without an explicit timer.

side channels [58] and TSX-based side channels [16] as well. Table 3 shows the number of false positives (FP) by changing the victim to an unmonitored cache line. The FP of each primitive is the average over 100 runs of different event frequencies from thousands to one million per second. To evaluate the noise resistance of TWM, we run the `stress` utility in parallel to create pressure on the CPU and the memory subsystem. Even with `stress` running on all cores, FP are below 300 per second. In comparison, the FP of TSX-based primitive increases from 1609 to 63 063 per second. This is not surprising, as TSX transactions abort if the memory used inside the transaction is evicted from the cache [24].

4.2 TLT: Timer-less Timing Measurement

In addition to monitoring transient writes, we can also exploit the timeout function itself to distinguish the runtime of events without an explicit timer. We refer to this primitive as TLT (timing-less timer). Our attack primitive aims to distinguish a long-running from a short-running event. This is a typical scenario for side-channel attacks, where the leakage often manifests itself in a timing difference. For example, cache attacks distinguish fast cache hits from slow cache misses. As in TWM, we again rely on the architectural information provided by the `umwait` instruction, i.e., the carry flag. Hence, we can convert timing differences caused by the microarchitecture into an architecturally accessible information without using any architectural timing primitive.

Figure 3 illustrates the basic idea of TLT. Core #Y monitors a cache line using `umonitor` and sleeps using `umwait`. Core #X executes an event that can either be fast ($\mathcal{F}$) or slow ($\mathcal{S}$). Such a scenario is, e.g., typical for a timing-based side channel, where the event might be a memory load. The memory load is fast if it is served by the cache, or slow if it has to be served by the main memory. Core #X creates a padding, e.g., using `nop` instructions, such that the execution of the padding plus $\mathcal{F}$ finishes before the timeout of `umwait`, but the execution of the padding plus $\mathcal{S}$ finishes after the timeout of `umwait`. Note that this padding has to be created only once, e.g., in an initialization step in which an attacker can repeatedly trigger both $\mathcal{F}$ and $\mathcal{S}$. After the event, Core #X executes a store to the cache line monitored by Core #Y. Both Core #X and Core #Y start executing at the same time. If $\mathcal{F}$ is executed, the store hits the cache line while Core #Y sleeps, causing the carry flag to be set to '0'. In contrast, if $\mathcal{S}$ is executed, the store is executed after the OS timeout of `umwait`. Thus, Core #Y wakes up due to the timeout, and the carry flag is '1'. As a result, the carry flag provides an attacker the architectural information whether the measured event was fast ($\mathcal{F}$) or slow ($\mathcal{S}$), without an architectural timer.

5 Time-less Covert Channel

In this section, we use TWM to build a timeless cross-core covert channel to show that not even a coarse-grained architectural timer is required for synchronized communication.

5.1 Setup

In the covert-channel setup, we transmit data from one physical CPU core to a different physical CPU core. We do not assume any specific core assignment for the sender or receiver. Both sender and receiver can only execute unprivileged code. In line with many previous covert channels [21], we assume shared memory between sender and receiver, which can be read-only (e.g., a shared library) or writable. There is no requirement for the sender or receiver to synchronize the communication or agree on any common information before the transmission. The covert channel does not require any architecturally accessible timer, e.g., to count the number of events during a specific time window. The covert channel does not require cache-specific information, e.g., to build eviction sets [16]. We evaluate the channel on an Intel Core i9-12900K with Ubuntu 20.04 LTS (kernel 5.11.0).

5.2 Design

The sender and receiver use a cache line on the shared memory page for transmitting data without modifying the content of the cache line. The receiver repeatedly monitors the shared cache line using TWM. Hence, the carry flag of the receiver is '0' if the sender transiently wrote to the cache line, and '1' otherwise. As there are no synchronization or time slices used by the sender or receiver, the data transmission must be self-synchronizing. Our covert channel relies on the Manchester encoding often used for self-clocking data transmission on physical layers, such as in NFC, RFID, or consumer infrared protocols. With that encoding, every bit is encoded as a signal change, i.e., on the physical level as raising or falling edge. In

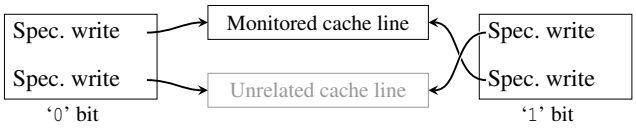


Figure 4: Every bit sent via the covert channel is encoded as the change of the carry flag. The only difference in sending a '0' and '1' is the order in which the speculative writes to the cache line are executed.

our scenario, the signal is the carry flag, and bits are encoded by a change in the carry flag (from '0' to '1' or from '1' to '0'). The sender either wakes up the receiver to set the carry flag to '0' or lets the receiver time out to set the carry flag to '1'. To send a Manchester-encoded bit, the sender toggles the carry flag from '0' to '1' or from '1' to '0'.

5.3 Implementation

For waking up the receiver, the sender speculatively writes to the monitored cache line. To increase the chance that the transient write wakes up the receiver, we ensure a large transient window that can transiently execute multiple writes. In contrast to the analysis by Ragab et al. [73], microcode assists did not lead to a longer transient window than branch mispredictions on our system. Hence, we used the RSB-based speculation technique introduced by Stecklina and Prescher [86] that was later classified as a Spectre variant as well (Spectre-RSB [12, 47, 56]). While this variant was not evaluated by Ragab et al. [73], it led to the most reliable transient window. With this mechanism, the transient write is executed successfully in 99.96 % of the cases.

Figure 4 shows how the Manchester encoding is implemented for the covert channel. To ensure that the execution times for triggering a wakeup and not triggering a wakeup do not differ significantly, the sender always accesses 2 cache lines, the monitored cache line, and an unrelated cache line. For a '0'-bit, the sender repeatedly transiently writes to the monitored cache line and then also repeatedly transiently to an unrelated cache line. Similarly, for a '1'-bit, the sender first repeatedly transiently writes to an unrelated cache line and then repeatedly to the monitored cache line.

While it is theoretically straightforward to trigger a change in the carry flag, there are certain pitfalls to avoid in the actual implementation. First, when the sender wakes up the receiver, the receiver records more events in the same time frame compared to when the receiver runs into the timeout. Thus, the period lengths for '0' and '1' bits differ, which requires special handling in the decoding step. Second, due to missing synchronization, the sender cannot simply induce a change in the carry flag. Instead, the sender has to keep each state of the carry flag for a longer period to ensure that the receiver captures it. This oversampling by the receiver also ensures that noise, e.g., unrelated changes of the carry flag, can be filtered in the post-processing phase. Third, one core can only monitor a single cache line at a time. Hence, synchronization of the bitstream, i.e., detecting at which bit the decoding has to start, must be solved on the protocol level, e.g., by sending a distinct preamble.

5.4 Evaluation

We evaluate the covert channel by repeatedly transmitting 100 B of data from one core to a different core. Due to the Manchester encoding, the data can be easily reconstructed without requiring an additional clock signal. '0' bits are encoded as rising edge ('0' → '1'), '1' bits are encoded as falling edge ('1' → '0'). Our covert channel achieves an average transmission rate of 697 bit/s with an error rate of 0 %.

6 Case Studies

In this section, we present 3 case studies based on the primitives introduced in Section 4 (TWM and TLT), and the interrupt-based wake-up behavior analyzed in Section 3.1. In the first case study (Section 6.1), we present Spectral, a way of exploiting Spectre attacks without requiring access to an architectural timer, neither a local timer [45, 47, 56, 87], nor a remote timer [82, 83]. In the second case study (Section 6.2), we demonstrate that we can reproduce the well-known cache attack on the OpenSSL AES T-table implementation without requiring an architectural timer. Finally, in the third case study (Section 6.3), we show that we can fingerprint the website a user opens without using any OS interface or architecturally available high-resolution timer.

6.1 Spectral: Architectural Spectre Attacks

In this case study, we present Spectral, a method to exploit existing Spectre variants without a timing-based side channel. While Spectral can be combined with different Spectre variants, we demonstrate it on Spectre-PHT [45] as this is a powerful combination and still easy to illustrate. Instead of using a timing-based side channel, Spectral uses TWM to convert transiently-accessed out-of-bounds data directly to architectural bits. We show that this variant is extremely reliable with leakage rates up to 200 kbit/s and error rates below 0.15 %, outperforming state-of-the-art attacks by factor 5.

6.1.1 Threat Model

We assume an unprivileged attacker without access to architectural timing primitives, including fine-grained architectural timing primitives, such as `rdtsc` or a counting thread [81], as well as coarse-grained architectural timers, e.g., timers with microsecond or lower resolution [83, 87]. Spectral relies on a bit-wise leakage gadget, similar to those used in previous

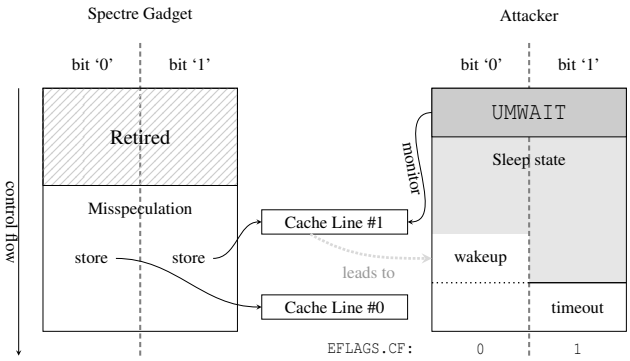


Figure 5: The mispredicted branch of the Spectral gadget reads a bit out of bounds. Based on this bit, there is a transient store to one of two cache lines. TWM monitors one of these cache lines to convert the microarchitectural state to an architectural state, and thus the leaked bit.

work [10, 55, 82, 83]. As discussed in existing Spectre attacks [45,47,56,82], finding such gadgets is orthogonal to our work. For the evaluation, we follow best practices and inject our own gadget into the victim [45, 47, 56, 82]. For the covert channel, we assume shared (read-only) memory between the victim and the attacker. We do not require hyperthreading.

6.1.2 Spectral Attack

Spectral relies on TWM to leak values without an architectural timer. As traditional Spectre attacks [45], Spectral also relies on the cache to temporarily encode the leaked values. Using the cache has the advantage that no rarely-used instructions have to be found for Spectre gadgets [10, 82, 98].

Figure 5 shows the overall idea of Spectral with a sample gadget shown by Loughlin et al. [55] (`if(x < array1_size) array2[array1[x] * 64] = ...;`). In case the user-provided array index `x` is out of bounds, there is a transient write to a memory location depending on the out-of-bound value. Instead of recovering the cache state via a timing side channel [12, 44, 47, 56], Spectral directly observes this transient memory write. Figure 6 illustrates the high-level overview. The attacker mistrains the conditional branch and uses TWM to monitor the cache line (`array2 + 64`) that is transiently modified if the leaked out-of-bound bit is '1'. In parallel to starting the monitoring, the attacker executes the gadget with an out-of-bound index addressing the target bit to leak. If the out-of-bound value is '1', the transient write wakes up the monitoring thread. As a result, the carry flag of the monitoring thread is cleared. Otherwise, if the out-of-bound value is '0', the transient write does not affect the monitoring thread. As a result, the `umwait` in the monitoring thread times out and sets the carry flag to '1'. Hence, the architectural carry flag of the monitoring thread is the inverse of the transiently leaked bit. These steps can be repeated for every bit that should be leaked. As the attacker calls the gadget and starts TWM, synchronization is trivial. Note that in contrast to SMoTherSpectre [10], Spectral does not require timing measurements, nor does it require hyperthreading or perfect synchronization. Another advantage of Spectral compared to traditional Spectre variants is that the gadget's spreading value has fewer limitations. For Spectral, this value only has to be ≥ 64. While cache-based covert channels theoretically work with ≥ 64, they typically use values ≥ 4096 to avoid prefetching effects [45]. TLB covert channels even require this value to be ≥ 4096 [49, 52, 55]. Note that a speculative write on read-only memory does not abort a TSX transaction, hence Spectral cannot be used with Prime+Abort [16]. Using Prime+Abort would either require shared writable memory in the gadget, or a gadget that evicts a cache set based on the secret out-of-bounds bit.

6.1.3 Results

We evaluate Spectral on an Intel Celeron N4500 (Jasper Lake) and an Intel Core i9-12900K (Alder Lake) running Ubuntu 20.04 LTS. All default Spectre mitigations are enabled. The victim containing the Spectral gadget and the attacker thread are running on different physical cores. We rely on the mistraining strategy from Kocher et al. [45] which provides 5 in-bound indices for every out-of-bound index. For every bit we leak, we call the gadget 15 times to increase the chance of inducing misspeculation.

With the default `mwait` OS timeout of 100 000 cycles on Linux, we achieve an average leakage rate of 58 963 bit/s with an error rate of 0.44 %, resulting in a true capacity of 56 562 bit/s. Due to the nature of Spectre attacks, an attacker can choose a trade-off between leakage rate and error rate by simply repeating the attack multiple times per bit. Spectral already has a low error rate, as the majority of errors are due to interrupts waking up the `mwait` instruction although there was no transient write. Hence, reducing the error rate is as simple as mounting the attack multiple times and only accepting a bit if the carry flag is the same for all runs.

This leakage rate outperforms all timing-based Spectre attacks and is nearly twice as fast as the fastest attack (cf. Schwarzl et al. [83] Table I). Still, in contrast to these attacks, the measured leakage rate of Spectral is "artificially" limited by the choice of the `umwait` timeout value. While not changeable by an unprivileged attacker, we still evaluate different timeout values to show the limits for the leakage rate.

Figure 6 shows the average true capacity and error rate for different `umwait` timeouts from 20 000 to 200 000 cycles. For timeouts below 23 000 cycles, we observe high error rates, as the `umwait` instruction often wakes up before the gadget can execute the transient write. Spectral achieves the highest true capacity for timeouts around 23 000 cycles, with an average true capacity of 199 800 bit/s. Larger timeouts lead to

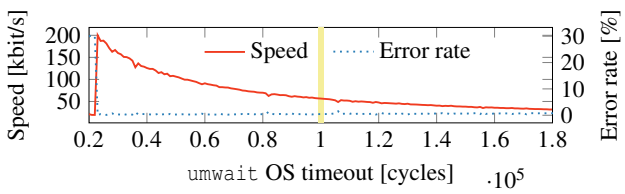


Figure 6: The average leakage (true capacity) and error rates for different values of the `umwait` timeout. The yellow rectangle highlights the default timeout of 100 000 cycles on Linux. Too short timeouts lead to aborts before the transient write is detected. Higher timeouts lead to lower leakage rates.

decreasing leakage rates, as every '0' bit has to wait for the entire timeout. Moreover, larger timeouts lead to increasing error rates due to the higher probability of interrupts waking up `umwait`. Still, the error rate is consistently below 1 %.

6.2 Timerless Cache Attacks

In this case study, we revisit the well-known cache attack on the OpenSSL AES T-table implementation [66]. As a victim, we use OpenSSL 1.0.1e, which is known to be susceptible to cache attacks [16, 23, 25, 37, 72, 93]. While OpenSSL deprecated and disabled the AES T-table implementation for security reasons, it is still a valuable benchmark for comparing cache attacks [16, 23, 72, 93]. We perform a timing-less cache attack on AES T-tables based on TLT (cf. Section 4.2) and compare it with Flush+Reload and Prime+Probe.

6.2.1 Setup

The AES T-table implementation features four T-tables, each spanning 16 cache lines to compute the ciphertext. The memory-access pattern of the AES T-table depends on the plaintext p and the secret key k.

In line with previous attacks [16, 25, 72], our attack is also fully automated. We rely on template attacks as proposed by Gruss et al. [25] to profile and exploit cache-based information leakage automatically. However, instead of measuring memory-access times, we rely on TLT. For TLT, our attack approach needs to find an appropriate padding to distinguish fast and slow events, i.e., cache hits and misses. The padding has to ensure that executing the padding, a subsequent *cache hit*, and a write all execute before the OS timeout of `umwait`. In contrast, executing the padding, a subsequent *cache miss*, and a write should not execute before the OS timeout. This padding can be determined automatically.

The entire automated calibration step takes around 250 s. The execution time of the calibration highly depends on the `umwait` OS timeout. For example, it takes 6 s to calibrate the threshold when the OS timeout of `umwait` is 500 cycles.

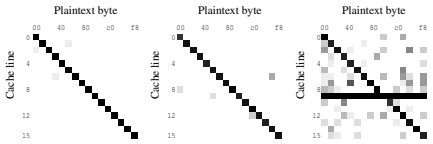


Figure 7: Cache hits on the first T-table with Flush+Reload (left), TLT (middle), Prime+Probe (right) with 300 encryptions. Darker means more cache hits. In all cases k_0 = `0x00`.

6.2.2 Exploitation

While previous cache attacks require an architectural timer to distinguish cache hits and misses, our attack does not. For the proof-of-concept implementation, we assume that the T-table addresses are known to the attacker. The attacker uses TLT on one CPU core, monitoring the T-table addresses on a different core. In line with previous work [16, 23, 72], we do not perform a full key-recovery attack since we only aim to compare our new attack with other techniques.

6.2.3 Results

We evaluated the new TLT-based attack and compared it to the Flush+Reload-based version and the Prime+Probe-based version on an Intel Celeron N4500 (Jasper Lake). Figure 7 presents a comparison of our AES T-table attack with regular Flush+Reload-based and Prime+Probe-based attacks. It shows the visible T-table access pattern with one fixed key byte, which is generated by these attacks within 300 encryptions separately. Similar results can be found in previous work [23, 25, 72]. We perform encryptions until the correct guess for the upper 4 bits of key byte k_0 from all the candidate key bytes can be identified with a 5 % margin. TLT requires around 1.6 times as many encryptions as Flush+Reload and 12.5 times less than Prime+Probe.

On average, we derived 64 bits of the secret key in around 38 s using the TLT-based attack. With fewer than 1000 encryptions, the number of encryptions has the same order of magnitude as Flush+Reload-based attacks. Hence, improvements for Flush+Reload-based attacks, such as noise reduction or performance degradation [4] also improve the TLT-based attack. Besides, the total attack time spent can be further reduced when a smaller OS timeout value of `umwait` exists.

6.3 Website Fingerprinting

In this case study, we show that the `mwait` instructions are not only useful for timerless attacks. While there is no `mwait` instruction on ARM, we show that the similar but much more limited `wfi` instruction that suspends the CPU core until an interrupt or a debug entry request is received [6] can also be used for this case study. When combined with a coarse-grained timer (around 10 ms), they can also be exploited for interrupt-based attacks. By counting the number of wakeups

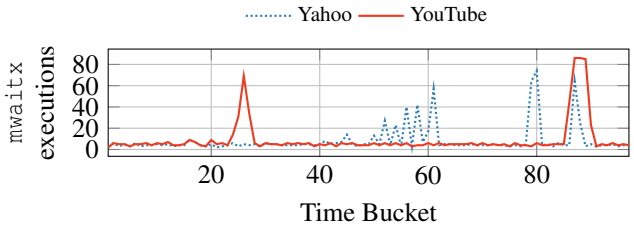


Figure 8: The number of `mwaitx` executions per 10 ms each over 1 s (100 time buckets) when opening Yahoo and YouTube. The more traffic there is in a time bucket, the more executions of `mwaitx` are observable.

during a 10 ms interval caused by network interrupts, we can reliably detect which website a user opens.

6.3.1 Threat Model

We assume an unprivileged attacker running an unprivileged application that can execute the `umwait` (Intel), `monitorx` (AMD), or `wfi` (ARM) instruction. The system does not expose interrupt or network statistics via any unprivileged interface [101]. The attacker can also use a coarse-grained timer with millisecond resolution, e.g., `clock_gettime` or `setitimer`. We assume that the system has more CPU cores than network cards to monitor interrupts on all network cards.

6.3.2 Spying on Network Interrupts

To spy on network interrupts, we rely on the property of the `mwait` instructions that they also wake up if there is an external interrupt [2, 34], which is also the case for the ARM `wfi` instruction. Transmitting and receiving network packets causes such interrupts that lead to wakeups. Hence, executing `umwait` (Intel), `mwaitx` (AMD), or `wfi` (ARM) on the core that receives the network interrupts leads to constant (spurious) wakeups when there is network traffic.

We rely on a coarse-grained timer to provide fixed time intervals, so-called *time buckets*. We count how many executions of an `mwait`/`wfi` instruction fall into a time bucket. The more often the instruction is interrupted, the faster the average execution time of the instruction. Hence, this results in a higher number of executions that fall into the same time bucket. For our evaluation, we chose 10 ms as a bucket size. Figure 8 shows a trace for opening two websites where the distribution of interrupts over the loading differs significantly.

6.3.3 Website Classification

For the classification of the websites, we rely on existing machine learning tools and techniques [69]. Our recorded traces are time series, as they contain features over the time it takes to load the website. In our case, the features are the number of executions of the `mwait`/`wfi` instruction. As we always record the interrupts for a fixed period, all traces have an equal length. However, the length of the information within a trace depends on two main factors: the size (and thus also the number of network requests) of the target website, as well as the transfer speed of the website. To account for both effects, we rely on Dynamic Time Warping (DTW) [7]. DTW measures the similarity of two time sequences where the speed changes dynamically within the time series. Hence, changes in the internet speed or latency do not impact the classification.

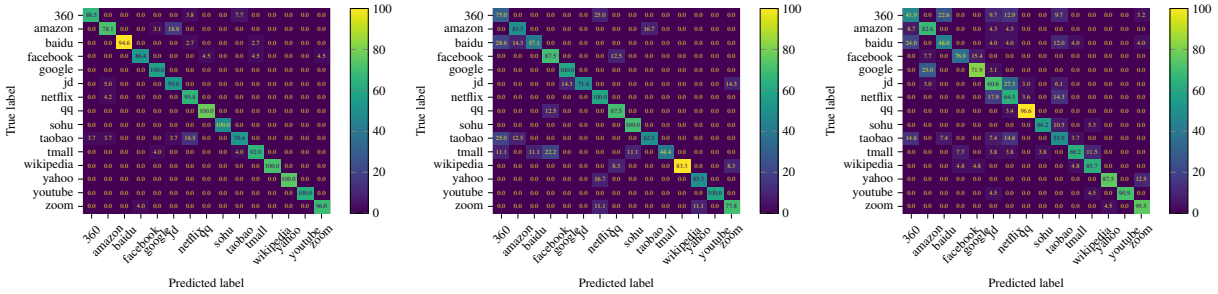
For the implementation, we rely on the open-source Python implementation by Löning et al. [54]. This implementation provides a ready-to-use k-nearest-neighbors time-series classifier with DTW. We split the collected traces into training and test data with a random 75 to 25 split. The k for the k-nearest-neighbors classification is simply set to the default of '1'. We cross-validate our classifier by running the evaluation multiple times with random splits of training and test data.

6.3.4 Results

We evaluated the website classification on three different machines, an Intel, an AMD, and an ARMv8 machine. On the Intel machine (Intel Celeron N4500 Jasper Lake, RTL8125-based Ethernet controller), we evaluate the classification with the unprivileged `umwait` instruction. On the AMD machine (AMD Ryzen 5 2500U, RTL8111/8168/8411-based Ethernet controller), we evaluate the classification with the unprivileged `mwaitx` instruction. On the ARMv8 machine (ARM Cortex A73, Amlogic Meson 6 DWMAC Ethernet controller), we evaluate the classification with the unprivileged `wfi` instruction. All machines run Ubuntu 20.04 LTS.

For the websites, we use the 100 most-visited websites as provided by the Alexa Top list.[3] We choose the Alexa Top 100 to be in line with related work [50, 75, 84], thus enabling a better comparison. We repeat the measurement 100 times for every website to ensure that our training and test sets are large enough for our classifier. For the sake of clarity, Figure 9 shows the confusion matrix for the top 15 sites based on the measurements on the 3 machines. Every cell represents the percentage that the classifier classifies a trace of the website specified by this row into the class specified by the column. For every site, the probability that the trace is classified correctly is the highest. This is also true for the top 100 sites. On AMD, the top 15 sites have an average classification accuracy of 92.6 %, and the top 100 sites still have an average classification accuracy of 78 %. Hence, the classifier performs significantly better than random guessing (6.7 % and 1 % respectively). For all tested websites, we achieve a precision of 92.7 % and a recall of 93.1 % for the top 15 sites, and a precision of 78 % and a recall of 78 % for the top 100 pages. The Intel machine performs slightly worse with a precision of 74.7 % and a recall of 73.9 % (top 15), and a precision of

[3] https://www.alexa.com/topsites



(a) AMD (b) ARM (c) Intel

Figure 9: The confusion matrix for the website classification on AMD, ARM, and Intel. For every site, the probability that the trace is classified correctly is the highest.

71 % and a recall of 70 % for the top 100 pages. Still, for every website, the probability that the trace is classified correctly is the highest. The reason for the worse performance is that the AMD machine has 4 physical cores with SMT, resulting in 8 logical CPUs, whereas the Intel machine only has 2 physical cores without SMT, resulting in 2 logical CPUs. Hence, the Intel machine suffers from more system noise, as all interrupts are distributed across fewer cores, leading to more different interrupt sources per core. On the AMD machine, Linux can distribute the interrupts more evenly across all the cores, resulting in much fewer other interrupts on the core that handles network interrupts. Still, both for the AMD and the Intel machine, the classifier vastly outperforms random guessing. On ARM, we achieve a precision of 67 % and a recall of 66 % for the top 100 pages. While slightly worse than on the x86 platforms, the results are still significantly better than random guessing.

Our attack performs similarly to previous work while having fewer requirements. Spreitzer et al. [85] report 89 % accuracy on 100 sites, whereas Jana and Shmatikov [40] report that between 30 % and 50 % of pages are distinguishable in the top 100 000 pages. Both approaches rely on the Linux `/proc` interface, which is restricted on newer Android versions, and unavailable on Windows. Lee et al. [48] exploit GPU vulnerabilities and report accuracies of 69.4 % and 60.9 % for the top 100 sites, depending on the exploited vulnerability. Gulmezoglu et al. [27] achieve an accuracy of 86.3 % for the top 30 sites plus 10 selected sites by relying on the–now restricted–access to hardware performance counters.

7 Countermeasures

In this section, we present countermeasures to prevent attacks exploiting the `mwait` instructions. We distinguish between countermeasures on the hardware, OS, and application layer.

7.1 Hardware

As a long-term solution, countermeasures should be implemented at the hardware level. We propose three solutions from which two are backward-compatible hardware changes to prevent our primitives.

Remove Wake-up Reason. The privileged `mwait` and the unprivileged AMD variant `mwaitx` do not provide architectural feedback. Hence, a possible hardware solution is to change the `umwait` instruction to not update the carry flag. We expect that a microcode update could change that behavior, as changing instruction behavior using microcode was done in the past as well [74, 80].

Remove Functionality. A simple solution is to introduce an MSR bit or use a bit in the `CR4` control register to convert the instruction to a `nop`. As applications have to handle cases in which `umwait` behaves architecturally equivalent to the `nop` instruction, this is fully backward compatible and similar to the handling of `mwaitx` inside VMs (cf. Section 3.5).

Instruction Deprecation. Another more drastic variant is to deprecate and disable the family of `mwait` completely. Even though this breaks backward compatibility, Intel already deprecated other instruction set extensions, such as TSX [60]. Doing so would trivially also prevent all discussed attacks. With `idle=nomwait`, Linux also supports a command-line kernel parameter to not use the `mwait` instruction at all.

7.2 Operating System and Hypervisor

Mitigations on the OS and hypervisor level are the most realistic short- and mid-term solutions as they can be deployed via software updates.

Disable in the OS. Setting the `CR4.TSD` bit disables the unprivileged execution of `rdtsc` and `rdtscp` [34]. Although not mentioned in the description of this bit, this bit is also repurposed to prevent the unprivileged execution of `umwait` according to the instruction description in the Intel manual [35]. We verified this behavior by setting this bit at runtime, which indeed leads to a general-protection fault when executing `umwait`. However, setting this bit has the side effect that no

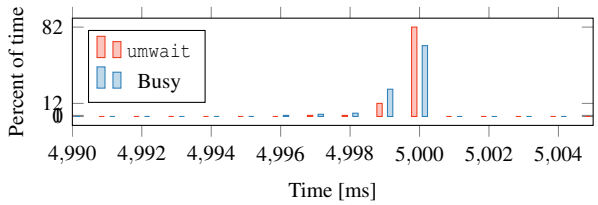


Figure 10: The CPU frequency distribution when executing `umwait` and a busy loop.

unprivileged application can use the time-stamp counter. On the hypervisor level, disabling `umwait` is possible without the side effect of disabling `rdtsc` and `rdtscp`. Intel provides a VM-execution-control bit to trap the `umwait` instruction [34]. As shown in Section 3.5, KVM in the current Linux kernel (5.11.0) does not trap `umwait`.

There is no documented method to disable the AMD-specific variants `monitorx` and `mwaitx`. The AMD manual explicitly states that the MSR (`0xc0010015`) does not affect these instructions [2]. Similar to `umwait`, a hypervisor can also trap the `mwaitx` instruction. KVM in the current Linux kernel (5.11.0) traps `mwaitx` and emulates it as a `nop`.

Virtualization-based security (VBS) [61, 97], as implemented in Windows 10 and 11, can mitigate our attack. With VBS, the OS is virtualized to implement security features in the hypervisor, letting the hypervisor trap the instructions.

Detection. Several approaches for detecting microarchitectural attacks use performance counters [13, 30, 38, 63, 68, 102, 104]. They generally exploit the fact that cache attacks typically result in a large number of cache misses while executing a small number of instructions. Hence, these techniques can be used indirectly, e.g., to detect our timing-less attack when used to measure cache hits and misses (cf. Section 6.2).

We did not find any performance counters on Intel or AMD related to the `mwait` instructions. Moreover, counters related to the C- and P-state did not capture the changes caused by the unprivileged `mwait` instructions. As the `mwait` instructions only switch to a special (sub) state within the C0 state [35], there is no difference in the counters. We also recorded the CPU frequency to detect possible frequency reductions when executing `umwait`. Figure 10 shows the CPU frequency distribution when running a busy loop and `umwait`. The busy loop ensures that the CPU is always at the maximum CPU frequency. However, `umwait` also stays in the highest C state, only in a substate. The entered sleep state is not deep enough to change the CPU frequency measurably. Future work has to investigate if our primitives can be detected indirectly, e.g., via stalling behavior and throughput.

OS Quota. Removing the `umwait` OS timeout mitigates our attack primitives (cf. Section 4). However, allowing an unprivileged user to halt or block a processor without a limit enables these users to mount trivial denial-of-service attacks against the system. Alternatively to disabling the timeout entirely, we propose to increase the timeout or randomize it. By increasing the timeout, the maximum bandwidth of covert channels and thus also the Spectral attack decreases. Moreover, distinguishing the timing of two events with TLT is less reliable with larger timeouts. An OS can randomize the timeout on context switch, introducing noise, and thus reducing the performance and reliability of the primitive.

7.3 Software

Our presented attacks are side-channel attacks, and hence an application can also be hardened against them.

Constant Time. Cryptographic algorithms can generally be implemented in a way that they are not susceptible to traditional side-channel attacks by ensuring that algorithms are data oblivious [36]. Such algorithms are in state-of-the-art cryptographic libraries such as OpenSSL or mbedTLS. While such implementations cannot be attacked using our primitives, they are still susceptible to Spectre attacks.

Spectre Mitigations. To prevent Spectre, including Spectral, the state-of-the-art mitigation technique is to add memory fences between conditional jumps and subsequent memory accesses [3, 5, 33]. Software developers have to ensure that no exploitable gadgets are in their software to prevent Spectre.

Noise. For single-shot attacks, such as the website classification (cf. Section 6.3), noise can be a viable hardening mechanism. By randomizing the order of requests or adding additional dummy requests, a browser could add randomness to the interrupt trace. While this does not entirely prevent an attack, it makes it more challenging, as many more traces are required to learn the interrupt behavior of a website. For attacks on repeatable events, such as the T-table attack (cf. Section 6.2), noise is not an effective mitigation. As any added random noise is statistically independent of the signal, it can be filtered out by averaging over many traces.

8 Discussion

Applicability As shown in our case studies, our primitives are a generic attack technique even if the ability only monitors write but not read accesses. They can be applied in scenarios where other cache attacks are also applicable. However, there are also scenarios where our primitives are advantageous over other attacks, as also shown in Section 4.1. Especially if noise resistance is required, our primitives are better suited than most other cache attacks. Moreover, our primitives perform especially well for monitoring high-frequency as it has a minimal blind spot. The most-similar attack is Prime+Abort [16]. However, the applicable CPUs for Prime+Abort and our primitives are mutually exclusive, as TSX is unavailable on CPU generations that support `umwait`. Furthermore, TSX is disabled on all currently available CPUs using microcode updates [60]. While there are differences to Prime+Abort, our side channel can act as a (drop-in) replacement on modern

CPUs. The differences are that there is no internal timeout for transactions [24]. Hence, Prime+Abort is not a generic timing replacement (cf. Section 4.2). Prime+Abort is also less stealthy, as TSX-related performance counters can be used as a detection, whereas we did not find counters for `umwait`.

Hypervisors and operating systems might block or emulate `rdtsc` [26,42,57,95]. Our paper shows that in such scenarios, it is also necessary to block or emulate the `umwait` instruction. Otherwise, an attacker can simply resort to `umwait`-based attacks. Similarly, static-detection methods [67] also require updates to not only detect `rdtsc`-based attacks.

Meltdown-type Attacks. We used our `umwait`-based primitive as the covert channel for a Spectre attack. Although we only show the primitive for Spectre attacks, we argue that the same is also true for the Meltdown-type class [12] of transient-execution attacks. For these attacks, it is even easier, as the attacker fully controls the leaking gadget and does not have to be found as in a Spectre attack. Moreover, multiple CPU cores can be used to leak multiple bits at once, improving the leakage rate compared to our Spectral attack. However, we, unfortunately, cannot evaluate such an attack because none of our machines that support the `umwait` instruction is vulnerable to any Meltdown-type attack.

Other OS. The concept is OS agnostic as we exploit the behavior of a CPU instruction. While the Linux kernel developers see a missing `umwait` timeout as a security problem, speculating that it can be used for covert channels or Spectre [14], Windows 10 does not specify a timeout. Coincidentally, it prevents attacks relying on the carry flag. We could not verify the default value on macOS, as there is no support for Tremont- or Alder-Lake-based CPUs.

9 Conclusion

In this paper, we presented novel techniques to convert microarchitectural states into architectural states without relying on time measurements but instead on undocumented behaviors of the unprivileged `umonitor` and `umwait` instructions. In three case studies, we showed the versatility of our primitive. We showed that we can mount efficient transient-execution attacks and traditional side-channel attacks without an architectural timer. In combination with a coarse-grained timer, we mounted interrupt-timing attacks on network requests. Our case studies highlight that the boundary between architecture and microarchitecture becomes blurry, leading to new attack variants and complicating effective countermeasures.

Availability

The source code for this paper is available on GitHub: https://github.com/CISPA/mwait.

Acknowledgments

We want to thank the anonymous reviewers for their guidance, comments and valuable suggestions. We also want to thank Niklas Flentje for fruitful discussions and providing feedback to drafts of this work. This work was supported in part by Semiconductor Research Corporation (SRC) Hardware Security Program (HWS).

References

[1] O. Acıiçmez, S. Gueron, and J.-p. Seifert, "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures," in *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, 2007.

[2] "AMD64 Architecture Programmer's Manual," Advanced Micro Devices Inc., 2017.

[3] "AMD64 Technology: Speculative Store Bypass Disable," Advanced Micro Devices Inc., 2018, revision 5.21.18.

[4] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom, "Amplifying Side Channels Through Performance Degradation," in *ACSAC*, 2016.

[5] "Cache speculation side-channels," ARM, 2018. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/Cache_Speculation_Side-channels_03May18.pdf

[6] "Dynamic power management," ARM, 2021. [Online]. Available: https://developer.arm.com/documentation/100095/0003/Functional-Description/Power-management/Dynamic-power-management

[7] D. J. Berndt and J. Clifford, "Using Dynamic Time Warping to Find Patterns in Time Series," in *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, 1994.

[8] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[9] S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, "Hardware prefetchers leak: A revisit of SVF for cache-timing attacks," in *MICRO*, 2012.

[10] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[11] 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.

[12] 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/.

[13] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," ePrint 2015/1034, 2015.

[14] J. Corbet, "Short waits with umwait," 2019. [Online]. Available: https://lwn.net/Articles/790920/

[15] W. Diao, X. Liu, Z. Li, and K. Zhang, "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis," in *S&P*, 2016.

[16] 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.

[17] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," in *USENIX Security*, 2022.

[18] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *CCS*, 2016.

[19] A. Fuchs and R. B. Lee, "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs," in *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*, 2015.

[20] C. P. García and B. B. Brumley, "Constant-Time Callees with Variable-Time Callers," in *USENIX Security Symposium*, 2017.

[21] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, 2016.

[22] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018.

[23] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[24] D. Gruss, F. Schuster, O. Ohrimenko, I. Haller, J. Lettner, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory," in *USENIX Security Symposium*, 2017.

[25] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[26] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *S&P*, 2011.

[27] 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.

[28] B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar, "A Faster and More Realistic Flush+Reload Attack on AES," in *COSADE*, 2015.

[29] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial prefetch: New cross-core cache side channel attacks," *arXiv:2110.12340*, 2021.

[30] 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.

[31] L. Hetterich and M. Schwarz, "Branch Different - Spectre Attacks on Apple Silicon," in *DIMVA*, 2022.

[32] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018.

[33] Intel, "Intel analysis of speculative execution side channels," 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf

[34] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2019.

[35] ——, "Intel 64 and IA-32 Architectures Software Developer′s Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z," 2021.

[36] 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

[37] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES," in *S&P*, 2015.

[38] ——, "Mascat: Preventing microarchitectural attacks before distribution," in *CODASPY*, 2018.

[39] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, Cross-VM attack on AES," in *RAID'14*, 2014.

[40] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *S&P'12*, 2012.

[41] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel," in *NDSS*, 2022.

[42] M. S. Karvandi, M. Gholamrezaei, S. Khalaj Monfared, S. Medi, B. Abbassi, A. Amini, R. Mortazavi, S. Gorgin, D. Rahmati, and M. Schwarz, "Hyperdbg: Reinventing hardware-assisted debugging," in *CCS*, 2022.

[43] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *DAC*, 2016.

[44] P. Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler," 2018.

[45] 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.

[46] A. Kogler, D. Weber, M. Haubenwallner, M. Lipp, D. Gruss, and M. Schwarz, "Finding and Exploiting CPU Features using MSR Templating," in *S&P*, 2022.

[47] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.

[48] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting gpu vulnerabilities," in *S&P*, 2014.

[49] M. Lipp, D. Gruss, and M. Schwarz, "AMD Prefetch Attacks through Power and Time," in *USENIX Security*, 2022.

[50] 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.

[51] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[52] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *AsiaCCS*, 2020.

[53] 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.

[54] M. Löning, A. Bagnall, S. Ganesh, V. Kazakov, J. Lines, and F. J. Király, "sktime: A unified interface for machine learning with time series," *arXiv:1909.07872*, 2019.

[55] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "DOLMA: Securing Speculation with the Principle of Transient Non-Observability," in *USENIX Security Symposium*, 2021.

[56] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.

[57] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," *ACM SIGARCH Computer Architecture News*, 2012.

[58] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.

[59] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv:1902.05178*, 2019.

[60] Michael Larabel, "Intel To Disable TSX By Default On More CPUs With New Microcode," June 2021. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode

[61] Microsoft, "Virtualization-based security (vbs)," 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs

[62] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis," in *USENIX Security Symposium*, 2020.

[63] 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.

[64] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[65] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter, "Spectre attack against SGX enclave," 2018.

[66] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[67] Y. Oyama, "How does malware use rdtsc? a study on operations executed by malware with cpu cycle measurement," in *DIMVA*, 2019.

[68] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS*, 2016.

[69] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.

[70] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.

[71] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[72] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.

[73] 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.

[74] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, 2021.

[75] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated Website Fingerprinting through Deep Learning," in *NDSS*, 2018.

[76] M. Schwarz, "Software-based Side-Channel Attacks and Defenses in Restricted Environments," Ph.D. dissertation, Graz University of Technology, 2019.

[77] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

[78] 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

[79] 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.

[80] 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.

[81] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.

[82] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.

[83] M. Schwarzl, P. Borrello, A. Kogler, K. Varda, T. Schuster, D. Gruss, and M. Schwarz, "Robust and scalable process isolation against spectre in the cloud," in *ESORICS*, 2022.

[84] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust Website Fingerprinting Through The Cache Occupancy Channel," in *USENIX Security Symposium*, 2019.

[85] R. Spreitzer, S. Griesmayr, T. Korak, and S. Mangard, "Exploiting data-usage statistics for website fingerprinting attacks on android," in *WiSec*, 2016.

[86] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," *arXiv:1806.07480*, 2018.

[87] Stephen Röttger and Artur Janc, "A Spectre proof-of-concept for a Spectre-proof web," 2021. [Online]. Available: https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html

[88] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols," *arXiv:1802.03802*, 2018.

[89] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[90] 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.

[91] 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.

[92] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution," in *USENIX Security Symposium*, 2017.

[93] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think," in *USENIX Security Symposium*, 2018.

[94] 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.

[95] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in Xen," in *CCSW*, 2011.

[96] D. Wang, A. Neupane, Z. Qian, N. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu, "Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries," in *NDSS*, 2019.

[97] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "SecPod: a framework for virtualization-based security systems," in *USENIX ATC*, 2015.

[98] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated Discovery of Microarchitectural Side Channels," in *USENIX Security*, 2021.

[99] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack," *Cryptology ePrint Archive, Report 2014/140*, 2014.

[100] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[101] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, 2009.

[102] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.

[103] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *CCS*, 2012.

[104] Y. Zhang and M. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *CCS*, 2013.