Speculative Dereferencing of Registers: Reviving Foreshadow

Martin Schwarzl
Graz University of Technology
martin.schwarzl@iaik.tugraz.at

Thomas Schuster
Graz University of Technology
thomas.schuster@student.tugraz.at

Michael Schwarz
CISPA Helmholtz Center for Information Security
michael.schwarz@cispa.saarland

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

ABSTRACT

Since 2016, multiple microarchitectural attacks have exploited an effect that is attributed to prefetching. These works observe that certain user-space operations can fetch kernel addresses into the cache. Fetching user-inaccessible data into the cache enables KASLR breaks and assists various Meltdown-type attacks, especially Foreshadow.

In this paper, we provide a systematic analysis of the root cause of this prefetching effect. While we confirm the empirical results of previous papers, we show that the attribution to a prefetching mechanism is fundamentally incorrect in all previous papers describing or exploiting this effect. In particular, neither the `prefetch` instruction nor other user-space instructions actually prefetch kernel addresses into the cache,[1] leading to incorrect conclusions and ineffectiveness of proposed defenses. The effect exploited in all of these papers is, in fact, caused by speculative dereferencing of user-space registers in the kernel. Hence, mitigation techniques such as KAISER do not eliminate this leakage as previously believed. Beyond our thorough analysis of these previous works, we also demonstrate new attacks enabled by understanding the root cause, namely an address-translation attack in more restricted contexts, direct leakage of register values in certain scenarios, and the first end-to-end Foreshadow (L1TF) exploit targeting non-L1 data. The latter is effective even with the recommended Foreshadow mitigations enabled and thus revives the Foreshadow attack. We demonstrate that these dereferencing effects exist even on the most recent Intel CPUs with the latest hardware mitigations, and on CPUs previously believed to be unaffected, *i.e.*, ARM, IBM, and AMD CPUs.

1 INTRODUCTION

Modern system security depends on isolating domains from each other. One domain cannot access information from the other domain, e.g., another process or the kernel. Hence, the goal of many attacks is to break this isolation and obtain information from other domains. Microarchitectural attacks like Foreshadow [94, 98] and Meltdown [57] gained broad attention due to their impact and mitigation cost. One building block that facilitates microarchitectural attacks is knowledge of physical addresses. Knowledge of physical addresses can be used for various side-channel attacks [24, 38, 58, 61, 75], bypassing SMAP and SMEP [43], and mounting Rowhammer attacks [7, 41, 46, 76, 86, 102]. As a mitigation to these attacks, operating systems do not make physical address information available to user programs [48]. Hence, the attacker has to leak the privileged physical address information first. The *address-translation attack* by Gruss et al. [22] solves this problem.[2] The address-translation attack allows unprivileged applications to fetch arbitrary kernel addresses into the cache and thus resolve virtual to physical addresses on 64-bit Linux systems. As a countermeasure against microarchitectural side-channel attacks on kernel isolation, e.g., the address-translation attack, Gruss et al. [21, 22] proposed the KAISER technique.

More recently, other attacks observed and exploited similar prefetching effects. Lipp et al. [57] described that Meltdown successfully leaks memory that is not in the L1 cache, but did not thoroughly explain why this is the case. Xiao et al. [103] show that this is only possible due to a prefetching effect, when performing Meltdown-US, where data is fetched from the L3 cache into the L1 cache. Van Bulck et al. [94] observe that for Foreshadow this effect does not exist. Foreshadow is still limited to the L1, however in combination with Spectre gadgets which fetch data from other cache levels it is possible to bypass current L1TF mitigations. This statement was further mentioned as a restriction by Canella et al. [11] and Nilsson et al. [68]. Van Schaik et al. state that Meltdown is not fully mitigated by L1D flushing [96].

We systematically analyze the root cause of the prefetching effect exploited in these works. We first empirically confirm the results from these papers, underlining that these works are scientifically sound, and the evaluation is rigorous. We then show that, despite the scientifically sound approach of these papers, the attribution of the root cause, *i.e.*, why the kernel addresses are cached, is incorrect in all cases. We discovered that this prefetching effect is actually unrelated to software prefetch instructions or hardware prefetching effects due to memory accesses and instead is caused by speculative dereferencing of user-space registers in the kernel. While there are multiple code paths which trigger speculative execution in the kernel, we focus on a code path containing a Spectre-BTB [11, 49] gadget which can be reliably triggered on both Linux and Windows.

Based on our new insights, we correct several assumptions from previous works and present several new attacks exploiting the underlying root cause. We demonstrate that an attacker can, in certain cases, observe caching of the address (or value) stored in a register of a different context. Based on this behavior, we present a cross-core covert channel that does not rely on shared memory.

[1] Various authors of papers exploiting the prefetching effect confirmed that the explanation put forward in this paper indeed explains the observed phenomena more accurately than their original explanations. We believe it is in the nature of empirical science that theories explaining empirical observations improve over time and root-cause attributions become more accurate.

[2] This attack is detailed in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [22]

While Spectre "prefetch" gadgets, which fetch data from the last-level cache into higher levels, are known [11], we show for the first time that they can directly leak actual data. Schwarz et al. [82] showed that prefetch gadgets can be used as a building block for ZombieLoad on affected CPUs to not only leak data from internal buffers but to leak arbitrary data from memory. We show that prefetch gadgets are even more powerful by also leaking data on CPUs unaffected by ZombieLoad. Therefore, we demonstrate for the first time data leakage with prefetch gadgets on non-Intel CPUs.

The implications of our insights affect the conclusions of several previous works. Most significantly, the difference that Meltdown can leak from L3 or main memory [57] but Foreshadow (L1TF) can only leak from L1 [94][3], was never true in pratice. For both, Meltdown and Foreshadow, the data has to be fetched in the L1 to get leaked. However, this restriction can be bypassed by exploiting prefetch gadgets to fetch data into L1. Therefore L1TF was in practice never restricted to the L1 cache, due to the same "prefetch" gadgets in the kernel and hypervisor that were exploited in Meltdown. Because of these gadgets, mounting the attack merely requires moving addresses from the hypervisor's address space into the registers. Hence, we show that specific results from previous works are only reproducible on kernels that still have such a "prefetch" gadget, including, e.g., Gruss et al. [22],[4] Lipp et al. [57],[5], Xiao et al. [103][6]. We also show that van Schaik et al. [96] (Table III [96]) erroneously state that L1D flushing does not mitigate Meltdown.

We then show that certain attacks can be mounted in JavaScript in a browser, as the previous assumptions about the root cause were incorrect. For instance, we recover physical addresses of a JavaScript variable to be determined with cache-line granularity et al. [22]. Knowledge of physical addresses of variables aids Javascript-based transient-execution attacks [49, 63], Rowhammer attacks [23, 41], cache attacks [71], and DRAMA attacks [83].

We then show that we can mount Foreshadow attacks on data not residing in L1 on kernel versions containing "prefetch" gadgets. Worse still, we show that for the same reason Foreshadow mitigations [94, 98] are incomplete. We reveal that a full mitigation of Foreshadow attacks additionally requires Spectre-BTB mitigations (`nospectre_v2`), a fact that was not known or documented so far.

We demonstrate that the prefetch address-translation attack also works on recent Intel CPUs with the latest hardware mitigations. Finally, we also demonstrate the attack on CPUs previously believed to be unsusceptible to the prefetch address-translation attack, *i.e.*, ARM, IBM Power9, and AMD CPUs.

Contributions. The main contributions of this work are:

(1) We empirically confirm the results of previous works whilst discovering an incorrect attribution of the root cause [22, 57, 103].
(2) We show that the underlying root cause is speculative execution. Therefore, CPUs from other hardware vendors like AMD, ARM, and IBM are also affected. Furthermore, the effect can even be triggered from JavaScript.
(3) We discover a novel way to exploit speculative dereferences, enabling direct leakage of data values stored in registers.
(4) We show that this effect, responsible for Meltdown from non-L1 data, can be adapted to Foreshadow by using addresses not valid in any address space of the guest.
(5) We analyze the implications for Meltdown and Foreshadow attacks and show that Foreshadow attacks on data from the L3 cache are possible, even with Foreshadow mitigations enabled, when the unrelated Spectre-BTB mitigations are disabled.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on virtual memory, cache attacks, and transient-execution attacks. In Section 3, we analyze the underlying root cause of the observed effect. In Section 4, we demonstrate the same effect on different architectures and improve the leakage rate. In Section 5, we measure the capacity using a covert channel. In Section 6, we demonstrate an attack from a virtual machine. In Section 7, we leak actual data with seemingly harmless prefetch gadgets. In Section 8, we present a JavaScript-based attack leaking physical and virtual address information. In Section 9, we discuss the implications of our attacks. We conclude in Section 10.

2 BACKGROUND AND RELATED WORK

In this section, we provide a basic introduction to address translation, CPU caches, cache attacks, Intel SGX, and transient execution. We also introduce transient-execution attacks and defenses.

2.1 Address Translation

Virtual memory is a cornerstone of today's system-level isolation. Each process has its own virtual memory space and cannot access memory outside of it. In particular, processes cannot access arbitrary physical memory addresses. The KAISER patch [21] introduces a strong isolation between user-space and address space, meaning that kernel memory is not mapped when running in user-space. Before the KAISER technique was applied, the virtual address space of a user process was divided into the user and kernel space. The user address space was mapped as user-accessible while the kernel space was only accessible when the CPU was running in kernel mode. While the user's virtual address space looks different in every process, the kernel address space looks mostly identical in all processes. To switch from user mode to kernel mode, the x86_64 hardware requires that parts of the kernel are mapped into the virtual address space of the process. When a user thread performs a syscall or handles an interrupt, the hardware simply switches into kernel mode and continues operating in the same address space. The difference is that the privileged bit of the CPU is set, and kernel code is executed instead of the user code. Thus, the entire user and kernel address mappings remain generally unchanged while operating in kernel mode. As sandboxed processes also use a regular virtual address space that is primarily organized by the kernel, the kernel address space is also mapped in an inaccessible way in sandboxed processes.

Many operating systems map physical memory directly into the kernel address space [44, 54], as shown in Figure 1, e.g., to

[3] Appendix Foreshadow's Cache Requirement [94] and subsequently also reported by Canella et al. [11] (Table 4 [11]), and Nilsson [68] (Section III.E [68].
[4] The address-translation oracle in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [22].
[5] The L3-cached and uncached Meltdown experiments in Section 6.2 [57].
[6] The L3-cached experiment in Section IV-E [103].

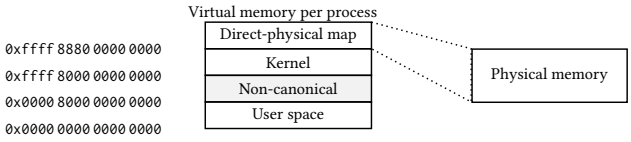

Figure 1: Physical memory is mapped into the huge virtual address space.

access paging structures and other data in physical memory. Paravirtualizing hypervisors also employ a direct map of physical memory [101]. Thus, every user page is mapped at least twice: once in user space and once in the kernel direct map. When performing operations on either one of the two virtual addresses, the CPU translates the corresponding address to the same physical address. The CPU then performs the operation based on the physical address.

For security reasons, access to virtual-to-physical address information requires root privileges [48]. The address-translation attack described in the Prefetch Side-Channel Attacks paper [22] obtains the physical address for any virtual address mapped in user space without root privileges. For the sake of brevity, we do not discuss the translation-level oracle also described in the Prefetch Side-Channel Attacks paper [22] which is an orthogonal attack and, to the best of our knowledge, works as described in the paper.

2.2 CPU Caches

Modern CPUs have multiple cache levels, hiding latency by buffering slower memory levels. Page tables are stored in memory and thus are cached by the regular data caches [35]. Page translation data is also stored in dedicated caches, called translation-lookaside buffers (TLBs), to speed up address translation. Software prefetch instructions hint to the CPU that a memory address will soon be accessed in execution and so it should be fetched into the cache early to improve performance. However, the CPU can ignore these hints [34]. Intel and AMD x86 CPUs have five software prefetch instructions: `prefetcht0`, `prefetcht1`, `prefetcht2`, `prefetchnta`, `prefetchw`, and on some models the `prefetchwt1`. On ARMv8-A CPUs we can instead use the `prfm` instruction and on IBM Power9 the `dcbt` instruction.

2.3 Cache Attacks

Cache attacks have been studied for more than two decades [5, 50, 72–74, 89]. Today, most attacks use either Prime+Probe [72], where an attacker occupies parts of the cache and waits for eviction due to cache contention with the victim, or Flush+Reload [104], where an attacker removes specific (read-only) shared memory from the cache and waits for a victim process to reload it. Prime+Probe has been used for many powerful cross-core covert channels and attacks [56, 58, 62, 71, 78, 81, 105]. Flush+Reload requires shared (read-only) memory, but is more accurate and thus has been the technique of choice in local cross-core attacks [25, 26, 39, 40, 106]. Flush+Reload has been used as a more generic primitive to test whether a memory address is in the cache or not [49, 57, 80, 94].

Prefetching attacks. Gruss et al. [22] observed that software prefetches appear to succeed on inaccessible memory. Using this effect on the kernel direct-physical map enables the user to fetch arbitrary physical memory into the cache. The attacker guesses the physical address for a user-space address, tries to prefetch the corresponding address in the kernel's direct-physical map, and then uses Flush+Reload on the user-space address. If Flush+Reload observes a hit, then the guess was correct. Hence, the attacker can determine the exact physical address for any virtual address, re-enabling side-channel [62, 75] and Rowhammer attacks [46, 86].

2.4 Intel SGX

Intel SGX is a trusted execution mechanism enabling the execution of trusted code in a separate protected area called an enclave. This feature was introduced with the Skylake microarchitecture as an instruction-set extension [35]. The hardware prevents access to the code or data of the enclave from any source other than the enclave code itself [37]. All code running outside of the enclave is treated as untrusted in SGX. Thus, code containing sensitive data is protected in the enclave even if the host operating system or hypervisor is compromised. Enclave memory is mapped in the virtual address space of the host application but is inaccessible to the host. The enclave has full access to the virtual address space of its host application to share data between enclave and host. However, as has been shown in the past, it is possible to exploit SGX via memory corruption [52, 80], ransomware [85], side-channel attacks [9, 81], and transient-execution attacks [82, 94, 96].

2.5 Transient Execution

Modern CPUs split instructions into micro-operations (μOPs) [18]. The μOPs can be executed *out of order* to improve performance and later on retire *in order* from reorder buffers. However, the *out-of-order* stream of μOPs is typically not linear. There are branches which determine which instructions, and thereby μOPs, follow next. This is not only the case for control-flow dependencies but also data-flow dependencies. As a performance optimization, modern CPUs rely on prediction mechanisms which predict which direction should be taken or what the condition value will be. The CPU then speculatively continues based on its control-flow or data-flow prediction. If the prediction was correct, the CPU utilized its resources more efficiently and saved time. Otherwise, the results of the executed instructions are discarded, and the architecturally correct path is executed instead. This technique is called speculative execution. Intel CPUs have multiple branch prediction mechanisms [34], including the Branch History Buffer (BHB) [6, 49], Branch Target Buffer (BTB) [17, 49, 53], Pattern History Table (PHT) [18, 49], and Return Stack Buffer (RSB) [18, 51, 60]. Lipp et al. [57] defined instructions executed out-of-order or speculatively but not architecturally as *transient instructions*. These *transient instructions* can have measurable side effects, e.g., modification of TLB and cache state. In transient-execution attacks, these side effects are then measured.

2.6 Transient-Execution Attacks & Defenses

As transient execution can leave traces in the microarchitectural state, attackers can exploit these state changes to extract sensitive information. This class of attacks is known as transient-execution attacks [1, 11]. In Meltdown-type attacks [57] an attacker deliberately accesses memory across isolation boundaries, which is possible due

to deferred permission checks in out-of-order execution. Spectre-type attacks [12, 27, 47, 49, 51, 60, 84] exploit misspeculation in a victim context. The attacker may facilitate this misspeculation, e.g., by mistraining branch predictors. By executing along the misspeculated path, the victim inadvertently leaks information to the attacker. To mitigate Spectre-type attacks several mitigations were developed [33]. For instance, retpoline [32] replaces indirect jump instructions with `ret` instructions. Therefore, the speculative execution path of the `ret` instruction is fixed to a certain path (e.g. to an endless loop) and does not misspeculate on potential code paths that contain Spectre gadgets. Foreshadow [94] is a Meltdown-type attack exploiting a cleared present bit in the page table-entry. It only works on data in the L1 cache or the line fill buffer [82, 96], which means that the data must have been recently accessed prior to the attack. An attacker cannot directly access the targeted data from the Foreshadow attack context, and hence a widely accepted mitigation is to flush the L1 caches and line fill buffers upon context switches and to disable hyperthreading [31].

3 FROM ADDRESS-TRANSLATION ATTACK TO FORESHADOW-L3

In this section, we systematically analyze the properties of the address-translation attack that were erroneously explained to be caused by the insecure behavior of software prefetch instructions.[7] We show that the address-translation attack [22] originally motivating the KAISER technique [21] was never related to prefetch instructions. Instead, it exploits a Spectre-BTB gadget [11] in the kernel and, as such, is not mitigated by the KAISER technique.[8]

In the address-translation attack [22] the attacker tries to verify whether two virtual addresses p and $\bar{p}$ map to the same physical address. For instance, on Linux, the corresponding direct-physical map address in the kernel can be used to verify the mapping. The attacker first flushes the user-space virtual address p. Then, the inaccessible (direct physical map address) $\bar{p}$ is prefetched using a software prefetch instruction. The address p is reloaded, and the timing of the reload is checked to verify whether the address is cached or uncached. If a cache hit is observed, the inaccessible virtual address $\bar{p}$ maps to the same physical address as the virtual address p. This procedure of flushing and reloading a virtual address is referred to as Flush+Reload [104]. The Flush+Reload part of the address-translation attack has an F1-Score very close to 1 [104], meaning that if there is a cache hit, it will be observed in virtually every case. The limiting factor of the attack is the probability that the guessed address is successfully "prefetched", as not every "prefetch" attempt brings the target address into the cache. Hence, we measure the attack performance in successful *fetches per second*. More fetches per second means a shorter time to mount an attack, e.g., one successful cache fetch enables leakage of 64 bytes in a Foreshadow attack, despite Foreshadow mitigations being enabled.

[7] This attack is detailed in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [22]. It should not be confused with the translation-level oracle described in Section 3.2 and Section 4 of that paper [22], which to the best of our knowledge has a correct technical explanation. We focus on the part that the authors confirmed to be incorrect, *i.e.*, the address-translation attack in Section 3.3 and Section 5.

[8] This was also independently confirmed by authors of the Prefetch Side-Channel Attacks paper [22] that are not co-authors of this paper.

```
for (size_t i = 0; i < 3; ++i) {
  sched_yield();
  prefetch(direct_phys_map_addr);
}
```

Listing 1: Original code of the released proof-of-concept implementation for the address-translation attack [29] from Gruss et al. [22].[14]The code "prefetches" a (guessed) physical address from the direct physical map. If the "prefetch" was successful and the physical address guess correct, the attacker subsequently observes a cache hit on the corresponding user-space address.

```
; %r14 contains the direct physical address
12b6: e8 c5 fd ffff callq  1080 <sched_yield@plt>
12bb: 41 0f 18 06     prefetchnta (%r14)
12bf: 41 0f 18 1e     prefetcht2 (%r14)
12c3: e8 b8 fd ffff callq  1080 <sched_yield@plt>
12c8: 41 0f 18 06     prefetchnta (%r14)
12cc: 41 0f 18 1e     prefetcht2 (%r14)
12d0: e8 ab fd ffff callq  1080 <sched_yield@plt>
12d5: 41 0f 18 06     prefetchnta (%r14)
12d9: 41 0f 18 1e     prefetcht2 (%r14)
```

Listing 2: Disassembly of the prefetching component of the prefetch address-translation attack.

The prefetching component of the original attack's proof-of-concept implementation [29] is shown in Listing 1. The compiled and disassembled code can be found in Listing 2. We analyze the original attack and observe the following requirements are described for the address-translation attack to succeed:

H1 the `prefetch` instruction (to instruct the prefetcher to prefetch);[9]
H2 the value stored in the register used by the `prefetch` instruction (to indicate which address the prefetcher should prefetch);[10]
H3 the `sched_yield` syscall (to give time to the prefetcher);[11]
H4 the use of the `userspace_accessible` bit (as kernel addresses could otherwise not be translated in a user context);[12]
H5 an Intel CPU – the "prefetching" effect only occurs on Intel CPUs, and other CPU vendors are not affected.[13]

We test each of the above hypotheses in this section.

3.1 H1: Prefetch instruction required

The first hypothesis is that the `prefetch` instruction is necessary for the address-translation attack. The reasoning is that the instruction causes the prefetcher to start prefetching the provided address even though the permission check for this address fails. To test this hypothesis, we replaced the `prefetch` instructions with `nop`

[9] "Our attacks are based on weaknesses in the hardware design of prefetch instructions" [22].

[10] "2. Prefetch (inaccessible) address $\bar{p}$. 3. Reload p. [...] the *prefetch of $\bar{p}$ in step 2 leads to a cache hit* in step 3 with a high probability." [22] with emphasis added.

[11] "[...] delays were introduced to lower the pressure on the prefetcher." [22]. These delays were implemented using a different number of `sched_yield` system calls, as can also be seen in the original attack code [29].

[12] "Prefetch can fetch inaccessible privileged memory into various caches on Intel x86." [22] and corresponding NaCl results.

[13] "[...] we were not able to build an address-translation oracle on [ARM] Android. As the prefetch instructions do not prefetch kernel addresses [...]" [22] describing why it does not work on ARM-based Android devices.

[14] This attack is detailed in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [22] and should not be confused with the translation-level oracle described in Section 3.2 and Section 4 of the Prefetch Side-Channel Attacks paper [22].

```
; %r14 contains the direct physical address
12b6: e8 c5 fd ffff callq  1080 <sched_yield@plt>
12bb: 0f 1f 40 00     nop
12bf: 0f 1f 40 00     nop
12c3: e8 b8 fd ffff callq  1080 <sched_yield@plt>
12c8: 0f 1f 40 00     nop
12cc: 0f 1f 40 00     nop
12d0: e8 ab fd ffff callq  1080 <sched_yield@plt>
12d5: 0f 1f 40 00     nop
12d9: 0f 1f 40 00     nop
```

Listing 3: The `prefetch` instructions in the address-translation attack are replaced by 4-byte `nop`s.

instructions of the same length, as shown in Listing 3. Surprisingly, the empirical result for this modified attack is identical to the original attack: there is no change in the number of cache fetches, even though there is no `prefetch` instruction in the code. In both cases, approx. 60 cache fetches per second occur (on an i7-8700K, Ubuntu 18.10 with kernel `4.15.0-55`)[15] Hence, as the empirical result for the address-translation attack does not change with or without the `prefetch` instruction, we conclude that the `prefetch` instruction is not a requirement for the address-translation attack.[16]

3.2 H2: Values in registers required

The second hypothesis is that providing the direct-physical map address via the register is necessary. We reproduced the results from Gruss et al. [22], *i.e.*, that a virtual address stored in the register is the one fetched into the cache in the address-translation attack.

While we already excluded software prefetching as the root cause, the original code (cf. Listing 1 and the modified attack code from Listing 3) could, in fact, trigger a hardware prefetcher. There are patents describing CPUs that train a predictor whenever a register value is dereferenced to prefetch memory locations pointed to by register values ahead of time in subsequent runs, reducing instruction latency [36]. We disable all hardware prefetchers via the model-specific register `0x1a4` [97] and rerun the experiment from `H1`. In this experiment, we still observe approx. 60 cache fetches per second, *i.e.*, disabling the prefetchers has no effect. Hence, this already rules out any of the documented prefetchers as the root cause.

We run the modified address-translation attack uninterrupted and without context switches (and without `sched_yield`) on one core. In this experiment, we do not observe any cache fetches on our i7-8700K with Linux 4.15.0-55 when running this address-translation attack for 10 hours on an isolated core (*i.e.*, no interrupts). Hence, we conclude that it is not pure register loading that triggers the effect. Still, the value in the register influences what is fetched into the cache.

The registers that must be used vary across kernel versions.[17] On Ubuntu 18.10 (kernel `4.18.0-17`), we observe cache hits if the registers `r12`,`r13` and `r14` are filled. If we omit these registers, we do not observe any cache hits. On Debian 8 (kernel `4.19.28-2` and Kali Linux `5.3.9-1kali1`), the registers `r9` and `r10` cause the leakage and on Linux Mint 19 (kernel `4.15.0-52`) `rdi` and `rdx` cause the leakage. Regardless of the kernel version, we observe many cache hits when prefetching a user-space address via instruction-pointer-relative addressing, *i.e.*, the virtual address to prefetch is never in a register. However, there is no cache hit if we use an instruction-pointer-relative address pointing into the kernel address space. Similarly, when specifying the target address using an x86 complex addressing mode, we only see prefetches for user-space addresses but not for kernel-space addresses. We only confirmed leakage if absolute virtual addresses are placed in registers.

We developed a variant of the address-translation attack, which loads the address into most of the general-purpose registers. This variant consistently works across all Linux versions, even with KAISER enabled. Thus, the KAISER technique never protected against this attack. Instead, the implementation merely changed the required registers, mitigating only the specific attack implementation and attack binary. On an Intel Xeon Silver 4208 CPU, which has in-silicon patches against Meltdown [57], Foreshadow, [94] and ZombieLoad [82], we still observe about 30 cache fetches per second on Ubuntu 19.04 (kernel `5.0.0-25`).

On Windows 10 (build `1803.17134`), there is no direct physical mapping we can use to fetch addresses into the cache and verify the mapping. We fill all general-purpose registers with a kernel address and perform the syscall `SwitchToThread`. Afterwards, we perform Flush+Reload in a kernel driver to verify the speculative dereferencing in the kernel. We observe about 15 cache fetches per second for our kernel address.

3.3 H3: sched_yield required

The third hypothesis is that the `sched_yield` syscall is required for the address-translation attack to work.

The idea is that for the prefetcher to consider our prefetching hint it must not be under high pressure already. We observed in the previous experiment that omitting the `sched_yield` syscall causes the address-translation attack to fail. Hence, we run the experiment with no `sched_yield` syscalls but with a large number of context switches using interrupts, e.g., by running `stress -i` or `stress -d`. Our results show that there is indeed another source of leakage resulting in cache fetches: whilst syscall handling is a primary source of leakage, further leakage occurs due to either context switching or handling of interrupts.

We first investigate whether the `sched_yield` in the address-translation attack can be replaced by other syscalls. We discover that other syscalls e.g., `gettid`, `pipe`, `write`, expose a similar number of cache fetches. This shows that `sched_yield` can be replaced with arbitrary syscalls.

We then investigate whether there might be another leakage source, in particular whether context switches or interrupts trigger leakage. We create another experiment where one process fills the registers with a chosen address in a loop, but never performs a syscall. Another process runs Flush+Reload in a loop on this specific address. We observe about 15 cache fetches per second on this address if the process filling the registers gets interrupted

[15] We used the original code from GitHub for comparison [29] that was used to generate Figure 6 in their paper [22].
[16] To the best of our knowledge, it is required for the other attack, *i.e.*, the translation-level oracle, presented by Gruss et al. [22].
[17] The original paper describes that "delays were introduced to lower the pressure on the prefetcher" [22]. In fact, this was done via recompilation. Note that recompilation with additional code inserted may have side effects such as a different register allocation, that we analyze in this subsection.

continuously, e.g., due to NVMe interrupts, keystrokes, window events, or mouse movement.

These hits appear to be similarly caused by speculative execution in the interrupt handler. Hence, we conclude that the essential part is performing syscalls or interrupts while specific registers are filled with an attacker-chosen address.

3.4 H4: userspace_accessible bit required

The fourth hypothesis is that user-mapped kernel pages are required, *i.e.*, access is prevented via the `userspace_accessible` bit.

We constructed an experiment where we allocate several pages of memory with `mmap`. Cache lines *A* and *B* are on different pages in this `mmap`'d region. The loop (in user space) dereferences *A* and then reloads and flushes it to see whether it was cached in each loop iteration. In the last loop iteration only, we speculatively exchange the register value *A* with either the address of *B* or the direct-physical map address of *B*. Hence, both the architectural and speculative dereferences happen at the same instruction pointer value and in the same register. If we are training a hardware prefetcher based on the register values, we can expect it to prefetch *B* into the cache in the speculative run. When dereferencing *B* directly, it is usually cached after the loop when the direct-physical map address of *B* is used. However, when we dereference *A* with its value speculatively exchanged for either the address of *B* or the direct-physical map address of *B*, *B* is never cached after the final run.

When disabling interrupts, we observe no cache hits on *B* on an Intel i7-4760HQ, i7-8700K, and an AMD Phenom II 1090t. As a null hypothesis test, we perform the same test but also access *A* in the last round. We then should not see any cache hits on address *B*. And indeed, none of our CPUs fetched *B* into the cache in this scenario.

We constructed a second experiment to confirm whether the root cause of the "prefetching" effect lies in the user or kernel space. While the original address-translation attack fetches addresses in the kernel direct-physical map, we can also try to fetch user addresses. However, we discovered that this only works when SMAP is disabled (using `nosmap` kernel boot flag). Thus, the root cause of the address-translation attack is a mechanism that adheres to SMAP (supervisor-mode access prevention) and is rooted in the kernel. This also correlates with the finding of Kocher et al. [49] that speculative execution cannot bypass SMAP. Hence, we can conclude that the root cause is some form of code execution in the kernel.

3.5 H5: Effect only on Intel CPUs

The fifth hypothesis is that the "prefetching" effect only occurs on Intel CPUs. We assume that all types of CPUs vulnerable to Spectre are also affected by the speculative dereferencing in the kernel [49].

Thus, we evaluate the same experiment explained in Section 3.4 on an AMD Ryzen Threadripper 1920X (Ubuntu 17.10, `4.13.0-46-generic`), an ARM Cortex-A57 (Ubuntu 16.04.6 LTS, `4.4.38-tegra`) and an IBM Power9 (Ubuntu 18.04, `4.15.0-29`). On the AMD Ryzen Threadripper 1920X, we achieve up to 20 speculative fetches per second. There, we observed a cache hit rate of 0.0004% on *B*, which is the standard false positive rate we observed for Flush+Reload attacks on this CPU. On the Cortex-A57, we observe 5 speculative

```
1 ;<do_syscall_64+106>
2 => 0xffffffff8100134a: callq 0xffffffff81802000
3 => 0xffffffff81802000: jmpq   *%rax
4 ; with retpoline
5 => 0xffffffff81802000:  callq  0xffffffff8180200c
6 => 0xffffffff8180200c:  mov    %rax,(%rsp)
7 => 0xffffffff81802010:  retq
```

Listing 4: While processing a syscall, the kernel performs multiple indirect jumps, e.g., one to the corresponding syscall handler. With retpoline [90], the kernel uses a `retq` for the indirect jump. Without retpoline the `jmp` instruction is used on a pointer in a register.

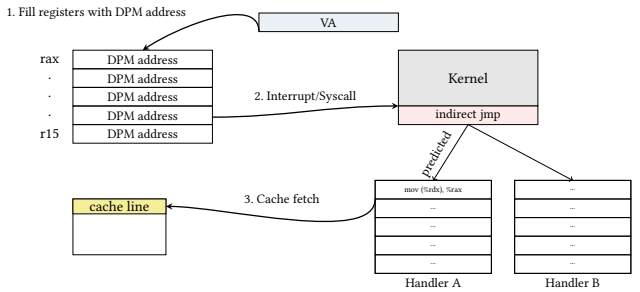

Figure 2: The kernel speculatively dereferences the direct-physical map address (DPM). With Flush+Reload, we observe cache hits on the corresponding user-space address.

fetches per second, and on the IBM Power9, we detect up to 15 speculative fetches per second. We do not observe any false positives on the ARM and Power9 CPUs during this experiment.

We run the same experiment on a Raspberry Pi 3 (ARM Cortex-A53,Ubuntu 18.04, kernel `4.15.0`), an in-order CPU with no branch prediction [4]. Thus, this CPU is not susceptible to any Spectre-type attacks. Running the same code for 1 hour, we do not observe any cache fetches. Therefore, as no leakage appears on an in-order CPU without branch prediction, the effect must be related to Spectre. The hypothesis that the effect is hardware-specific to Intel CPUs is incorrect; any CPU susceptible to Spectre-BTB is vulnerable to speculative dereferencing in the kernel if the mitigations are not enabled.

3.6 Speculative Execution in the Kernel

From the previous analysis of the hypotheses, we can conclude that the leakage is not due to the software or hardware prefetchers but due to speculative code execution in the kernel. While this conclusion might not be suprising with the knowledge of Spectre, Spectre was only discovered one year after the original prefetch paper [22] was published. We now show that the primary leakage is caused by Spectre-BTB-SA-IP (branch target buffer, training in same address space, and in-place) [11].

First, we observe that during a syscall, the kernel performs multiple indirect jumps to execute the corresponding system-call handler (cf. Listing 4). With retpoline, the kernel uses a `retq` for the indirect jump, which traps the speculative execution path to a fixed branch. Without retpoline, the `jmp` instruction is used on a pointer

in a register. This causes speculative execution based on Spectre-BTB-SA-IP. The address-translation attack then succeeds because different syscalls use a different number of arguments. The unified interface does not zero out registers that a given syscall does not require. Consequently, during speculative execution, the CPU might use an incorrect prediction from the branch-target buffer (BTB) and speculate into the wrong syscall. Figure 2 illustrates the speculative execution in the kernel dereferencing. In this misspeculated syscall, registers containing attacker-chosen addresses are used. This can either be because the registers were never initialized and instead still contain the attacker-chosen addresses, or because they are deliberately initialized to attacker-chosen addresses through the syscall entry code.

We evaluate the leakage rate of other syscalls and the impact of mistraining the branch prediction mechanisms in Section 4. On recent kernels, the leakage completely disappears unless `nospectre_v2` (*i.e.*, disable Spectre-BTB countermeasures) is passed as a boot flag. Disabling the Spectre V2 mitigations is interesting for cloud computing since the mitigations introduce a big performance overhead [87]. Thus, the address-translation attack is mitigated using the Spectre-BTB countermeasures and not, as described in previous work [21, 22], by KAISER (KPTI) [21], or LAZARUS [19].

We observed other speculative execution in the kernel that exposes the same effects. However, we observe 15 speculative fetches per second on an i5-8250U (kernel `5.0.0-20`) if we eliminate the Spectre-BTB-SA-IP leak from Listing 4, empirically confirming that this is one of the main leakage sources. As already mentioned, there are further Spectre gadgets in the interrupt handling.

As Canella et al. [11] showed, there were about 172 unmitigated Spectre v1 "prefetch" gadgets found in the Linux kernel. These gadgets enable the same attacks as presented in this paper. Currently there is no consistent plan to mitigate these gadgets. However, any prefetch gadget can be used for an address-translation attack [22] and thus would also re-enable Foreshadow-VMM attacks [94, 98]. As concurrent work showed, there are gadgets in the Linux kernel which can be used to fetch data into the L1D cache in Xen [100] and an artificial gadget was exploited by Stecklina [88].

In the case of interrupts, we analyzed the interrupt handling in the Linux kernel version 4.19.0 and observed that the register values from `r8-r15` are cleared but stored on the stack and restored after the interrupt. Thus, either there is a misspeculation on old register values, or the leakage comes from the stored stack values [60]. Additionally, we found several `jmp` instructions that occur in the analyzed instruction trace, which might trigger speculative cache fetches. Again, when using the Spectre-BTB mitigations we could not detect any leakage while triggering interrupts, showing that this is a crucial element for the speculative dereferencing.

3.7 Meltdown-L3 and Foreshadow-L3

The speculative dereferencing was also noticed but misattributed to the prefetcher in subsequent work. For instance, the Meltdown paper [57] reports that data is fetched from L3 into L1 while mounting a Meltdown attack. Van Bulck et al. [94] did not observe this prefetching effect for Foreshadow. Based on this observation, further works also mentioned this effect without analyzing it thoroughly [11, 68, 96]. In SpeechMiner the explanation provided is that performing a Meltdown-US attack causes data to be repeatedly prefetched from L1 to L3 [103].

We used similar Meltdown-L3 setups as SpeechMiner [103] and Meltdown [57]. For this purpose, we contacted the authors to ask for their specific experiment setup. According to the authors of SpeechMiner [103], the kernel boot flags `nopti`, `nokaslr` were used on kernel 4.4.0-134. We used Ubuntu 16.04 on an Intel i7-6700K to reproduce the attack. The authors of Meltdown used Ubuntu 16.10 (kernel 4.8.0), which at that moment of writing did not have any mitigations against Spectre at all [57].

We construct our Meltdown-L3 experiment as follows. One physical core constantly accesses a secret to ensure that the value stays in the L3, as the L3 is shared across all physical cores. On a different physical core, we run Meltdown on the direct-physical map. On recent Linux kernels with full Spectre v2 mitigations implemented, we could not reproduce the result on the same machine with the default mitigations enabled. With the `nospectre_v2` flag, our Meltdown-L3 attack works again when triggering the prefetch gadget in the kernel. Since we run Meltdown on the direct-physical map, we place the corresponding direct-physical map address in a register. Now, when a syscall is performed, or an interrupt is triggered, the direct-physical map address is speculatively dereferenced, causing the data to be fetched into L1.

Concluding the above experiment, on Linux kernels 4.4.0-137 and 4.8, as respectively used in SpeechMiner [103] and Meltdown [57], not all Spectre-BTB mitigations such as IBPB and RSB stuffing were implemented. Thus, the Meltdown-L3 prefetching works because these mitigations are not implemented on these kernel versions [59]. Without our new insights that the prefetching effect is caused by speculative execution, it is almost inevitable to not misdesign these experiments, inevitably leading to incomplete or incorrect observations and conclusions on Meltdown and Foreshadow and their mitigations. We confirmed with the authors that their experiment design was not robust to our new insight and therefore lead to wrong conclusions.

`Foreshadow-L3`, The same prefetching effect can be used to perform Foreshadow [94]. If a secret is present in the L3 cache and the direct-physical map address is derefenced in the hypervisor kernel, data can be fetched into the L1. This reenables Foreshadow even with Foreshadow mitigations enabled if the unrelated Spectre-BTB mitigations are disabled. We demonstrate this attack in KVM in Section 6.

In Meltdown and Foreshadow, as in other transient-execution attacks, common implementations transmit a secret byte from the transient-execution realm via a Flush+Reload cache covert channel to the architectural realm. Most implementations transmit 1 byte of data by accessing one of 256 offsets in an array. Several papers, including Meltdown and Foreshadow, observed a bias towards the '0' index, where a secret value of '0' is falsely reported to the attacker. This effect was observed and explained by the zeroing of invalid loads [57, 94]. We also tried to reproduce these results. However, we only observed a bias towards zero on systems with hardware mitigations against Meltdown and Foreshadow, which by design return zeros in these attack scenarios [10]. We observed no bias towards zero on other systems with the most recent software patches and software mitigations. To transmit a value of '0' through the Flush+Reload covert channel, the offset '0' is accessed, *i.e.*, the array base

Table 1: Evaluated systems, their CPUs, operating systems, and kernel versions used in the syscall evaluation.

CPU	Operating System	Kernel
Intel i5-8250U	Linux Mint 19	`4.15.0-52`
Intel i7-8700K	Ubuntu 18.04	`4.15.0-55`
ARM Cortex-A57	Ubuntu 16.04.6	`4.4.38-tegra`
AMD Threadripper 1920X	Ubuntu 17.10	`4.13.0-46`

address. However, the Flush+Reload array base address is stored in a register during the Flush+Reload loop. Thus, the base address is speculatively dereferenced due to interrupts and the `sched_yield` found in the Flush+Reload loops in these implementations. This indicates that the speculative dereferencing of user-space registers creates at least part of the zero bias, if not all, since the bias is no longer visible on more recent systems with full software mitigations against Spectre enabled.

4 IMPROVING THE LEAKAGE RATE

With the knowledge that the root cause of the prefetching effect is speculative execution in the kernel, we can try to optimize the number of cache fetches. As already observed in Section 3.3, the `sched_yield` syscall can be replaced by an arbitrary syscall to perform the address-translation attack. In this section, we compare different syscalls and their impact on the number of speculative cache fetches on different architectures and kernel versions. We investigate the impact of executing additional syscalls before and after the register filling and measure their effects on the number of speculative cache fetches.

Setup. Table 1 lists the test systems used in our experiments. On the Intel and AMD CPUs, we disabled the Spectre-BTB mitigations using the kernel flag `nospectre_v2`. On the evaluated ARM CPU, Spectre-BTB mitigations are not supported by the tested firmware. We evaluate the speculative dereferencing using different syscalls to observe whether the number of cache fetches increases. Based on the number of correct and incorrect cache fetches of two virtual addresses, we calculate the F1-score, *i.e.*, the harmonic average of precision and recall.

When performing a syscall, the CPU might mispredict the target syscall based on the results of the BTB. If a misprediction occurs, another syscall which dereferences the values of user-space registers might be speculatively executed. Therefore if we perform syscalls before we fill the registers with the direct-physical map address, we might mistrain the BTB and trigger the CPU to speculatively execute the mistrained syscall. We evaluate the mistraining of the BTB for `sched_yield` in Appendix A.

We create a framework that runs the experiment from Section 3.4 with 20 different syscalls (after filling the registers) and computes the F1-score. We perform different syscalls before filling the registers to mistrain the branch prediction. One direct-physical-map address has a corresponding mapping to a virtual address and should trigger speculative fetches into the cache. The other direct-physical-map address should not produce any cache hits on the same virtual address. If there is a cache hit on the correct virtual address, we count it as a true positive. Conversely, if there is no hit when there should have been one, we count it as a false negative. On the second address, we count the false positives and true negatives. For syscalls with parameters, e.g., `mmap`, we set the value of all parameters to the direct-physical-map address, *i.e.*, `mmap(addr, addr, addr, addr, addr, addr)`. We repeat this experiment 1000 times for each syscall on each system and compute the F1-Score.

Table 2: F1-Scores for speculative cache fetches with different syscalls on different CPU architectures.

Syscall	Syscall executed before	i5-8250U	i7-8700K	Threadripper 1920X	Cortex-A57
sched_yield	None	66.40%	91.49%	99.29%	76.61%
	send-to	56.42%	4.60%	52.94%	44.88%
	geteuid	46.62%	1.90%	63.94%	48.82%
	stat	77.37%	57.44%	69.28%	63.57%
pipe	None	100%	99.35%	100%	100%
	send-to	99.9%	99.60%	100%	100%
	geteuid	99.9%	99.61%	100%	100%
	stat	99.9%	99.55%	99.9%	100%
read	None	10.42%	0.09%	8.50%	57.95%
	send-to	14.47%	21.26%	1.90%	78.86%
	geteuid	15.32%	56.73%	2.35%	73.73%
	stat	28.32%	24.07%	9.70%	23.32%
write	None	7.69%	91.24%	76.46%	58.95%
	send-to	14.29%	9.88%	11.00%	45.68%
	geteuid	15.49%	32.21%	52.94%	49.47%
	stat	9.16%	9.70%	52.83%	12.03%
nanosleep	None	21.2%	27.43%	52.61%	87.40%
	send-to	46.59%	13.43%	76.23%	82.83%
	geteuid	29.93%	96.05%	89.62%	69.63%
	stat	59.84%	99.14%	89.68%	77.67%

Evaluation. We evaluate different syscalls for branch prediction mistraining by executing a single syscall before and after filling the registers with the target address. Table 2 lists the F1-scores of syscalls which achieved the highest number of cache fetches after filling registers with addresses. The results show that the same effects occur on both AMD and ARM CPUs, with similar F1-scores.

Executing the `pipe` syscall after filling the register seems to always trigger speculative dereferencing in the kernel on each architecture. However, this syscall has to perform many operations and takes 3 to 5 times longer to execute than `sched_yield`. On recent Linux kernels (version 5), we observe that the number of cache fetches decreases. This is due to a change in the implementation of the syscall handler, and thus other paths need to be executed to increase the probability of speculative dereferencing. We observe that an additional, different, syscall executed before filling the registers also mistrains the branch prediction. Thus, we also compare the number of cache fetches with an additional syscall added before the registers are filled. If we add additional syscalls like `stat`, `sendto`, or `geteuid` before filling the registers, we achieve higher F1-scores in some cases. For instance, executing the syscalls `read` and `nanosleep` after the register filling performs significantly better (up to 80% higher F1-scores) with prior syscall mistraining. However, as listed in Table 2, not every additional syscall increases the number of cache fetches.

5 COVERT CHANNEL

For the purpose of a systematic analysis, we evaluate the capacity of our discovered information leakage by building a covert channel. Note that while covert channels assume a colluding sender and receiver, it is considered best practice to evaluate the maximum performance of a side channel by building a covert channel. Similar to previous works [75, 99], our covert channel works without shared memory and across CPU cores. The capacity of the covert channel indicates an upper bound for potential attacks where the attacker and victim are not colluding.

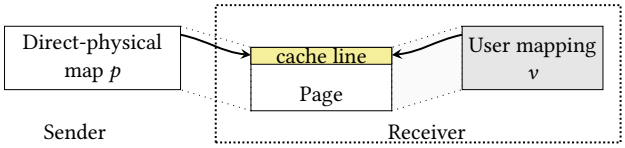

Figure 3: The setup for the covert channel. The receiver allocates a page accessible through the virtual address v. The sender uses the direct-physical mapping p of the page to influence the cache state.

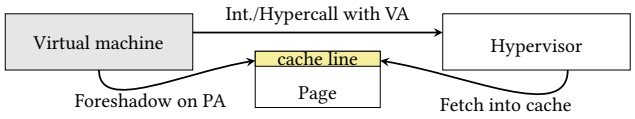

Figure 4: If a guest-chosen address is speculatively fetched into the cache during a hypercall or interrupt and not flushed before the virtual machine is resumed, the attacker can perform a Foreshadow attack to leak the fetched data.

Setup. Figure 3 shows the covert-channel setup. The receiver allocates a memory page which is used for the communication. The receiver can access the page through the virtual address v. Furthermore, the receiver retrieves the direct-physical-map address p of this page. This can be done, *i.e.*, using the virtual-to-physical address-translation technique we analyzed in Section 3. The address p is used by the sender to transmit data to the receiver. The address p also maps to the page, but as it is a kernel address, a user program cannot access the page via this virtual address. The direct-physical-map address p is a valid kernel address for every process. Moreover, as the shared last-level cache is physically indexed and physically tagged, it does not matter for the cache which virtual address is used to access the page.

Transmission. The transmitted data is encoded into the cache state by either caching a cache line of the receiver page ('1'-bit) or not caching the cache line of the receiver page ('0'-bit). To cache a cache line of the receiver page, the sender uses Spectre-BTB-SA-IP in the kernel to speculatively access the kernel address p. For this, the sender constantly fills all x86-64 general-purpose registers with the kernel address p and performs a syscall. The kernel address is then speculatively dereferenced in the kernel and the CPU caches the chosen cache line of the receiver page. Hence, we can use this primitive to transmit one bit of information. To synchronize the two processes, we define a time window per bit for sender and receiver. On the receiver side, we reaccess the same cache line to check whether the address v, *i.e.*, the first cache line of the receiver page, is cached. After the access, the receiver flushes the address v to repeat the measurement. A cache hit is interpreted as a '1'-bit. Conversely, if the sender wants to transmit a '0'-bit, the sender does not write the value into the registers and instead waits until the time window is exceeded. Thus, if the receiver encounters a cache miss, it is interpreted as a '0'-bit.

Evaluation. We evaluated the covert channel by transmitting random messages between two processes running on different physical CPU cores. Our test system was equipped with an Intel i7-6500U CPU, running Linux Mint 19 (kernel `4.15.0-52-generic`, `nospectre_v2` boot flag).

In our setup, we transmit 128 bytes from the sender to the receiver and run the experiment 50 times. We observed that additional interrupts on the core where the syscall is performed increases the performance of the covert channel. These interrupts trigger the speculative execution we observed in the interrupt handler. In particular, I/O interrupts, *i.e.*, syncing the NVMe device, create additional cache fetches. While we achieved a transmission rate of up to 30 bit/s, at this rate we had a high standard error of approx. 1%. We achieved the highest capacity at a transmission rate of 10 bit/s. At this rate, the standard error is, on average, 0.1%. This result is comparable to related work in similar scenarios [75, 99]. To achieve an error-free transmission, error-correction techniques [62] can be used. Compared to to the Flush+Prefetch covert channel demonstrated by Gruss et al. [22] is that our covert channel does not require any shared memory. Thus, while slower, it is more powerful as it can be used in a wider range of scenarios.

6 SPECULATIVE DEREFERENCES AND VIRTUAL MACHINES

In this section, we examine speculative dereferencing in virtual machines. We demonstrate a successful end-to-end attack using interrupts from a virtual-machine guest running under KVM on a Linux host [15]. The attack succeeds even with the recommended Foreshadow mitigations enabled, provided that the unrelated Spectre-BTB mitigations are disabled. Against our expectations, we did not observe any speculative dereferencing of guest-controlled registers in Microsoft's Hyper-V HyperClear Foreshadow mitigation. We provide a thorough analysis of this negative result.

Since we observe speculative dereferencing in the syscall handling, we investigate whether hypercalls trigger a similar effect. The attacker targets a specific host-memory location where the host virtual address and physical address are known but inaccessible.

Foreshadow Attack on Virtualization Software. If an address from the host is speculatively fetched into the L1 cache on a hypercall from the guest, we expect it to have a similar speculative-dereferencing effect. With the speculative memory access in the kernel, we can fetch arbitrary memory from L2, L3, or DRAM into the L1 cache. Consequently, Foreshadow can be used on arbitrary memory addresses provided the L1TF mitigations in use do not flush the entire L1 data cache [88, 91, 100]. Figure 4 illustrates the attack using hypercalls or interrupts and Foreshadow. The attacking guest loads a host virtual address into the registers used as hypercall parameters and then performs hypercalls. If there is a prefetching gadget in the hypercall handler and the CPU misspeculates into this gadget, the host virtual address is fetched into the cache. The attacker then performs a Foreshadow attack and leaks the value from the loaded virtual address.

6.1 Foreshadow on Patched Linux KVM

Concurrent work showed that prefetching gadgets in the kernel, in combination with L1TF, can be exploited on Xen and KVM [88, 100]. The default setting on Ubuntu 19.04 (kernel `5.0.0-20`) is to only conditionally flush the L1 data cache upon VM entry via KVM [91], which is also the case for Kali Linux (kernel `5.3.9-1kali1`). The L1

data cache is only flushed in nested VM entry scenarios or in situations where data from the host might be leaked. Since Linux kernel `4.9.81`, Linux's KVM implementation clears all guest clobbered registers to prevent speculative dereferencing [16]. In our attack, the guest fills all general-purpose registers with direct-physical-map addresses from the host.

End-To-End Foreshadow Attack via Interrupts. In Section 3.3, we observed that context switches triggered by interrupts can also cause speculative cache fetches. We use the example from Section 3.3 to verify whether the "prefetching" effect can also be exploited from a virtualized environment. In this setup, we virtualize Linux buildroot (kernel `4.16.18`) on a Kali Linux host (kernel `5.3.9-1kali1`) using qemu (`4.2.0`) with the KVM backend. In our experiment, the guest constantly fills a register with a direct-physical-map address and performs the `sched_yield` syscall. We verify with Flush+Reload in a loop on the corresponding host virtual address that the address is indeed cached. Hence, we can successfully fetch arbitrary hypervisor addresses into the L1 cache on kernel versions before the patch, *i.e.*, with Foreshadow mitigations but incomplete Spectre-BTB mitigations. We observe about 25 speculative cache fetches per minute using NVMe interrupts on our Debian machine. The attacker, running as a guest, can use this gadget to prefetch data into the L1. Since data is now located in the L1, this reenables a Foreshadow attack [94], allowing guest-to-host memory reads. As described before, 25 fetches per minute means that we can theoretically leak up to $64 \cdot 25 = 1600$ bytes per minute (or 26.7 bytes per second) with a Foreshadow attack despite mitigations in place. However, this requires a sophisticated attacker who avoids context switches once the target cache line is cached.

We develop an end-to-end Foreshadow-L3 exploit that works despite enabled Foreshadow mitigations, provided the unrelated Spectre-BTB mitigations are disabled. In this attack the host constantly accesses a secret on a physical core, which ensures it remains in the shared L3 cache. We assign one isolated physical core, consisting of two hyperthreads, to our virtual machine. In the virtual machine, the attacker fills all registers on one logical core (hyperthread) and performs the Foreshadow attack on the other logical core. Note that this is different from the original Foreshadow attack where one hyperthread is controlled by the attacker and the sibling hyperthread is used by the victim. Our scenario is more realistic, as the attacker controls both hyperthreads, *i.e.*, both hyperthreads are in the same trust domain. With this proof-of-concept attack implementation, we are able to leak 7 bytes per minute successfully [18]. Note that this can be optimized further, as the current proof-of-concept produces context switches regardless of whether the cache line is cached or not. Our attack clearly shows that the recommended Foreshadow mitigations alone are not sufficient to mitigate Foreshadow attacks, and Spectre-BTB mitigations must be enabled to fully mitigate our Foreshadow-L3 attack.

No Prefetching gadget in Hypercalls in KVM We track the register values in hypercalls and validate whether the register values from the guest system are speculatively fetched into the cache. We neither observe that the direct-physical-map address is still located in the registers nor that it is speculatively fetched into the cache. However, as was shown in concurrent work [88, 100], prefetch gadgets exist in the kernel that can be exploited to fetch data into the cache, and these gadgets can be exploited using Foreshadow.

[18] An anonymized demonstration video can be found here: https://streamable.com/8ke5ub

6.2 Negative Result: Foreshadow on Hyper-V HyperClear

We examined whether the same attack also works on Windows 10 (build `1803.17134`), which includes the latest patch for Foreshadow. As on Linux, we disabled the mitigations for Spectre-BTB and tried to fetch hypervisor addresses from guest systems into the cache.

Microsoft's Hyper-V HyperClear Mitigation [65] for Foreshadow claims to only flush the L1 data cache when switching between virtual cores. Hence, it should be susceptible to the same basic attack we described at the beginning of this section. For our experiment, the attacker passes a known virtual address of a secret variable from the host operating system for all parameters of a hypercall. However, we could not find any exploitable timing difference after switching from the guest to the hypervisor. Our experiments concerning this negative result are discussed in Appendix C.

7 LEAKING VALUES FROM SGX REGISTERS

In this section, we present a novel method, *Dereference Trap*, to leak register contents from an SGX enclave in the presence of only a speculative register dereference. We show that this technique can also be generalized and applied to other contexts. Leaking the values of registers is useful, e.g., to extract parts of keys or intermediate values from cryptographic operations. While there are already Spectre attacks on SGX enclaves [12, 69], they require the typical Spectre-PHT gadget [49], *i.e.*, a double indirect memory access after a conditional branch.

7.1 Dereference Trap

For *Dereference Trap*, we exploit transient code paths inside an enclave which speculatively dereference a register containing a secret value. The setup is similar to the kernel case we examined in Section 3.6. An SGX enclave has access to the entire virtual address space [37]. Hence, any speculative memory access to a valid virtual address caches the data at this address.

The basic idea of *Dereference Trap* is to ensure that the entire virtual address space of the application is mapped. Thus, if a register containing a secret is speculatively dereferenced, the corresponding virtual address is cached. The attacker can detect which virtual address is cached and infer the secret. However, in practice, there are two main challenges which must be resolved to implement *Dereference Trap*. Firstly, the virtual address space is much larger than the physical address space. Thus it is not possible to simply map all virtual addresses to physical addresses. Secondly, the Flush+Reload attack is a bottleneck, as even a highly-optimized Flush+Reload attack takes around 300 CPU cycles [80]. Hence, probing every cache line of the entire user-accessible virtual address space of 2^{47} bytes would require around 2 days on a 4 GHz CPU. Moreover, probing this many cache lines does not work as the cached address does not remain in the cache if many other addresses are accessed.

Divide and Conquer. Instead of mapping every page in the virtual address space to its own physical pages, we only map 2 physical pages $p1$ and $p2$, as illustrated in Figure 5. By leveraging shared

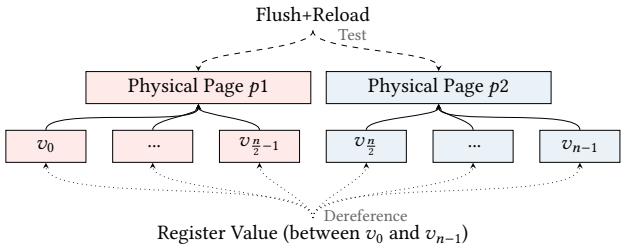

Figure 5: Leaking the value of an x86 general-purpose register using *Dereference Trap* and Flush+Reload on two different physical addresses. v_0 to v_{n-1} represent the memory mappings on one of the shared memory regions.

```
1  class Object {
2  public:
3    virtual void print() = 0;
4  };
5  class Dummy : public Object {
6  private:
7    char* data;
8  public:
9    Dummy() { data = "TEST"; }
10   virtual void print() { puts(data); }
11 };
12 class Secret : public Object {
13 private:
14   size_t secret;
15 public:
16   Secret() { secret = 0x12300000; }
17   virtual void print() { }
18 };
19 void printObject(Object* o) { o->print(); }
```

Listing 5: Speculative type confusion which leaks the secret of `Secret` class instances using *Dereference Trap*.

memory, we can map one physical page multiple times into the virtual address space. By default, the number of `mmap`ed segments which can be mapped simultaneously is limited to 65 536 [45]. However, as the attacker in the SGX threat model is privileged [37] we can easily disable this limit. The maximum allowed value is $2^{31}-1$, which makes it possible to map $1/16^{th}$ of the user-accessible virtual address space. If we only consider 32-bit secrets, *i.e.*, secrets which are stored in the lower half of 64-bit registers, 2^{20} mappings are sufficient. Out of these, the first 2^{10} virtual addresses map to physical page $p1$ and the second 2^{10} addresses map to page $p2$. Consequently the majority of 32-bit values are now valid addresses that either map to $p1$ or $p2$. Thus, after a 32-bit secret is speculatively dereferenced inside the enclave, the attacker only needs to probe the 64 cache lines of each of the two physical pages. A cache hit reveals the most-significant bit (bit 31) of the secret as well as bits 6 to 11, which define the cache-line offset on the page.

To learn the remaining bits 12 to 30, we continue in a fashion akin to binary-search. We unmap all mappings to $p1$ and $p2$ and create half as many mappings as before. Again, half of the new mappings map to $p1$ and half of the new mappings map to $p2$. From a cache hit in this setup, we can again learn one bit of the secret. We can repeat these steps until all bits from bit 6 to 31 of the secret are known. As the granularity of Flush+Reload is one cache line, we cannot leak the least-significant 6 bits of the secret.

As a privileged attacker, we can also disable the hardware prefetchers on Intel CPUs by setting the model-specific register `0x1a4` to 15 [97]. This prevents spurious cache hits, which is especially important for probing the cache lines on a single page.

We evaluated *Dereference Trap* on our test system and recovered a 32-bit value stored in a 64-bit register within 15 minutes.

7.2 Speculative Type Confusion

SGX registers are invisible to the kernel and can thus not be speculatively dereferenced from outside SGX. Hence, the dereference gadget has to be inside the enclave. While there is a mechanism similar to a context switch when an enclave is interrupted, we could not find such a gadget in either the current SGX SDK or driver code. This is unsurprising, as this code is hardened with memory fences for nearly all memory loads to prevent LVI [95] as well as other transient-execution attacks.

Hence, to leak secret registers using *Dereference Trap*, the gadget must be in the enclave code. Such a gadget can easily be introduced, e.g., when using polymorphism in C++. Listing 5 shows a minimal example of introducing such a gadget.

The virtual functions are implemented using *vtable*s for which the compiler emits an indirect call in Line 19. The branch predictor for this indirect call learns the last call target. Thus, if the call target changes because the type of the object is different, speculative execution still executes the function of the last object with the data of the current object.

In this code, calling `printObject` first with an instance of `Dummy` mistrains the branch predictor to call `Dummy::print`, dereferencing the first member of the class. A subsequent call to `printObject` with an instance of `Secret` leads to speculative execution of `Dummy::print`. However, the dereferenced member is now the secret (Line 16) of the `Secret` class.

The speculative type confusion in such a code construct leads to a speculative dereference of a value which would never be dereferenced architecturally. We can leak this speculatively dereferenced value using the *Dereference Trap* attack.

However, there are also many different causes for such gadgets [32], e.g., function pointers or (compiler-generated) jump tables.

7.3 Generalization of Dereference Trap

Dereference Trap is a generic technique which also applies to any other scenario where the attacker can set up the hardware and address space accordingly. *Dereference Trap* applies to all Spectre variants. Thus, Spectre-v2 mitigations alone are not sufficient to hinder *Dereference Trap*. Many in-place Spectre-v1 gadgets that are not the typical encoding array gadget are still entirely unprotected with no plans to change this. For instance, Intel systems before Haswell and AMD systems before Zen do not support SMAP. Also, more recent systems may have SMAP disabled. On these systems, we can also `mmap` memory regions and the kernel will dereference 32-bit values misinterpreted as pointers (into user space). We prepared an experiment where a kernel module speculatively accesses a secret value. The user-space process performs the *Dereference Trap*.

Using this technique the attacker can reliably leak a 32-bit secret which is speculatively dereferenced by the kernel module using an artificial Spectre gadget. Cryptographic implementations often store keys in the lower 32 bits of 64bit registers (OpenSSL AES round key u32 *rk; for instance) [70]. Hence, those implementations might be susceptible to *Dereference Trap*.

We evaluated the same experiment on an Intel i5-8250U, ARM Cortex-A57, and AMD ThreadRipper 1920X with the same result of 15 minutes to recover a 32-bit secret. Thus, Spectre-BTB mitigations and SMAP must remain enabled to mitigate attacks like *Dereference Trap*.

8 LEAKING PHYSICAL ADDRESSES FROM JAVASCRIPT USING WEBASSEMBLY

In this section, we present an attack that leaks the physical address (cache-line granularity) of a variable from within a JavaScript context. Our main goal is to show that the "prefetching" effect is much simpler than described in the original paper [22], *i.e., it does not require native code execution*. The only requirement for the environment is that it can keep a 64-bit register filled with an attacker-controlled 64-bit value.

In contrast to the original paper's attempt to use NaCl to run in native code in the browser, we describe how to create a JavaScript-based attack to leak physical addresses from Javascript variables and evaluate its performance in common JavaScript engines and Firefox. We demonstrate that it is possible to fill 64-bit registers with an attacker-controlled value in JavaScript by using WebAssembly.

Attack Setup. JavaScript encodes numbers as double-precision floating-point values in the IEEE 754 format [66]. Thus, it is not possible to store a full 64-bit value into a register with vanilla JavaScript, as the maximum precision is only 53-bit. The same is true for BigInteger libraries, which represent large numbers as structures on the heap [92]. To overcome this limitation, we leverage WebAssembly, a binary instruction format which is precompiled for the JavaScript engine and not further optimized by the engine [92]. The precompiled bytecode can be loaded and instantiated in JavaScript. To prevent WebAssembly from harming the system, the bytecode is limited to calling functions provided by the JavaScript scope.

Our test operating system is Debian 8 (kernel`5.3.9-1kali1`) on an Intel i7-8550U. We observe that on this system registers `r9` and `r10` are speculatively dereferenced in the kernel. In our attack, we focus on filling these specific registers with a guessed direct-physical-map address of a variable. The WebAssembly method `load_pointer` of Listing 6 (Appendix B) takes two 32-bit JavaScript values, which are combined into a 64-bit value and populated into as many registers as possible. To trigger interrupts we rely on web requests from JavaScript, as suggested by Lipp et al. [55].

We can use our attack to leak the direct-physical-map address of any variable in JavaScript. The attack works analogously to the address-translation attack in native code [22].

(1) Guess a physical address p for the variable and compute the corresponding direct-physical map address $d(p)$.
(2) Load $d(p)$ into the required registers (`load_pointer`) in an endless loop, e.g., using endless-loop slicing [55].
(3) The kernel fetches $d(p)$ into the cache when interrupted.
(4) Use Evict+Reload on the target variable. On a cache hit, the physical address guess p from Step 1 was correct. Otherwise, continue with the next guess.

Attack from within Browsers. Before evaluating our attack in an unmodified Firefox browser, we evaluate our experiment on the JavaScript engines V8 version 7.7 and Spidermonkey 60. To verify our experiments, we use Kali Linux (kernel `5.3.9-1kali1`) running on an Intel i7-8550U. As it is the engines that execute our WebAssembly, the same register filling behavior as in the browser should occur when the engines are executed standalone. In both engines, we use the C-APIs to add native code functions [67, 93], enabling us to execute syscalls such as `sched_yield`. This shortcuts the search to find JavaScript code that constantly triggers syscalls. Running inside the engine with the added syscall, we achieve a speed of 20 speculative fetches per second. In addition to testing in the standalone JavaScript engines, we also show that speculative dereferencing can be triggered in the browser. We mount an attack in an unmodified Firefox 76.0 by injecting interrupts via web requests. We observe up to 2 speculative fetches per hour. If the logical core running the code is constantly interrupted, e.g., due to disk I/O, we achieve up to 1 speculative fetch per minute. As this attack leaks parts of the physical and virtual address, it can be used to implement various microarchitectural attacks [20, 23, 49, 71, 75, 79, 83]. Hence, the address-translation attack is possible with JavaScript and WebAssembly, without requiring the NaCl sandbox as in the original paper [22].

Upcoming JavaScript extensions expose syscalls to JavaScript [13]. However, at the time of writing, no such extensions are enabled by default. Hence, as the second part of our evaluation, we investigate whether a syscall-based attack would also yield the same performance as in native code. To simulate the extension, we expose the `sched_yield` syscall to JavaScript. We observe the same performance of 20 speculative fetches per second with the syscall function. Thus, new extensions for JavaScript may improve the performance of our previously described attack on unmodified Firefox.

Limitations of the Attack. We conclude that the bottleneck of this attack is triggering syscalls. In particular, there is currently no way to directly perform a single syscall via Javascript in browsers without high overhead. We traced the syscalls of Firefox using `strace`. We observed that syscalls such as `sched_yield, getpid, stat, sendto` are commonly performed upon `window` events, e.g., opening and closing pop-ups or reading and writing events on the JavaScript console. However, the registers `r9` and `r10` get overwritten before the syscall is performed. Thus, whether the registers are speculatively dereferenced while still containing the attacker-chosen values strongly depends on the engine's register allocation and on other syscalls performed. As Jangda et al. [42] stated, not all registers are used in Chrome and Firefox in the JIT-generated native code. Not all registers can be filled from within the browser, e.g., Chrome uses the registers `r10` and `r13` only as scratch registers, and Firefox uses `r15` as the heap pointer [42].

9 DISCUSSION

The "prefetching" of user-space registers was first observed by Gruss et al. [22] in 2016. In May 2017, Jann Horn discovered that

speculative execution can be exploited to leak arbitrary data. In January 2018, pre-prints of the Spectre [49] and Meltdown [57] papers were released. Our results indicate that the address-translation attack was the first inadvertent exploitation of speculative execution, albeit in a much weaker form where only metadata, *i.e.*, information about KASLR, is leaked rather than real data as in a full Spectre attack. Even before the address-translation attack, speculative execution was well known [77] and documented [35] to cause cache hits on addresses that are not architecturally accessed. This was often mentioned together with prefetching [28, 104]. Currently, the address-translation attack and our variants are mitigated on both Linux and Windows using the retpoline technique to avoid indirect branches. In particular, the Spectre-BTB gadget in the syscall wrapper can be fixed by using the `lfence` instruction.

Another possibility upon a syscall is to save user-space register values to memory, clear the registers to prevent speculative dereferencing, and later restore the user-space values after execution of the syscall. However, as has been observed in the interrupt handler, there might still be some speculative cache accesses on values from the stack. The retpoline mitigation for Spectre-BTB introduces a large overhead for indirect branches. The performance overhead can in some cases be up to 50 % [87]. This is particularly problematic in large scale systems, e.g., cloud data centers, that have to compensate for the performance loss and increased energy consumption. Furthermore, retpoline breaks CET and CFI technologies and might thus also be disabled [8]. As an alternative, randpoline [8] could be used to replace the mitigation with a probabilistic one, again with an effect on Foreshadow mitigations. And indeed, mitigating memory corruption vulnerabilities may be more important than mitigating Foreshadow in certain use cases. Cloud computing concepts that do not rely on traditional isolation boundaries are already being explored in industry [2, 14, 30, 64]. Future work should investigate mitigations which take these new computing concepts into account rather than enforcing isolation boundaries that are less necessary in these use cases.

On current CPUs, Spectre-BTB mitigations, including retpoline, must remain enabled. On newer kernels for ARM Cortex-A CPUs, the branch prediction results can be discarded, and on certain devices branch prediction can be entirely disabled [3]. Our results suggest that these mechanisms are required for context switches or interrupt handling. Additionally, the L1TF mitigations must be applied on affected CPUs to prevent Foreshadow. Otherwise, we can still fetch arbitrary hypervisor addresses into the cache. Finally, our attacks also show that SGX enclaves must be compiled with the retpoline flag. Even with LVI mitigations, this is currently not the default setting, and thus all SGX enclaves which speculatively load secrets are potentially susceptible to *Dereference Trap*.

10 CONCLUSION

We confirmed the empirical results from several previous works [22, 57, 94, 103] while showing that the underlying root cause was misattributed in these works, resulting in incomplete mitigations [11, 21, 57, 68, 94, 96]. Our experiments clearly show that speculative dereferencing of a user-space register in the kernel causes the leakage. As a result, we were able to improve the performance of the original attack and show that CPUs from other hardware vendors like AMD, ARM, and IBM are also affected. We demonstrated that this effect can also be exploited via JavaScript in browsers, enabling us to leak the physical addresses of JavaScript variables. To systematically analyze the effect, we investigated its leakage capacity by implementing a cross-core covert channel which works without shared memory. We presented a novel technique, *Dereference Trap*, to leak the values of registers used in SGX (or privileged contexts) via speculative dereferencing. We demonstrated that it is possible to fetch addresses from hypervisors into the cache from the guest operating system by triggering interrupts, enabling Foreshadow (L1TF) on data from the L3 cache. Our results show that, for now, retpoline must remain enabled even on recent CPU generations to fully mitigate high impact microarchitectural attacks such as Foreshadow.

ACKNOWLEDGMENTS

We want to thank Moritz Lipp, Clémentine Maurice, Anders Fogh, Xiao Yuan, Jo Van Bulck, and Frank Piessens of the original papers for reviewing and providing feedback to drafts of this work and for discussing the technical root cause with us. Furthermore, we want to thank Intel and ARM for valuable feedback on an early draft. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Additional funding was provided by generous gifts from Cloudflare, from Intel, and from ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] 2020. Refined Speculative Execution Terminology. https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology
[2] Amazon AWS. 2019. AWS Lambda@Edge. https://aws.amazon.com/lambda/edge/
[3] ARM Limited. 2018. ARM: Whitepaper Cache Speculation Side-channels. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper
[4] ARM Limited. 2019. ARM Developer - Cortex-A53. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53
[5] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
[6] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. 2017. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. *Cryptology ePrint Archive, Report 2017/968* (2017).
[7] Sarani Bhattacharya and Debdeep Mukhopadhyay. 2016. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*.
[8] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. 2019. Efficient mitigation of side-channel based attacks against speculative execution processing architectures. US Patent App. 16/023,564.
[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*.
[10] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*.
[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/.
[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves

via Speculative Execution. In *EuroS&P*.
[13] Chromium. 2020. Mojo in Chromium. https://chromium.googlesource.com/chromium/src.git/+/master/mojo/README.md
[14] Cloudflare. 2019. Cloudflare Workers. https://www.cloudflare.com/products/cloudflare-workers/
[15] KVM contributors. 2019. Kernel-based Virtual Machine. https://www.linux-kvm.org
[16] Elixir bootlin. 2018. https://elixir.bootlin.com/linux/latest/source/arch/x86/kvm/svm.c#L5700
[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*.
[18] Agner Fog. 2016. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.
[19] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. 2017. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *RAID*.
[20] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.
[21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.
[22] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.
[23] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*.
[24] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[25] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*.
[26] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. A Faster and More Realistic Flush+Reload Attack on AES. In *COSADE*.
[27] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass.
[28] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.
[29] IAIK. 2016. Prefetch Side-Channel Attacks V2P. https://github.com/IAIK/prefetch/blob/master/v2p/v2p.c
[30] IBM. 2019. https://cloud.ibm.com/functions/
[31] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. Revision 4.0.
[32] Intel. 2018. Retpoline: A Branch Target Injection Mitigation. Revision 003.
[33] Intel. 2018. Speculative Execution Side Channel Mitigations. Revision 3.0.
[34] Intel. 2019. Intel 64 and IA-32 Architectures Optimization Reference Manual.
[35] Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide.
[36] Intel. US 9,280.474 B2, Mar, 2016. Adaptive data prefetching.
[37] Intel Corporation. 2014. Software Guard Extensions Programming Reference, Rev. 2. (2014).
[38] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P*.
[39] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Know Thy Neighbor: Crypto Library Detection in Cloud. *PETS* (2015).
[40] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 Strikes Back. In *AsiaCCS*.
[41] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*.
[42] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *USENIX ATC*.
[43] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*.
[44] kernel.org. 2009. Virtual memory map with 4 level page tables (x86_64). https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
[45] kernel.org. 2019. Documentation for /proc/sys/vm/* kernel version 2.6.29. https://www.kernel.org/doc/Documentation/sysctl/vm.txt
[46] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*.
[47] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).
[48] Kirill A. Shutemov. 2015. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce
[49] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.
[50] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.
[51] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*.
[52] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium*.
[53] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*.
[54] Jonathan Levin. 2012. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons.
[55] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS*.
[56] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.
[57] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.
[58] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.
[59] LKML. 2018. x86/pti updates for 4.16. http://lkml.iu.edu/hypermail/linux/kernel/1801.3/03399.html
[60] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.
[61] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *DIMVA*.
[62] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.
[63] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178* (2019).
[64] Microsoft. 2019. Azure serverless computing. https://azure.microsoft.com/en-us/overview/serverless-computing/
[65] Microsoft Techcommunity. 2018. Hyper-V HyperClear Mitigation for L1 Terminal Fault. https://techcommunity.microsoft.com/t5/Virtualization/Hyper-V-HyperClear-Mitigation-for-L1-Terminal-Fault/ba-p/382429
[66] Mozilla. 2019. JavaScript data structures. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures
[67] Mozilla. 2019. JS API Reference. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/JSAPI_reference
[68] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. *A Survey of Published Attacks on Intel SGX*.
[69] 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. 2018. Spectre attack against SGX enclave.
[70] OpenSSL. 2019. OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org
[71] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*.
[72] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.
[73] Dan Page. 2002. Theoretical use of cache memory as a cryptanalytic side-channel. *Cryptology ePrint Archive, Report 2002/169* (2002).
[74] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.
[75] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*.
[76] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*.
[77] Chester Rebeiro, Debdeep Mukhopadhyay, Junko Takahashi, and Toshinori Fukunaga. 2009. Cache timing attacks on Clefia. In *International Conference on Cryptology in India*.
[78] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*.
[79] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*

(2019).
[80] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2018. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. *AsiaCCS* (2018).
[81] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks . In *DIMVA*.
[82] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.
[83] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*.
[84] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*.
[85] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical Enclave Malware with Intel SGX. In *DIMVA*.
[86] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings*.
[87] Slashdot EditorDavid. 2019. Two Linux Kernels Revert Performance-Killing Spectre Patches. https://linux.slashdot.org/story/18/11/24/2320228/two-linux-kernels-revert-performance-killing-spectre-patches
[88] Julian Stecklina. 2019. An demonstrator for the L1TF/Foreshadow vulnerability. https://github.com/blitz/l1tf-demo
[89] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. 2003. Cryptanalysis of DES implemented on computers with cache. In *CHES*.
[90] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886
[91] Ubuntu Security Team. 2019. L1 Terminal Fault (L1TF). https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/L1TF
[92] V8 team. 2018. v8 - Adding BigInts to V8. https://v8.dev/blog/bigint
[93] V8 team. 2019. v8 - Documentation. https://v8.dev/docs
[94] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.
[95] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*.
[96] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.
[97] Vish Viswanathan. [n.d.]. Disclosure of Hardware Prefetcher Control on Some Intel Processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors
[98] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf
[99] Zhenyu Wu, Zhang Xu, and Haining Wang. 2014. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. *IEEE/ACM Transactions on Networking* (2014).
[100] xenbits. 2019. Cache-load gadgets exploitable with L1TF. https://xenbits.xen.org/xsa/advisory-289.html
[101] xenbits.xen.org. 2009. page.h source code. http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;hb=refs/heads/stable-4.3;f=xen/include/asm-x86/x86_64/page.h
[102] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium*.
[103] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. https://doi.org/10.14722/ndss.2020.23105
[104] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.
[105] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P*.
[106] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*.

A MISTRAINING BTB FOR `SCHED_YIELD`

We evaluate the mistraining of the BTB by calling different syscalls, fill all general-purpose registers with DPM address and call `sched_yield`.

Table 3: Table of syscalls which achieve the highest numbers of cache fetches, when calling sched_yield after the register filling.

Syscall	Parameters	Avg. # cache fetches
readv	readv(0,NULL,0);	13766.3
getcwd	syscall(79,NULL,0);	7344.7
getcwd	getcwd(NULL,0);	6646.9
readv	syscall(19,0,NULL,0);	5541.4
mount	syscall(165,s_cbuf,s_cbuf,s_cbuf,s_ulong,(void*)s_cbuf);	4831.6
getpeername	syscall(52,0,NULL,NULL);	4600.0
getcwd	syscall(79,s_cbuf,s_ulong);	4365.8
bind	syscall(49,0,NULL,0);	3680.6
getcwd	getcwd(s_cbuf,s_ulong);	3619.3
getpeername	syscall(52,s_fd,&s_ssockaddr,&s_int);	3589.3
connect	syscall(42,s_fd,&s_ssockaddr,s_int);	2951.2
getpeername	getpeername(0,NULL,NULL);	2822.4
connect	syscall(42,0,NULL,0);	2776.4
getsockname	syscall(51,0,NULL,NULL);	2623.4
connect	connect(0,NULL,0);	2541.5

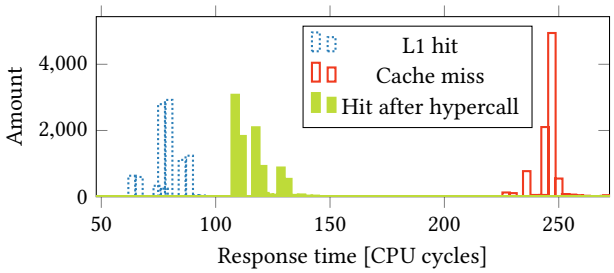

Figure 6: Timings of a cached and uncached variable and the access time after a hypercall in a Ubuntu VM on Hyper-V.

Our test system was equipped with Ubuntu 18.04 (kernel 4.4.143-generic) and an Intel i7-6700K. We repeated the experiment by iterating over various syscalls with different parameters (valid parameters,NULL as parameters) 10 times with 200 000 repetitions. Table 3 lists the best 15 syscalls to mistrain the BTB when sched_yield is performed afterwards. On this kernel version it appears that the read and getcwd syscalls mistraing the BTB best if sched_yield is called after the register filling.

B WEBASSEMBLY REGISTER FILLING

The WebAssembly method `load_pointer` of Listing 6 takes two 32-bit JavaScript values as input parameters. These two parameters are loaded into a 64-bit integer variable and stored into multiple global variables. The global variables are then used as loop exit conditions in the separate loops. To fill as many registers as possible with the direct-physical-map address, we create data dependencies within the loop conditions. In the `spec_fetch` function, the registers are filled inside the loop. After the loop, the JavaScript function `yield_wrapper` is called. This tries to trigger any syscall or interrupt in the browser by calling JavaScript functions which may incur syscalls or interrupts. Lipp et al. [55] reported that web requests from JavaScript trigger interrupts from within the browser.

C NO FORESHADOW ON HYPER-V HYPERCLEAR

We set up a Hyper-V virtual machine with a Ubuntu 18.04 guest (kernel `5.0.0-20`). We access an address to load it into the cache and perform a hypercall before accessing the variable and measuring

```
extern void yield_wrapper();
uint64_t G1 = 5;
uint64_t G2 = 5;
uint64_t G3 = 5;
uint64_t G4 = 5;
uint64_t G5 = 5;
uint64_t value = 0;

void spec_fetch()
{
  for (uint64_t i = G1+5; i > G1; i--)
    for (uint64_t k = G3+5; k > G3; k--)
      for (uint64_t j = G2-5; k < G2; j++)
        for(uint64_t l = G4; i < G4;l++)
          for(uint64_t m = G5-5;m<G5;m++)
            value = l + j + k + i;
    yield_wrapper();
}

int load_pointer(int high, int low)
{
  uint64_t a = (((uint64_t)high) << 32ull) |
   ((uint64_t)(unsigned int)low);
  G1 = a;
  G2 = a;
  G3 = a;
  G4 = a;
  G5 = a;
  spec_fetch();
  return a;
}

int main()
{
  load_pointer(0x12345678,0x9abcdef0);
}
```

Listing 6: WebAssembly code to speculatively fetch an address from the kernel direct-physical map into the cache. We combine this with a state-of-the-art Evict+Reload loop in JavaScript to determine whether the guess for the direct-physical map address was correct.

the access time. Since hypercalls are performed from a privileged mode, we developed a kernel module for our Linux guest machine which performs our own malicious hypercalls. We observe a timing difference (see Figure 6) between a memory access which hits in the L1 cache (dotted), a memory access after a hypercall (grid pattern), and an uncached memory access (crosshatch dots). We observe that after each hypercall, the access times are approx. 20 cycles slower. This indicates that the guest addresses are flushed from the L1 data cache. In addition, we create a second experiment where we load a virtual address from a process running on the host into several registers when performing a hypercall from the guest. On the host system, we perform Flush+Reload on the virtual address in a loop and verify whether the virtual address is fetched into the cache. We do not observe any cache hits on the host process when performing hypercalls from the guest system. Thus we conclude that either the L1 cache is always flushed, contradicting the documentation, or creating a situation where the L1 cache is not flushed requires a more elaborate attack setup. However, we believe that speculative dereferencing is the reason why Microsoft adopted the retpoline mitigation despite having other Spectre-BTB mitigations already in place.