Speculative Dereferencing: Reviving Foreshadow (Extended Version)

Martin Schwarzl[1], Thomas Schuster[1], Michael Schwarz[2], and Daniel Gruss[1]

[1]Graz University of Technology, Austria [2]CISPA Helmholtz Center for Information Security, Germany

Abstract. In this paper, we provide a systematic analysis of the root cause of the prefetching effect observed in previous works and show that its attribution to a prefetching mechanism is incorrect in all previous works, leading to incorrect conclusions and incomplete defenses. We show that the root cause is speculative dereferencing of user-space registers in the kernel. This new insight enables the first end-to-end Foreshadow (L1TF) exploit targeting non-L1 data, despite Foreshadow mitigations enabled, a novel technique to directly leak register values, and several side-channel attacks. While the L1TF effect is mitigated on the most recent Intel CPUs, all other attacks we present still work on all Intel CPUs and on CPUs by other vendors previously believed to be unaffected.

1 Introduction

For security reasons, operating systems hide physical addresses from user programs [34]. Hence, an attacker requiring this information has to leak it first, e.g., with the *address-translation attack* by Gruss et al. [17, §3.3 and §5]. It allows user programs to fetch arbitrary kernel addresses into the cache and thereby to resolve virtual to physical addresses. As a mitigation against e.g., the address-translation attack, Gruss et al. [17, 16] proposed the KAISER technique.

Other attacks observed and exploited similar prefetching effects. Meltdown [41] practically leaks memory that is not in the L1 cache. Xiao et al. [72] show that this relies on a prefetching effect that fetches data from the L3 cache into the L1 cache. However, Van Bulck et al. [66] observe no such effect for Foreshadow.

We systematically analyze the root cause of the prefetching effect exploited in these works. We show that, despite the sound approach of these papers, the attribution of the root cause, *i.e.*, why the kernel addresses are cached, is incorrect in all cases. The root cause is 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 many speculative code paths in the kernel, we focus on code paths with Spectre [35, 6] gadgets that can be reliably triggered on both Linux and Windows.

These new insights correct several wrong assumptions from previous works, also leading to new attacks. Most significantly, the difference that Meltdown can leak from L3 or main memory [41] but Foreshadow (L1TF) can only leak from

L1 [66, Appendix A], was never a limitation in practice. The same effect that allowed Meltdown to leak data from L3, enables our slightly modified Foreshadow attack to leak data from L3 as well, *i.e.*, L1TF was in practice never restricted to the L1 cache. Worse still, we show that for the same reason Foreshadow mitigations [66, 69] are still incomplete. We reveal that Foreshadow attacks are unmitigated on many kernel versions even with all mitigations and even on the most recent kernel versions. However, retpoline affects the success rate, but it is only enabled on some kernel versions and some microarchitectures.

We present a new technique that uses dereferencing gadgets to directly leak data without an encoding attack step. We show that we can leak data from registers, e.g., cryptographic key material, from SGX and that the assumptions in previous works were incorrect, making certain attacks only reproducible on kernels susceptible to speculative dereferencing, including, e.g., results from Gruss et al. [17, §3.3 and §5], Lipp et al. [41, §6.2], and Xiao et al. [72, §4-E]. This also allowed us to improve the performance of address-translation attacks and to mount them in JavaScript [17]. We demonstrate that the address-translation attack also works on recent Intel CPUs with the latest hardware mitigations with all mitigations enabled. Finally, we also demonstrate the attack on CPUs previously believed to be unaffected by the prefetch address-translation attack, *i.e.*, ARM, IBM Power9, and AMD CPUs.

Contributions. The main contributions of this work are:

1. We discover an incorrect attribution of the root cause in previous works to prefetching effects [72, 17, 41].
2. We show that the root cause is speculative execution, leaving CPUs from other vendors equally affected and the effect exploitable from JavaScript.
3. We discover a novel way to exploit speculative dereferences, enabling direct leakage data in registers.
4. We show that this effect, responsible for Meltdown from non-L1 data, can be adapted to Foreshadow and show that Foreshadow attacks on data from the L3 cache are possible, even with Foreshadow mitigations enabled.

Outline. Section 2 provides background. Section 3 analyzes the root cause. Section 4 improves and extends the attacks. Section 5 presents cross-VM data leakage. Section 6 presents a new leakage method. Section 7 presents a JavaScript-based attack. Section 8 discusses implications. Section 9 concludes.

2 Background and Related Work

In this section, we provide relevant details regarding virtual memory, CPU caches, Intel SGX, and transient execution attacks and defenses.

Virtual Memory. In modern systems, each process has its own virtual address space, divided into user and kernel space. Many operating systems map physical memory directly into the kernel [30, 39], e.g., to access paging structures. Thus, every user page is mapped at least twice: in user space and in the kernel direct-physical map. Access to virtual-to-physical address information requires

root privileges [34]. The prefetch address-translation attack [17, §3.3 and §5] obtains the physical address for any user-space address via a side-channel attack.

Caches and Prefetching. Modern CPUs have multiple cache levels, hiding latency of slower memory levels. Software prefetch instructions hint the CPU that a memory address should already be fetched into the cache early to improve performance. Intel and AMD x86 CPUs have 5 software `prefetch*` instructions.

Prefetching attacks. Gruss et al. [17] 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 [73] on the user-space address. On a cache hit, the guess was correct. Hence, the attacker can determine the exact physical address for any virtual address, re-enabling various mircorarchitectural attacks [43, 50, 60, 32].

Intel SGX. Intel SGX is a trusted execution mechanism enabling the execution of trusted code in a separate protected area called an enclave [26]. Although enclave memory is mapped in the virtual address space of the host application, the hardware prevents access to the code or data of the enclave from any source other than the enclave code itself [27]. However, as has been shown in the past, it is possible to exploit SGX via memory corruption [37, 54], ransomware [59], side-channel attacks [5, 55], and transient-execution attacks [66, 56, 52, 67].

Transient Execution. Modern CPUs execute instructions *out of order* to improve performance and then retire *in order* from reorder buffers. Another performance optimization, speculative execution, predicts control flow and data flow for not-yet resolved conditional control- or data-flow changes. Intel CPUs have several branch predictors [25], e.g., the Branch History Buffer (BHB) [3, 35], Branch Target Buffer (BTB) [38, 12, 35], Pattern History Table (PHT) [13, 35], and Return Stack Buffer (RSB) [13, 42, 36]. Instructions executed out-of-order or speculatively but not architecturally are called *transient instructions* [41].

These *transient instructions* can have measurable side effects, e.g., modification of TLB and cache state, that can be exploited to extract secrets in so-called transient-execution attacks [6, 28]. Spectre-type attacks [35, 33, 7, 19, 36, 42, 58] exploit misspeculation in a victim context. By executing along the misspeculated path, the victim inadvertently leaks information to the attacker. To mitigate Spectre-type attacks several mitigations were developed [24], such as retpoline [23], which replaces indirect jump instructions with `ret` instructions.

In Meltdown-type attacks [41], such as Foreshadow [66], an attacker deliberately accesses memory across isolation boundaries, which is possible due to deferred permission checks in out-of-order execution. Foreshadow exploits a cleared present bit in the page table-entry to leak data from the L1 cache or the line fill buffer [52, 56]. A widely accepted mitigation is to flush the L1 caches and line fill buffers upon context switches and to disable hyperthreading [22].

```
1 41 0f 18 06   prefetchnta (%r14) ; replace with nop for testing, r14 = direct phys. addr.
2 41 0f 18 1e   prefetcht2 (%r14)  ; replace with nop for testing, r14 = direct phys. addr.
```

Listing 1: Disassembly of the prefetching in the address-translation attack.

3 From Address-Translation Attack to Foreshadow-L3

In this section, we systematically analyze the properties of the address-translation attack erroneously attributed to the software prefetch instructions [17, §3.3 and §5]. We identify the root cause to be unmitigated misspeculation in the kernel, leading to a new Foreshadow-L3 attack that works despite mitigations [66].

In the address-translation attack [17] the attacker tries to find a direct physical map address $\bar{p}$ for a virtual address p. The attacker flushes the user-space address p, and prefetches the inaccessible direct physical map address $\bar{p}$. If Flush+Reload [73] determines that p was reloaded via $\bar{p}$, the physical address of p is $\bar{p}$ minus the known direct-physical-map offset. We measure the attack performance in *fetches per second*, *i.e.*, how often per second p was cached via $\bar{p}$.

The prefetching component of the original attack's proof-of-concept [20] runs a loop, `for (size_t i = 0; i < 3; ++i) { sched_yield(); prefetch(direct_phys_map_addr); }`. The compiled and disassembled code can be found in Listing 1. We extracted the following hypotheses(H1-H5) from the original attack (cf. Appendix A for quotes):

H1 the `prefetch` instruction (to instruct the prefetcher to prefetch);
H2 the value stored in the register used by the `prefetch` instruction (to indicate which address the prefetcher should prefetch);
H3 the `sched_yield` syscall (to give time to the prefetcher);
H4 the use of the `userspace_accessible` bit (as kernel addresses could otherwise not be translated in a user context);
H5 an Intel CPU – other CPU vendors are claimed to be unaffected.

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. We replace the `prefetch` instructions in the original code [20] with same-size `nop`s (cf. Listing 1). Surprisingly, we observe no change in the number of cache fetches, *i.e.*, we measure 60 cache fetches per second (i7-8700K, Ubuntu 18.10, kernel `4.15.0-55`), without any `prefetch` instruction. We also exclude the hardware prefetcher by disabling them via the model-specific register `0x1a4` [68] during the experiment. We still observe ≈ 60 cache fetches per second.

Documented prefetchers are not required for the address-translation attack.

3.2 H2: Values in registers required

The second hypothesis is that providing the direct-physical map address via the register is necessary. The registers that must be used vary across kernel versions.

We identified the registers `r12,r13,r14` (Ubuntu 18.10, kernel `4.18.0-17`), `r9,r10` (Debian 8, kernel `4.19.28-2` and Kali Linux, kernel `5.3.9-1kali1`) and `rdi,rdx` (Linux Mint 19, kernel `4.15.0-52`). Gruss et al. [17] used recompiled binaries that used different registers for the kernel address (cf. Appendix A).

A referenced location is only fetched into the cache if the absolute virtual address is stored in one of these registers.

We additionally verified that only the absolute virtual address causes this effect. Any other addressing mode for the prefetch instruction does not leak. By loading the address into most general-purpose registers, we observe leakage across all Linux versions, even with KPTI enabled, meaning that the KAISER technique [16] never protected against this attack. Instead, the implementation merely changed the required registers, hiding the effect for a specific binary-kernel combination. On an Intel Xeon Silver 4208 CPU with in-silicon patches against Meltdown [41], Foreshadow [66], and ZombieLoad [56], we still observe about 30 cache fetches per second on Ubuntu 19.04 (kernel `5.0.0-25`). On Windows 10 (build `1803.17134`), which has no direct physical map, we fill all registers with a kernel address and perform the syscall `SwitchToThread`. We observe ≈ 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. We observe that other syscalls e.g., `gettid`, expose a similar number of cache fetches. This shows that `sched_yield` is not required and can be replaced with other syscalls. To test whether syscalls in the main attack loop are required, we run a address-translation attack without context switches or interrupts and without `sched_yield` on an isolated core. Here, we do not observe any cache fetches (i7-8700K, kernel 4.15.0-55) when running this attack for 10 hours. However, when inducing a large number of context switches using interrupts, we observe about 15 cache fetches per second if the process filling the registers gets interrupted continuously. These hits occur during speculative execution in the interrupt handler, as we validated manually via code changes and fencing in interrupt handlers.

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. We choose cache lines A and B on different pages. In a loop, we dereference a register pointing to A and use Flush+Reload to detect whether A was cached. In the last loop iteration, we speculatively exchange the register value to point to either B or the direct-physical map address of B. Hence, both the architectural and speculative

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

Listing 2: The kernel performs indirect jumps, e.g., to syscall handlers. With retpoline [63], the kernel uses a `retq` instead of the indirect jump.

dereferences happen at the same instruction pointer value and in the same register. With a register-value-based hardware prefetcher, we would expect B to be cached. When dereferencing the direct-physical-map address of B architecturally, B is usually cached after the loop. However, when we dereference the register with its value speculatively changed from A to either B or the direct-physical map address of B, B is never cached after the final run. In a second experiment, we show that the effect originates from the kernel. While prefetching direct-physical-map addresses works, user-space addresses are only fetched when SMAP (supervisor-mode access prevention) is disabled. Thus, the root cause of the address-translation attack adheres to SMAP.

Hence, we can conclude that the root cause is 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 run our experiments (cf. Section 3.4) on an AMD Ryzen Threadripper 1920X (Ubuntu 17.10, kernel `4.13.0-46generic`), an ARM Cortex-A57 (Ubuntu 16.04.6 LTS, kernel `4.4.38-tegra`), and an IBM Power9 (Ubuntu 18.04, kernel `4.15.0-29`). On the AMD Ryzen Threadripper 1920X, we achieve up to 20, on the Cortex-A57 up to 5, and on the IBM Power9 up to 15 speculative fetches per second.

Any Spectre-susceptible CPU is also susceptible to speculative dereferencing.

3.6 Speculative Execution in the Kernel

From the previous analysis, we conclude that the leakage is due to speculative execution in the kernel. While this might not be suprising with the knowledge of Spectre, Spectre was only discovered one year after the original prefetch paper [17] was published. We show that the primary leakage is caused by Spectre-BTB-SA-IP (training in same address space, and in-place) [6].

During a syscall, the kernel performs multiple indirect jumps (cf. Listing 2), which are generally susceptible to Spectre-BTB-SA-IP. The address-translation attack succeeds because misspeculated branch targets dereference registers without sanitization. With retpoline, the kernel uses a `retq` instead of the indirect jump to trap the speculative execution to a fixed branch. Thus, during speculative execution, the CPU might use an incorrect prediction from the branch-target buffer (BTB) and speculate into the wrong syscall while registers contain attacker-chosen addresses (cf. Figure 1). In the misspeculated syscall, registers containing attacker-chosen addresses are used. On recent kernels

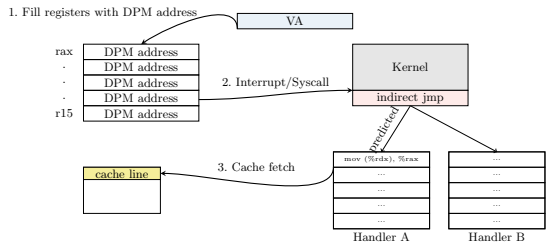

Fig. 1: The kernel speculatively dereferences the direct-physical map address. Flush+Reload detects cache hits on the corresponding user-space address.

```
1 movzbl (%rax,%rdi,1),%eax
2 <op> (%rcx,%rax,1),%dl
3 ; gadget in Linux kernel
4 98d4be:       0f b6 34 06             movzbl (%rsi,%rax,1),%esi
5 98d4c2:       45 01 3c b3             add    %r15d,(%r11,%rsi,4)
```

Listing 3: If the attacker controls three register values, it is possible to leak arbitrary kernel memory.

(4.19 or newer), retpoline eliminates the leakage. We provide a full analysis of the `sched_yield` gadget causing speculative dereferences in Appendix B. Even worse, cloud providers still use older kernel versions (e.g., the first option on AWS at the time of writing is Amazon Linux 2 AMI with kernel 4.14) where retpoline does *not* fully eliminate the leakage. On the other hand, recent systems such as Ice Lake do not use retpoline anymore due to improved hardware mitigations, which unfortunately have *no* effect on our speculative dereferencing attack. Hence, our attack remains unmitigated on many systems, and is most importantly not mitigated by KAISER (KPTI) [16], or LAZARUS [14] as claimed in previous works. The Spectre-BTB-SA-IP leak from Listing 2 is only one of many, e.g., we still observe $\approx$ 15 speculative fetches per second on an i5-8250U (kernel `5.0.0-20`) if we eliminate this specific leak. However, any prefetch gadget [6], based on PHT, BTB, or RSB mispredictions, can be used for an address-translation attack [17] and thus would also re-enable Foreshadow-VMM attacks [66, 69]. Concurrent work showed that there are kernel gadgets to fetch data into the L1D cache in Xen [71] and an artificial gadget was exploited by Stecklina for that purpose [62].

We also 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. In between, stack dereferences in misspeculated branches can still access these values. On recent Ice Lake processors, retpoline is replaced by enhanced IBRS. Unfortunately, this is a security regression, re-enabling Spectre-BTB in-place attacks and, thus, moves our focus on a set of previously overlooked gadgets, where the user only controls certain register values in the transient domain. We measure the performance of our attack by exploiting such a Spectre-BTB gadget in a kernel module and evaluate it

on our Ice Lake CPU. Listing 3 illustrates an eIBRS-bypassing Spectre-BTB gadget containing only two instructions, where the attacker controls, e.g., three registers. The smallest eIBRS-bypassing Spectre-BTB gadget we found contains only 7 bytes.

We demonstrate that on Ice Lake, this regression re-enables transient leakage of kernel memory like the original Spectre attack paper described [35], *i.e.*, measured by leaking a 1024 B secret key. We observe a completely noise-free leakage rate of 30 B/s ($n = 1000, \sigma_{\bar{x}} = 0.1429$). By shifting the byte *i.e.*, binary searching via two consecutive cache lines, we then can recover the exact byte value [35]. We analyzed the Linux kernel 5.4.0-48 (vmlinux binary) and looked for similar opcodes and found a gadget at offset `0x984dbe`(see Listing 3 line `3` and `4`).

3.7 Meltdown-L3 and Foreshadow-L3

The speculative dereferencing was noticed but also misattributed to the prefetcher in subsequent work. The Meltdown paper [41] reports that data is fetched from L3 into L1 while mounting a Meltdown attack. Van Bulck et al. [66] confirmed the effect for Meltdown but did not observe this prefetching effect for Foreshadow. Based on this observation, further works also mentioned this effect without analyzing it thoroughly [6, 47, 52, 31]. Xiao et al. [72] state that a Meltdown-US attack causes data to be repeatedly prefetched from L1 to L3 [72].

We used similar Meltdown-L3 setups as SpeechMiner [72] (kernel 4.4.0-134 with boot flags `nopti,nokaslr` and Meltdown [41] (Ubuntu 16.10, kernel 4.8.0, no mitigations existed back then). In our Meltdown-L3 experiment, 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. With the `nospectre_v2` flag, our Meltdown-L3 attack works again by triggering the prefetch gadget in the kernel on the direct-physical map address. In the SpeechMiner [72] and Meltdown [41] experiment, no mitigation (including retpoline) eliminates the leakage fully. 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 [66]. 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. We demonstrate this attack in KVM in Section 5.

4 Improving the Leakage Rate

We can leverage our insights to increase the leakage by using syscalls other than `sched_yield`, and executing additional syscalls to mistrain the branch predictor.

Setup. We tested our attacks on an Intel i5-8250U (Linux kernel 4.15.0-52), an i7-8700K (Linux kernel 4.15.0-55), an ARM Cortex-A57 (Linux kernel 4.4.38-tegra), and an AMD Threadripper 1920X (Linux kernel 4.13.0-46). As retpoline is not available on all machines, we run the tests without retpoline. By performing syscalls before filling the registers with the direct-physical map address, we can mistrain the BTB, triggering the CPU to speculatively execute this syscall. We evaluate this mistraining for `sched_yield` in Appendix C.

Evaluation. We evaluated different syscalls for branch prediction mistraining by executing a single syscall before and after filling the registers with the target address. We observe that effects occur for different syscalls and both on AMD and ARM CPUs, with similar success rates (cf. Appendix G). Alternating syscalls additionally mistrains the branch prediction and increases the success rate, e.g., with syscalls like `stat`, `sendto`, or `geteuid`. However, not every additional syscall increases the number of cache fetches. On recent Linux kernels (version `5`), we observe that the number of speculative cache fetches decreases, due to a change in syscall handling. Our results show that the `pipe` syscall much more reliably triggers speculative dereferencing ($\geq$99.9 %), but the execution time of `sched_yield` is much lower and thus despite the lower success rate (around 66.4 % in the most basic case) it yields a higher attack performance.

Capacity Measurement in a Cross-Core Covert Channel. We measure the capacity of our attack in a covert channel by using the speculative dereferencing effect ('1'-bit) or not ('0'-bit). The receiver uses Flush+Reload to measure whether the cache state of cache line dereferenced in the kernel. We evaluated the covert channel on random data and across physical CPU cores. Our test system was equipped with an Intel i7-6500U CPU Linux 4.15.0-52 with the `nospectre_v2` boot flag. 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 [50, 70]. To achieve an error-free transmission, error-correction techniques [43] can be used. I/O interrupts, *i.e.*, syncing the NVMe device, create additional speculative dereferences and can thus further improve the capacity.

5 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 [10]. We leak data (e.g., cryptographic keys) from other virtual machines and the hypervisor, like the original Foreshadow attack. We do not observe any speculative dereferencing of guest-controlled registers in Microsoft's Hyper-V HyperClear Foreshadow mitigation which additionally uses retpoline, or on more recent kernel versions with

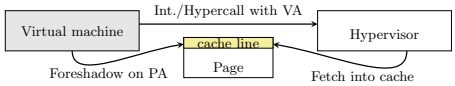

Fig. 2: If a guest-chosen address is speculatively fetched into the cache during a hypercall or interrupt and not flushed before the virtual machine is resumed, the attacker can perform a Foreshadow attack to leak the fetched data.

retpoline. We provide a thorough analysis of this negative result. However, the attack succeeds even with the recommended Foreshadow mitigations enabled and with kernel versions before 4.18 (e.g., as used by default on AWS Amazon Linux 2 AMI) with all default mitigations enabled, *i.e.*, including retpoline. We investigate whether speculative dereferencing also exists in hypercalls. 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, it has 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 [64, 71, 62]. Figure 2 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.

5.1 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 [71, 62]. 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 [64], 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 [11]. 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 [66], allowing guest-to-host memory reads. 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. In this attack the host constantly performs encryptions using a secret key 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 [1]. 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 retpoline 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 [62, 71], prefetch gadgets exist in the kernel that can be exploited to fetch data into the cache, and these gadgets can be exploited using Foreshadow.

5.2 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 retpoline and tried to fetch hypervisor addresses from guest systems into the cache. Microsoft's Hyper-V HyperClear Mitigation [45] 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

[1] Demonstration video can be found here: https://streamable.com/8ke5ub

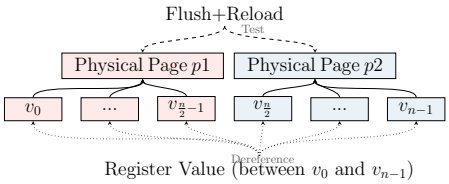

Fig. 3: Leaking the value of an x86 general-purpose register using *Dereference Trap* and Flush+Reload on two different physical addresses. v_0 to v_{n-1} represent the memory mappings on one of the shared memory regions.

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 **??**.

6 Leaking Values from SGX Registers

In this section, we present a novel method, *Dereference Trap*, to leak register contents via speculative register dereference. Leaking the values of registers is useful, e.g., to extract parts of keys from cryptographic operations.

6.1 Dereference Trap

The setup for *Dereference Trap* is similar as in Section 3.6. We exploit transient code paths inside an SGX enclave that speculatively dereference a register containing a secret value. Such a gadget is easily introduced in an enclave, e.g., when using polymorphism in C++. Listing 5 (Appendix D) shows a minimal example of introducing such a gadget. However, there are also many different causes for such gadgets [23], e.g., function pointers or (compiler-generated) jump tables. 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, it is infeasible to back every virtual address with unique physical pages and mount Flush+Reload on every cache line, as that takes 2 days on a 4 GHz CPU [54].

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 3. By leveraging shared memory, we can map one physical page multiple times into the virtual address space. The maximum number of mappings per page 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. On our test system, we recovered a 32-bit value (without the least-significant 6 bits) stored in a 64-bit register within 15 minutes with *Dereference Trap*.

6.2 Generalization of Dereference Trap

Dereference Trap is a generic technique that applies to any scenario where the attacker can set up the hardware and address space accordingly. *Dereference Trap* applies to all Spectre variants. 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, and 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). Using this technique the attacker can reliably leak a 32-bit secret which is speculatively dereferenced by the kernel. Cryptographic implementations often store keys in the lower 32 bits of 64bit registers (*i.e.*, OpenSSL AES round key u32 *rk) [48]. Hence, these 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 (without the least-significant 6 bits). Thus, retpoline and SMAP must remain enabled to mitigate attacks like *Dereference Trap*.

7 Leaking Physical Addresses from JavaScript using WebAssembly

In this section, we present an attack that leaks the physical address (cache-line granularity) of a JavaScript variable. This shows that the "prefetching" effect is much simpler than described in the original paper [17], *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 native code in browser, we create a JavaScript-based attack to leak physical addresses from Javascript variables

and evaluate its performance in Firefox. We demonstrate that it is possible to fill 64-bit registers with an attacker-controlled value via WebAssembly.

Attack Setup. JavaScript encodes numbers as 53-bit double-precision floating-point values [46]. It is not possible to store a full 64-bit value into a register with vanilla JavaScript. Hence, we leverage WebAssembly, a binary instruction format which is precompiled for the JavaScript engine and not further optimized [65]. On our test system (i7-8550U, Debian 8, kernel`5.3.9-1kali1`) registers `r9` and `r10` are speculatively dereferenced in the kernel. Hence, we fill these registers with a guessed direct-physical-map address of a variable. The WebAssembly method `load_pointer` (Listing 6 Appendix E) takes two 32-bit values that are combined into a 64-bit value and populated into as many registers as possible. To trigger interrupts, we use web requests, as shown by Lipp et al. [40]. Our attack leaks the direct-physical-map address of a JavaScript variable. The attack works analogously to the native-code address-translation attack [17].

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 [40].
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. 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 [49, 50, 57, 18, 15, 35, 53]. Hence, the address-translation attack is possible with JavaScript and WebAssembly, without requiring the NaCl sandbox as in the original paper [17]. Upcoming JavaScript extensions expose syscalls to JavaScript [8]. 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.

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. [29] stated, not all registers are used in JIT-generated native code [29].

8 Discussion

The "prefetching" effect was first observed by Gruss et al. [17] in 2016. In May 2017, Jann Horn discovered that speculative execution can be exploited to leak arbitrary data, later on published in the Spectre [35] paper. 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 [51] and documented [26] to cause cache hits on addresses that are not architecturally accessed. Currently, the address-translation attack and our variants are mitigated on both Linux and Windows using the retpoline technique to avoid indirect branches. 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 % [61]. 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 [4]. As an alternative, randpoline [4] 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 [1, 9, 44, 21]. On current CPUs, retpoline must remain enabled, which is not the default in many cases. Other Spectre-BTB mitigations, including enhanced IBRS, do not mitigate our attack. 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 [2]. 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 are potentially susceptible to *Dereference Trap*.

9 Conclusion

We showed that the underlying root cause of prefetching effects was misattributed in previous works [16, 41, 66, 6, 47, 52, 31] and speculative dereferencing of a user-space register in the kernel actually causes the leakage. As a result, we were able to mount a Foreshadow (L1TF) attack on data from the L3 cache,

even with the latest mitigations enabled. Furthermore, we were able to improve the performance of the original attack, apply it to AMD, ARM, and IBM and exploit the effect via JavaScript in browsers. Our novel technique, *Dereference Trap*, leaks the values of registers used in SGX (or privileged contexts) via speculative dereferencing.

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). Additional funding was provided by generous gifts from Cloudflare, Intel and Red Hat. 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. Amazon AWS: AWS Lambda@Edge (2019), `https://aws.amazon.com/lambda/edge/`
2. ARM: ARM: Whitepaper Cache Speculation Side-channels (2018), `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper`
3. Bhattacharya, S., Maurice, C.m.t.n., Bhasin, S., Mukhopadhyay, D.: Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. Cryptology ePrint Archive, Report 2017/968 (2017)
4. Branco, R., Hu, K., Sun, K., Kawakami, H.: Efficient mitigation of side-channel based attacks against speculative execution processing architectures (2019), uS Patent App. 16/023,564
5. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT (2017)
6. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium (2019), extended classification tree and PoCs at https://transient.fail/.
7. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P (2019)
8. Chromium: Mojo in Chromium (2020), `https://chromium.googlesource.com/chromium/src.git/+/master/mojo/README.md`
9. Cloudflare: Cloudflare Workers (2019), `https://www.cloudflare.com/products/cloudflare-workers/`
10. contributors, K.: Kernel-based Virtual Machine (2019), `https://www.linux-kvm.org`

11. Elixir bootlin: (2018), {https://elixir.bootlin.com/linux/latest/source/arch/x86/kvm/svm.c#L5700}
12. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over aslr: Attacking branch predictors to bypass aslr. In: MICRO (2016)
13. Fog, A.: The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers (2016)
14. Gens, D., Arias, O., Sullivan, D., Liebchen, C., Jin, Y., Sadeghi, A.R.: LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In: RAID (2017)
15. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)
16. Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: KASLR is Dead: Long Live KASLR. In: ESSoS (2017)
17. Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S.: Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS (2016)
18. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA (2016)
19. Horn, J.: speculative execution, variant 4: speculative store bypass (2018)
20. IAIK: Prefetch Side-Channel Attacks V2P (2016), {https://github.com/IAIK/prefetch/blob/master/v2p/v2p.c}
21. IBM: (2019), https://cloud.ibm.com/functions/
22. Intel: Intel Analysis of Speculative Execution Side Channels (2018), revision 4.0
23. Intel: Retpoline: A Branch Target Injection Mitigation (2018), revision 003
24. Intel: Speculative Execution Side Channel Mitigations (2018), revision 3.0
25. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (2019)
26. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide (2019)
27. Intel Corporation: Software Guard Extensions Programming Reference, Rev. 2. (2014)
28. Intel Corporation: Refined Speculative Execution Terminology (2020), https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology
29. Jangda, A., Powers, B., Berger, E.D., Guha, A.: Not so fast: Analyzing the performance of webassembly vs. native code. In: USENIX ATC (2019)
30. kernel.org: Virtual memory map with 4 level page tables (x86_64) (2009), https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
31. Kim, T., Shin, Y.: Reinforcing meltdown attack by using a return stack buffer. IEEE Access 7 (2019)
32. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: ISCA (2014)
33. Kiriansky, V., Waldspurger, C.: Speculative Buffer Overflows: Attacks and Defenses. arXiv:1807.03757 (2018)
34. Kirill A. Shutemov: Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace (2015), https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce
35. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: S&P (2019)
36. Koruyeh, E.M., Khasawneh, K., Song, C., Abu-Ghazaleh, N.: Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT (2018)

37. Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B.: Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In: USENIX Security Symposium (2017)
38. Lee, S., Shih, M., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium (2017)
39. Levin, J.: Mac OS X and IOS Internals: To the Apple's Core. John Wiley & Sons (2012)
40. Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C.m.t.n., Mangard, S.: Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS (2017)
41. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium (2018)
42. Maisuradze, G., Rossow, C.: ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS (2018)
43. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS (2017)
44. Microsoft: Azure serverless computing (2019), https://azure.microsoft.com/en-us/overview/serverless-computing/
45. Microsoft Techcommunity: Hyper-V HyperClear Mitigation for L1 Terminal Fault (2018), https://techcommunity.microsoft.com/t5/Virtualization/Hyper-V-HyperClear-Mitigation-for-L1-Terminal-Fault/ba-p/382429
46. Mozilla: Javascript data structures (2019), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures
47. Nilsson, A., Nikbakht Bideh, P., Brorsson, J.: A Survey of Published Attacks on Intel SGX (2020)
48. OpenSSL: OpenSSL: The Open Source toolkit for SSL/TLS (2019), http://www.openssl.org
49. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS (2015)
50. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium (2016)
51. Rebeiro, C., Mukhopadhyay, D., Takahashi, J., Fukunaga, T.: Cache timing attacks on clefia. In: International Conference on Cryptology in India (2009)
52. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-flight Data Load. In: S&P (2019)
53. Schwarz, M., Canella, C., Giner, L., Gruss, D.: Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. arXiv:1905.05725 (2019)
54. Schwarz, M., Gruss, D., Lipp, M., Clémentine, M., Schuster, T., Fogh, A., Mangard, S.: Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. AsiaCCS (2018)
55. Schwarz, M., Gruss, D., Weiser, S., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA (2017)
56. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS (2019)
57. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC (2017)

58. Schwarz, M., Schwarzl, M., Lipp, M., Gruss, D.: NetSpectre: Read Arbitrary Memory over Network. In: ESORICS (2019)
59. Schwarz, M., Weiser, S., Gruss, D.: Practical Enclave Malware with Intel SGX. In: DIMVA (2019)
60. Seaborn, M., Dullien, T.: Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat Briefings (2015)
61. Slashdot EditorDavid: Two Linux Kernels Revert Performance-Killing Spectre Patches (2019), https://linux.slashdot.org/story/18/11/24/2320228/two-linux-kernels-revert-performance-killing-spectre-patches
62. Stecklina, J.: An demonstrator for the L1TF/Foreshadow vulnerability (2019), {https://github.com/blitz/l1tf-demo}
63. Turner, P.: Retpoline: a software construct for preventing branch-target-injection (2018), https://support.google.com/faqs/answer/7625886
64. Ubuntu Security Team: L1 Terminal Fault (L1TF) (2019), https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/L1TF
65. V8 team: v8 - Adding BigInts to V8 (2018), https://v8.dev/blog/bigint
66. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium (2018)
67. Van Bulck, J., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yuval, Y., Sunar, B., Gruss, D., Piessens, F.: LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P (2020)
68. Viswanathan, V.: 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
69. Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution (2018), https://foreshadowattack.eu/foreshadow-NG.pdf
70. Wu, Z., Xu, Z., Wang, H.: Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. ACM Transactions on Networking (2014)
71. xenbits: Cache-load gadgets exploitable with L1TF (2019), https://xenbits.xen.org/xsa/advisory-289.html
72. Xiao, Y., Zhang, Y., Teodorescu, R.: SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In: NDSS (2020)
73. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)

A Extracting Hypotheses from Previous Works

The hypotheses are extracted from previous works as detailed in this section. The footnotes for each hypothesis provide the exact part of the previous work that we reference.

H1 the `prefetch` instruction (to instruct the prefetcher to prefetch);[2]

[2] "Our attacks are based on weaknesses in the hardware design of prefetch instructions" [17].

H2 the value stored in the register used by the `prefetch` instruction (to indicate which address the prefetcher should prefetch);[3]
H3 the `sched_yield` syscall (to give time to the prefetcher);[4]
H4 the use of the `userspace_accessible` bit (as kernel addresses could otherwise not be translated in a user context);[5]
H5 an Intel CPU – the "prefetching" effect only occurs on Intel CPUs, and other CPU vendors are not affected.[6]

The original paper also describes that "delays were introduced to lower the pressure on the prefetcher" [17]. In fact, this was done via recompilation. Note that recompilation with additional code inserted may have side effects such as a different register allocation, which we find to be an important influence factor to the attack.

B Actual Spectre V2 gadget in Linux kernel

We analyzed the Linux kernel 4.16.18 and used the GNU debugger(GDB) to debug our kernel. As our target syscall we analyzed the path of the `sched_yield` syscall. We used the same experiment, which fills all general-purpose registers with the corresponding DPM address, perform `sched_yield` and verify the speculative dereference with Flush+Reload. We repeat this experiment 10 000 000 times. We analyzed each indirect branch in this code path and replaced the indirect call/jump with a retpolined version. Furthermore, we analyzed all general-purpose registers and traced their content if the DPM-address is still valid in some registers. By systematically retpolining the indirect branches, we observed that the indirect call `current->sched_class->yield_task(rq);` in the function `sys_sched_yield` causes the main leakage. We set a breakpoint to this function and observed that four general-purpose registers (`%rcx,%rsi,%r8,%r9`) still contain the kernel address we set in our experiment.

Listing 4 lists the register values before the indirect call is performed. By only filling one of those registers, we observed that the value `%rsi` causes the main leakage and is speculatively dereferenced in the kernel. If we retpoline this branch, there are nearly no more speculative dereferences for our experiment. Therefore, we conclude that this branch gets mispredicted and causes speculative dereferences of our address chosen in our experiment. We further analyzed where the speculative dereference happended and single-stepped the

[3] "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." [17] with emphasis added.
[4] "[...] delays were introduced to lower the pressure on the prefetcher." [17]. These delays were implemented using a different number of `sched_yield` system calls, as can also be seen in the original attack code [20].
[5] "Prefetch can fetch inaccessible privileged memory into various caches on Intel x86." [17] and corresponding NaCl results.
[6] "[...] we were not able to build an address-translation oracle on [ARM] Android. As the prefetch instructions do not prefetch kernel addresses [...]" [17] describing why it does not work on ARM-based Android devices.

Table 1: 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

`sched_yield` syscall. In the function `put_prev_task_fair`, the `%rsi` register is dereferenced. To check whether this dereference cause the leakage, we add an `lfence` instruction at the beginning of the function. We run the same experiment again and observe almost no cache fetches on our address. Listing 4 shows the execution trace of the function. The `%rsi` register is dereferenced in line 48

C Mistraining BTB for `sched_yield`

We evaluate the mistraining of the BTB by calling different syscalls, fill all general-purpose registers with direct-physical map address and call `sched_yield`. 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 1 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.

D Speculative Dereference Gadget in SGX

Listing 5 shows a minimal example of introducing a speculative-dereference gadget that can be used for *Dereference Trap* (cf. Section 6). The virtual functions are implemented using *vtable*s for which the compiler emits an indirect call in Line 14. 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.

```
; sys_sched_yield
0xffffffff8106b610 <+0>:     push   %rbp
0xffffffff8106b611 <+1>:     mov    %rsp,%rbp
0xffffffff8106b614 <+4>:     push   %rbx
0xffffffff8106b615 <+5>:     cli
0xffffffff8106b616 <+6>:     nopw   0x0(%rax,%rax,1)
0xffffffff8106b61c <+12>:    mov    $0x1dd40,%rbx
0xffffffff8106b623 <+19>:    add    %gs:0x7efa3b0d(%rip),%rbx        # 0xf138 <this_cpu_off>
0xffffffff8106b62b <+27>:    mov    %rbx,%rdi
0xffffffff8106b62e <+30>:    callq  0xffffffff81538a30 <_raw_spin_lock>
0xffffffff8106b633 <+35>:    mov    %gs:0x14d00,%rax
0xffffffff8106b63c <+44>:    mov    0x78(%rax),%rax
0xffffffff8106b640 <+48>:    mov    %rbx,%rdi
0xffffffff8106b643 <+51>:    mov    0x18(%rax),%rax
0xffffffff8106b647 <+55>:    callq  0xffffffff81802000 <__x86_indirect_thunk_rax>
;indirect call to current->sched_class->yield_task(rq);
0xffffffff81802000 <+0>: jmpq *%rax
; info registers
$rax   : 0xffffffff8106e590  ->  0x56415741e5894855  ->  0x56415741e5894855
$rbx   : 0xffff880007c1dd40  ->  0x0000000100000001  ->  0x0000000100000001
$rcx   : 0xffff880001cc0000  ->  0x000000000000009f  ->  0x000000000000009f
$rdx   : 0x0000000000000001  ->  0x0000000000000001
$rsp   : 0xffffc90000183ed0  ->  0xffffc90000183f58  ->  0xffff880001cc0000  ->  0
      x000000000000009f  ->  0x000000000000009f
$rbp   : 0xffffc90000183ed8  ->  0xffffc90000183f48  ->  0x0000000000000000  ->  0
      x0000000000000000
$rsi   : 0xffff880001cc0000  ->  0x000000000000009f  ->  0x000000000000009f
$rdi   : 0xffff880007c1dd40  ->  0x0000000100000001  ->  0x0000000100000001
$rip   : 0xffffffff8106b660  ->  0x0003c6ffffffeae8  ->  0x0003c6ffffffeae8
$r8    : 0xffff880001cc0000  ->  0x000000000000009f  ->  0x000000000000009f
$r9    : 0xffff880001cc0000  ->  0x000000000000009f  ->  0x000000000000009f
$r10   : 0x0000000000000000  ->  0x0000000000000000
$r11   : 0x0000000000000000  ->  0x0000000000000000
$r12   : 0x0000000000000000  ->  0x0000000000000000
$r13   : 0x0000000000000000  ->  0x0000000000000000
$r14   : 0x0000000000000000  ->  0x0000000000000000
$r15   : 0x0000000000000000  ->  0x0000000000000000
; SPECULATION STARTS HERE
ffffffff8106df10 <put_prev_task_fair>:
ffffffff8106df10:    55                       push   %rbp
ffffffff8106df11:    48 89 e5                 mov    %rsp,%rbp
ffffffff8106df14:    41 57                    push   %r15
ffffffff8106df16:    41 56                    push   %r14
ffffffff8106df18:    41 55                    push   %r13
ffffffff8106df1a:    49 89 f5                 mov    %rsi,%r13
ffffffff8106df1d:    41 54                    push   %r12
ffffffff8106df1f:    49 83 ed 80              sub    $0xffffffffffffff80,%r13
ffffffff8106df23:    53                       push   %rbx
ffffffff8106df24:    74 27                    je     ffffffff8106df4d <put_prev_task_fair+0
      x3d>
ffffffff8106df26: 8b 46 3c mov 0x3c(%rsi),%eax
ffffffff8106df29:    8b 96 c0 00 00 00        mov    0xc0(%rsi),%edx
```

Listing 4: Register values when setting a breakpoint at `current->sched_class->yield_task(rq)` still contain the direct phyiscal map address in `%rcx,%rdi,%r8,%r9`. The indirect jmp is misspeculated into the function `put_prev_task_fair` and performs a dereference of the value in $\%rsi$.

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

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

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

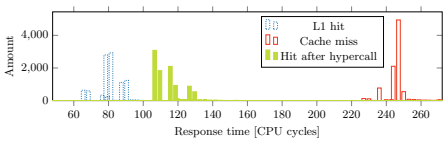

Fig. 4: Timings of a cached and uncached variable and the access time after a hypercall in a Ubuntu VM on Hyper-V.

to speculative execution of `Dummy::print`. However, the dereferenced member is now the secret (Line 11) 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.

E 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. [40] reported that web requests from JavaScript trigger interrupts from within the browser.

F 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

```
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.

accessing the variable and measuring 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 4) 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 ≈ 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 documen-

tation, 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.

G Evaluation Framework for Speculative Dereferencing in Syscalls

We created 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 lists the results of our evalution. As can be seen the `pipe` syscall achieves the highest 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	100.00 %	99.35 %	100.00 %	100.00 %
	send-to	99.90 %	99.60 %	100.00 %	100.00 %
	geteuid	99.90 %	99.61 %	100.00 %	100.00 %
	stat	99.90 %	99.55 %	99.90 %	100.00 %
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	21.20 %	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 %