AMD Prefetch Attacks through Power and Time

Moritz Lipp
Graz University of Technology

Daniel Gruss
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Modern operating systems fundamentally rely on the strict isolation of user applications from the kernel. This isolation is enforced by the hardware. On Intel CPUs, this isolation has been shown to be imperfect, for instance, with the prefetch side-channel. With Meltdown, it was even completely circumvented. Both the prefetch side channel and Meltdown have been mitigated with the same software patch on Intel. As AMD is believed to be not vulnerable to these attacks, this software patch is not active by default on AMD CPUs.

In this paper, we show that the isolation on AMD CPUs suffers from the same type of side-channel leakage. We discover timing and power variations of the prefetch instruction that can be observed from unprivileged user space. In contrast to previous work on prefetch attacks on Intel, we show that the prefetch instruction on AMD leaks even more information. We demonstrate the significance of this side channel with multiple case studies in real-world scenarios. We demonstrate the first microarchitectural break of (fine-grained) KASLR on AMD CPUs. We monitor kernel activity, e.g., if audio is played over Bluetooth, and establish a covert channel. Finally, we even leak kernel memory with 52.85 B/s with simple Spectre gadgets in the Linux kernel. We show that stronger page table isolation should be activated on AMD CPUs by default to mitigate our presented attacks successfully.

1 Introduction

Many performance optimizations in modern CPUs rely on predicting the control and data flow of applications. Thus, a CPU contains many microarchitectural elements for prediction, such as branch predictors or hardware prefetchers. Still, it can help if a developer or compiler provides the CPU with additional hints on data that will likely be accessed or modified next, e.g., the family of `prefetch` instructions on x86. The CPU can either ignore the hint or load the requested virtual address and potentially subsequent addresses into the cache level specified by the used prefetch instruction.

The prefetch instructions have the interesting property that they cannot cause an exception. When providing an invalid virtual address, the instruction is simply ignored. This is true for virtual addresses that are not backed by physical addresses, non-canonical addresses, and inaccessible addresses, such as kernel addresses. Hence, this instruction can be used without sanity checks, e.g., to prefetch the next pointer of a linked list, even if the next pointer is not valid [2–4].

The prefetch instructions have been investigated on Intel CPUs for side-channel leakage. Gruss et al. [25] showed that the prefetch instructions leak information about the paging hierarchy of a virtual address, *i.e.*, on which page table level a page table walk is aborted. As a mitigation against prefetch side channels, Gruss et al. [24] suggested stronger kernel isolation, which was adopted against Meltdown [42] and is widely deployed on systems with Intel CPUs [18]. Canella et al. [12] showed that this attack can also be mitigated by mapping all pages in the kernel with dummy pages, ensuring that the page table walk always ends at the same level. Further, Gruss et al. [25] showed that in rare cases, the prefetch instructions are not ignored for inaccessible addresses, leading to the caching of kernel data. However, Schwarzl et al. [60] showed that this was wrongly attributed to the prefetch instructions and is mitigated by the existing Spectre mitigations [6, 31].

In this paper, we show that there is indeed side-channel leakage caused by the prefetch instructions on AMD. We observe that the timing of the prefetch instructions on kernel addresses depends on which page table level the page table walk aborts, similarly to the effect on Intel but with an inverse timing distribution. Beyond the leakage described by Gruss et al. [25], we can also infer the TLB state of these addresses from the timing. Based on the TLB state, we can mount microarchitectural attacks that previously were only possible on Intel CPUs, such as the attacks demonstrated by Gras et al. [22] with TLBleed, Schwarz et al. [54] with Store-to-Leak forwarding, and Canella et al. [11] with Fallout.

Specifically, we show the first full microarchitectural KASLR (kernel address space layout randomization) break on AMD that works on all major operating systems. We demonstrate the KASLR break on laptops, desktop PCs, and from within virtual machines on the Amazon EC2 cloud. Furthermore, we demonstrate the first break of fine-grained KASLR

(FGKASLR) active on the Linux kernel. With a runtime of 0.15 s and a reliability of 100 %, this KASLR break is on par with state-of-the-art microarchitectural KASLR breaks for Intel CPUs [12, 25, 29, 35, 54]. Additionally, we demonstrate that the side-channel leakage of the prefetch instructions allows spying on kernel activity. As the timing of the prefetch instructions does not only rely on the page-table level but additionally on the TLB state, our attack leaks whether the kernel currently uses a targeted kernel page. Using Spectre gadgets, we show that this side channel can leak data from the kernel with up to 58.98 B/s without shared memory and active supervisor mode access prevention (SMAP). Thus, this prefetch leakage is an alternative covert channel to mount Spectre attacks [38] without relying on the cache.

Furthermore, we show that the side-channel leakage is not limited to timing differences but can also be exploited through power consumption. Since the Zen (family 17H) microarchitecture, AMD CPUs provide a Running Average Power Limit (RAPL) interface [7]. The recently released Linux 5.8 kernel includes the `amd_energy` driver [14] providing unprivileged access to per-core energy measurements. Consequently, with this kernel version, AMD CPUs are vulnerable to power side-channel attacks from user space as an alternative to timing-based attacks.

Thus, this paper shows that AMD CPUs are not immune to attacks on the isolation boundary between user and kernel space. Hence, stronger kernel isolation [24] is not only useful for Meltdown-affected Intel CPUs but also for AMD CPUs, as it protects against a range of attacks with acceptable overhead [23].

Contributions. The main contributions of this work are:

1. We analyze the side-channel leakage of prefetch instructions on AMD CPUs.
2. We present the first full microarchitectural KASLR break, including fine-grained KASLR, on AMD CPUs.
3. We show that we can not only spy on kernel activity but also leak kernel memory using a Spectre-type attack.
4. We demonstrate the first software-based power side-channel attack on AMD CPUs.
5. We evaluate the effectiveness of existing countermeasures against these attacks.

Responsible Disclosure We responsibly disclosed parts of our findings to AMD on June 16th, 2020, and the remaining parts on November 24th, 2020. AMD acknowledged our findings and provided feedback on February 16th, 2021.

Outline. The paper is organized as follows. In Section 2, we provide the necessary background. In Section 3, we analyze the side channel and present the exploitation primitives. Section 4 presents case studies to demonstrate the leakage in real-world scenarios. Section 5 discusses countermeasures preventing exploitation. In Section 6, we discuss implications of our attacks and related work. We conclude in Section 7.

2 Background

In this section, we provide background on virtual memory, prefetching, address space layout randomization, and RAPL.

2.1 Virtual Memory

To achieve memory isolation, modern processors support virtual memory as an abstraction layer to the system's physical memory. For this purpose, the memory is organized in so-called pages. For each process, the operating system maps virtual pages to physical pages, creating a virtual address space. By using multi-level page translation tables, the processor resolves virtual addresses to physical addresses. The root of the translation tables of a process is in a dedicated processor register, e.g., the CR3 register on x86-64 architectures. On a context switch, the operating system switches to the address space of the next process by updating this register. In addition to the physical address, translation tables also store page properties, e.g., whether it can be accessed, written to, executed, or accessed from user space.

On modern Intel and AMD processors, these translation tables have 4 levels. However, with Intel's recent Ice Lake microarchitecture, support for 5-level paging has been introduced. Each paging structure is 4 kB in size and comprises 512 entries of each 8 B. Thus, with 4 levels, bits 0 to 47 of the virtual address are used; with 5 levels, bits 0 to 56 are used to index the different page table levels. With 5-level paging, the top-most level is the page map level 5 (PML5). It divides the 57-bit address space into 512 entries. Each PML5E entry maps to a page map level 4 (PML4). With 4-level paging, the PML4 is the top-most level. The PML4 contains 512 entries, each mapping to a page directory pointer table (PDPT). Each PDPT entry defines a 1 GB region of physical memory (a 1 GB page) or maps to a page directory (PD). Each PD entry either maps a 2 MB region of physical memory (a 2 MB page) or maps to a page table (PT). Each PT entry then maps a 4 kB page of physical memory.

Translation-Lookaside Buffer (TLB). The CPU resolves virtual addresses to physical addresses by walking the page tables, which are stored in physical memory. To speed up the translation, the processor uses special caches called translation-lookaside buffers (TLB) to cache page-table entries. The TLB is then queried for the entry of a virtual address, and only if the translation has not been cached before, the CPU is required to walk the page-table entries.

AMD Zen CPUs have a dedicated multi-level TLB for the instruction and data cache [4]. While the L1 TLB can store entries for all page sizes, the L2 TLB can store entries for 4 kB and 2 MB pages, and additionally, page-directory entries (PDEs) for faster page walks [4]. In contrast to AMD, Intel has a shared L2 TLB [32].

On Intel processors, in addition to TLBs, so-called *page-translation caches* (or paging-structure caches) [33] are used to buffer the translated linear addresses and property bits of the paging tables. On a TLB lookup, the paging-structure caches are queried to speed up the translation process. Van Schaik et al. [63] reverse-engineered the internal architecture, size, and behavior of these caches.

2.2 Prefetch

To speed up access to frequently used memory locations, the CPU uses multiple levels of caches to store data and instructions. Furthermore, CPUs use different hardware prefetchers that preload instructions and data into the cache based on the behavior of the running program. For instance, the adjacent cache-line prefetcher caches data that is adjacent to the data currently being loaded [28]. Developers and compilers can also use instructions to hint the CPU to prefetch specific addresses to a cache specified by a locality hint [3, §3.9.6.1]. With `prefetcht0`, data should be prefetched to all cache levels while `prefetcht1` should prefetch into level 2 and higher, and `prefetcht2` should prefetch to level 3 cache and higher. A non-temporal prefetch hint (`prefetchnta`) should prefetch data into a non-temporal cache structure close to the CPU. On AMD CPUs, such cache lines are marked for *quick eviction* [4] as they are likely used only once.

Software prefetches are an important way to improve the performance of an application and are sometimes automatically inserted by the compiler, e.g., for loops. It must be noted that prefetch instructions can, by definition, not raise any exception (besides an undefined instruction exception if the `LOCK` prefix is used). Hence, even when executed with invalid virtual addresses as an argument, e.g., non-present, non-canonical, and inaccessible addresses like kernel addresses, prefetch instructions do not raise an exception. Thus, it can be safely used within a loop iteration even if the next address used is invalid, e.g., a null pointer. Gruss et al. [25] showed that prefetch instructions leak information on Intel CPUs about the paging hierarchy, *i.e.*, on which page-table level a page-table walk is aborted. Their further observation that inaccessible virtual addresses are prefetched into the cache has been invalidated by Schwarzl et al. [60] and shown that the actual cause is speculative execution in the kernel.

2.3 Address-Space Layout Randomization

Address-space layout randomization (ASLR) is a lightweight defense against memory corruption bugs. Kernel ASLR (KASLR) applies ASLR in the kernel, randomizing the locations of kernel code, data, and drivers on every boot, to impede exploitation of kernel bugs. However, various side-channel attacks have been demonstrated in the past that either reduce the entropy of KASLR or break it entirely. Hund et al. [29] presented the double page-fault attack, measuring the execution time of the kernel page-fault handler. If a kernel address is accessed, the translation entries are copied into the TLB, although the kernel address is user inaccessible. The attacker measures the execution time of a second page fault to the same address, which is faster if the memory location is valid as the translation is cached in the TLB. Jang et al. [35] exploited the same effect by utilizing Intel TSX transactions. If a page fault occurs within a transaction, it is aborted without any operating system interaction. This allows an attacker to learn which kernel memory locations are valid.

Gruss et al. [25] exploit the software prefetch instruction that leaks information on the paging hierarchy of a virtual address. By observing the instruction's execution time, the attacker learns not only whether an inaccessible address is valid but also the corresponding page size.

Lipp et al. [40] exploited AMD's L1D cache way predictor to reduce the entropy of KASLR of the Linux kernel. Canella et al. [12] exploited the behavior of hardware mitigations against Meltdown [42] to break KASLR. Koschel et al. [39] exploited tagged TLBs to break KASLR even in the face of state-of-the-art mitigations. Gras et al. [21] exploited that page-table pages are stored in the last-level cache to break code and heap ASLR from JavaScript.

2.4 RAPL

With the Sandy Bridge microarchitecture, Intel introduced the Intel Running Average Power Limit (RAPL) mechanism to ensure the processor remains within desired thermal and power constraints [20]. To implement the software-level control logic to adjust the CPU frequency while ensuring the power limits, it is necessary to provide feedback on the current energy consumption to software. The hardware implementation here may use voltage regulator current monitoring [20] or estimate the energy consumption in the core itself, e.g., in Sandy Bridge and Ivy Bridge microarchitectures. Intel defines different domains for RAPL, *i.e.*, package, power planes, and DRAM, and, thus, provides the energy consumption for the entire CPU package. On Linux, the power capping framework `powercap` exposes the Intel RAPL model-specific registers (MSRs) to unprivileged user-space access via `sysfs`.

Since the Zen microarchitecture, AMD also provides an interface that is partially compatible with Intel RAPL [7]. In contrast, AMD's implementation provides per-core counters. With Linux kernel 5.8, this interface also gives access to unprivileged user-space applications via the `procfs` interface [14]. However, a more recent patch limits the availability of the interface to Rome CPUs as other microarchitectures apparently report unreliable measurements [49].

3 AMD Prefetch Side Channel

In this section, we show how the prefetch instructions on AMD leak side-channel information via timing and power

Table 1: CPU type, model, and microarchitecture for each AMD device under test.

Type	CPU	Microcode	μ-arch
Mobile	Ryzen 5 2500U	0x810100b	Zen
Desktop	Ryzen Threadripper 1920X	0x8001137	Zen
Desktop	Ryzen 5 3600	0x8701021	Zen 2
Desktop	Ryzen 7 3700X	0x8701021	Zen 2
Desktop	A10-7870K	0x6003106	Steamroller
Cloud	EPYC 7402P	0x830104d	Zen
Cloud	EPYC 7571	0x800126c	Zen

consumption and compare the leakage to known leakage on Intel CPUs. We present the three exploitation techniques, Prefetch+Time, Prefetch+Power, and TLB-Evict+Prefetch that we use in our case studies (cf. Section 4). All our code can be found in a GitHub repository.[1]

Experimental Setup. Throughout this work, we ran our experiments for the leakage analysis and the presented case studies (Section 4) on AMD mobile, desktop, and server CPUs (Table 1). In the mobile setting, we used a Ryzen 5 2500U CPU, in the server setting, an EPYC 7402P CPU and an AMD EPYC 7571 running on the Amazon AWS cloud, and for the desktop setting, we used a Ryzen Threadripper 1920X, a Ryzen 5 3600, a Ryzen 7 3700X, as well as an A10-7870K CPU. All tested devices run a recent Ubuntu, CentOS, or ArchLinux operating system using the default configuration with Linux kernels ranging from 3.10 to 5.9. We also used an Intel Core i7-8565U to compare measurements to Intel CPUs.

3.1 Leakage Analysis Primitives

To analyze the leakage of the prefetch instruction regarding power and time, we require high-resolution timers, performance counters, and the RAPL interface. In this section, we describe the used primitives for the analysis.

Timing. Since the Zen microarchitecture, AMD CPUs have reduced the update interval of the timestamp counter [40]. While it still provides timings in cycle resolution, the update interval is reduced to 20-40 cycles, depending on the specific CPU [40]. Especially with Zen 2, we only measured an update interval of 36 cycles for AMD Ryzen 7 3700X and AMD Ryzen 5 3600. While this is sufficient for cache attacks [34, 38, 40], it requires more repetition of the measurements to exploit the small timing differences of the prefetch instruction.

Hence, if available, we rely on `rdpru`, a previously unexplored CPU instruction introduced with the Zen 2 microarchitecture [3]. This instruction allows reading the `APERF` and `MPERF` MSRs from user space. The `MPERF` MSR is incremented by the CPU with the P0 frequency, and the `APERF` MSR is incremented by the actual clock cycles [5]. While the `rdpru` instruction can be disabled for unprivileged users by setting `CR4.TSD` [3, §3.2.5], it was enabled on all our tested systems running Ubuntu 20.04 .

To compare the `rdpru` instruction to `rdtsc(p)`, we evaluated the update frequency, reordering behavior, and register dependencies. We evaluated the update frequency of `rdtsc` and `rdpru` (`APERF` and `MPERF`) on an AMD Ryzen 5 3600 using the technique described by Lipp et al. [40]. The update interval for `rdtsc` is 36 cycles, the same as the interval for `MPERF` read using `rdpru`. However, when reading `APERF` using `rdpru`, we measure an update interval of 1 cycle. We also verified that these fast updates are reliable by measuring a linearly increasing number of `nop`s. While the execution time measured with `rdtsc` and `rdtscp` was always a multiple of 36, we measured cycle-accurate values with `rdpru`. Only `rdtscp` is documented to force the retirement of older instructions [3], measuring time with `rdtsc` requires additional serializing instructions. We analyzed the retirement behavior of `rdpru` by adding it behind the mispredicted branch in a Spectre-PHT attack. In case the instruction is serializing, such as a memory fence, the leakage is mitigated [6]. For `rdtsc` and `rdpru`, the leakage is not mitigated, while for `rdtscp`, the leakage is mitigated. From this, we infer that `rdpru` has the same reordering behavior as `rdtsc`. Finally, the register dependencies are the same as for `rdtscp`, `rdpru` modifies `RAX`, `RDX`, and `RCX`. Hence, on Zen 2, we can use `rdpru` as a drop-in replacement for `rdtsc` and also for `rdtscp` if we add additional fences. Using `rdpru` results in a high-resolution timer with an even higher resolution than the `rdtsc` instruction on Intel CPUs. Therefore, we use `rdpru` in all our experiments and case studies if available.

Power. Intel CPUs provide the Running Average Power Limit (RAPL) interface to provide power-management information to software. Previous work used this interface to distinguish cryptographic keys [44]. Since AMD Family 17h, if bit 14 of CPUID `0x80000007` EDX is set, AMD provides an interface that is partially compatible with Intel RAPL. The interface provides power consumption values for the core domain and package domains via the MSRs `CORE_ENERGY_STAT` and `PKG_ENERGY_STAT` respectively [5]. Since Linux kernel 5.8, this interface is available to unprivileged user-space applications via the `sysfs` interface. An advantage of AMD's implementation compared to Intel's implementation is that information is available per core, not only per package. Hence, this allows measurements with less noise. Linux also provides an additional interface for power measurement on AMD. With the Linux kernel 5.6, the `k10temp` driver shows the currents and voltages of the CPU core and SoC of Ryzen CPUs and exposes them via the `hwmon` interface to unprivileged users.

However, we only use the RAPL interface for our work and evaluated it regarding its resolution. For the RAPL-compatible interface accessible via the `sysfs` interface, we

[1] https://github.com/amdprefetch/amd-prefetch-attacks

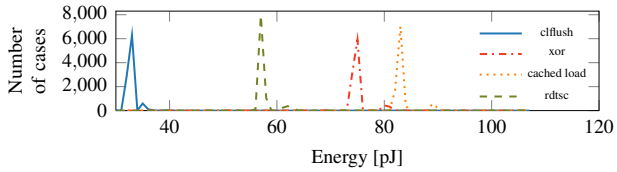

Figure 1: The energy consumption over 1 s of four different types of instructions measured using the RAPL interface.

measure an update interval of 1002.618 µs ($\sigma = 0.146, n = 5000$). This is close to the update interval stated by the CPU in the `RAPL_PWR_UNIT` MSR, which is 976 µs. The reported power consumption is in microjoule, with a resolution of 15.3 µJ as reported by the `RAPL_PWR_UNIT` MSR. The resolution and update interval is not sufficient to distinguish single events, e.g., the type of instruction or whether a memory load is served from the cache or DRAM. However, when events are repeatable, the resolution and update interval is sufficient to measure such differences. As an example, Figure 1 shows the histograms of the power consumption for repeatedly executing various instructions for 1 s (the exact instruction sequences can be found in Appendix C).

Performance Counters. To analyze the root cause of the leakage, we rely on performance counters. As performance counters are typically only available to privileged users, we do not rely on performance counters for *exploiting* the leakage but only for *analyzing* the leakage. We mainly rely on performance counters directly related to prefetching, TLB events, as well as μop dispatching. Table 4 (Appendix A) contains an overview of the used counters.

For comparing the microarchitectural effects with similar effects on the Intel microarchitecture, we also rely on the equivalent performance counters on Intel.

3.2 Prefetch Leakage

Based on our analysis primitives, we investigate the information leakage of software prefetch instructions. We discover 2 different properties ($\mathbb{P}1$, $\mathbb{P}2$) that leak through the prefetch instruction. An unprivileged attacker from user space can measure these properties. While there are 4 different prefetch instructions (`prefetcht0`, `prefetcht1`, `prefetcht2`, `prefetchnta`), they only differ in the targeted cache level, which is not relevant for our attacks. Furthermore, AMD documents that `prefetcht0`, `prefetcht1`, and `prefetcht2` are implemented exactly the same and do not differ [2]. Hence, for the remainder of the paper, we do not further distinguish the different instructions. To ensure noise-free measurements for the analysis, we isolate the cores on which we measure using the `isolcpus` command-line option and fix the CPU frequency using the `cpupower` utility.

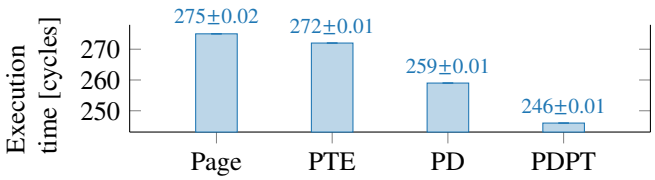

Figure 2: Execution time in cycles of `prefetch` including the measurement standard error (± cycles) on an AMD Ryzen 5 2500U for different number of page-table levels mapped.

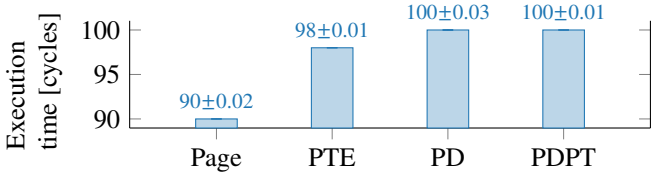

Figure 3: Execution time in cycles of `prefetch` including the measurement standard error (± cycles) on an Intel Core i7-8565U for different number of page-table levels mapped.

All measurements are repeated multiple times to average out possible outliers.

3.2.1 $\mathbb{P}1$: Page-Table Level

The first property is the information about the level on which the page-table walk stops. While it has been shown that this property can be leaked from prefetch instruction on Intel CPUs [25], it has not been analyzed on AMD CPUs. Surprisingly, although the prefetch instruction does leak information about the page-table level on AMD CPUs, the behavior is inverse to that on Intel CPUs.

We measure the execution time of the prefetch instruction for 4 different virtual addresses. These addresses are chosen such that the page-table walk ends at different page-table levels. For every address, we measure how long the fenced execution of one prefetch instruction to this address takes. To average out noise, we calculate the average execution timer over 10 million measurements. Intuitively, the fewer page tables are mapped for an address, the earlier the page-table walk aborts. Figure 2 shows the execution time in cycles, including the standard error, of the prefetch instruction for these virtual addresses. On AMD, the execution time of the prefetch instruction is proportional to the number of mapped page-table levels for the virtual address. On Intel, however, the execution time is *inversely* proportional to the number of mapped page-table levels, as observed by Gruss et al. [25]. We can confirm this, as shown in Figure 3.

The page-table level leakage is measurable both for accessible addresses, *i.e.*, addresses within the user address space of the application, and for inaccessible addresses, *i.e.*, kernel ad-

Table 2: The number of data TLB (dTLB) misses for 1000 prefetches of an inaccessible virtual address.

	dTLB misses / 1000 prefetches
Intel	1000 (`dtlb_load_misses.miss_causes_a_walk`)
AMD Zen	2000 (`ls_l1_d_tlb_miss.all`)
AMD Zen 2	1 (`ls_l1_d_tlb_miss.all`)

dresses mapped into the application. However, the leakage for inaccessible addresses is more interesting. If not mentioned otherwise, we always exploit the leakage for inaccessible addresses in the remainder of the paper.

To prefetch an address into the cache, the CPU requires the physical address of the target. The physical address can either be retrieved from the TLB if it is cached there, or the translation requires a page-table walk. As the permission, *i.e.*, whether a virtual address is user-accessible, is stored in the page-table entry (or TLB), a prefetch has to trigger a page-table walk if the address is not in the TLB.

From the timings (cf. Figure 2), we see how many page-table levels the page-table walker has to translate. Up to the page directory, the timing increases linearly with every page-table level. From the performance counters (cf. Table 2, we see that AMD Zen seems to trigger 2 page-table walks for an inaccessible address, whereas Intel only triggers a single page-table walk. Hence, the timing difference is amplified by the repeated page-table walks. We assume that this is an implementation flaw, as this is not the case anymore on the Zen 2 microarchitecture. On Zen 2, there is only one TLB miss on the first access, as Zen 2 also stores invalid translations in the TLB.

On Intel processors, the lookup direction for the TLB and page translation caches is from logically lower levels (*i.e.*, PTEs) to logically higher levels (*i.e.*, PML4Es). The timings indicate that on Intel, only the lowest missing level is fetched from memory, whereas all others are served from the page translation caches [8, 25, 33, 63].

3.2.2 $\mathbb{P}2$: TLB State

In addition to the page-table level, the prefetch instructions also leak the TLB state, *i.e.*, whether an address is currently cached in the TLB or not. As shown in Table 3, the software prefetch has a higher execution time if the address is not cached in the TLB. This property of the prefetch instructions is neither officially documented, nor was it measured on Intel CPU by Gruss et al. [25]. With the TLB state, it is possible to mount TLB attacks that have only been shown on Intel CPUs so far [22, 29, 35, 54].

We evaluate the influence on the TLB state for a virtual address *cached* in the TLB and for the same virtual address *flushed* from the TLB. We evaluate this leakage on an AMD Ryzen 5 2500U (Ubuntu 20.04 LTS, kernel 5.4.0-42). Prefetching an inaccessible virtual address that is cached in the TLB takes on average 125 cycles (n = 1 000 000). When invalidating the TLB before prefetching the address, the average time increases to 156 cycles (n = 1 000 000).

To further investigate the influence of memory types or properties of the page table entries, e.g., whether the page can be accessed from user space, we conducted the following experiment. As illustrated in Table 3, we set up pages with different permission bits set (executable, present, user-space accessible, dirty, global, accessed) and alternative memory types (write-back, write-combining, write-through, write-protected, uncacheable, uncacheable minus). For each microarchitecture, we measured the execution time of the prefetch instruction on the prepared page in two scenarios. First, we flushed the TLB entry using dedicated privileged functionality in the Linux kernel, measuring a TLB miss. Second, we leave the targeted page cached in the TLB to measure a TLB-hit by using the prefetch instruction in the kernel. To overcome measurement noise, we repeated the measurement 10 000 times.

We can observe timing differences between TLB hits and misses on all microarchitectures. We want to note that on the Zen microarchitecture, the access times differ manifold depending on the set properties. On the Intel microarchitecture, it is possible to differentiate between inaccessible pages that are present and not present. Furthermore, on the Zen 2 microarchitecture, almost all measurements that should hit the TLB yield the same execution time. We conclude that Zen 2 caches the translations regardless of the permission bits.

We observed that on AMD, the prefetch instruction sets the accessed bit in the page table entry. Thus, on a subsequent access or prefetch, the translation is served from the TLB. However, Intel CPUs do not seem to modify the accessed bits using a prefetch instruction, and, hence, the translation will not be cached in the TLB. Intel caches only translations if the accessed bits are set in all paging-structure entries [30].

An exception marks the not-accessed page (199 cycles) and the not-accessed kernel page with a higher timing on Zen 2 (220 cycles). In the case of pages where we unset the access bit, we omit the prefetch instruction in the kernel to avoid re-setting the accessed bit. We assume that the page-table walk caused by the prefetch instruction might require a microcode assist to set the accessed bit in the page table entry, as is the case on Intel CPUs [59]. The inaccessible address later causes a memory fault, treating the instruction itself as a no-operation (NOP) [47]. However, the microcode assist has already been dispatched. While AMD does not provide a dedicated performance counter for microcode assists like Intel (`other_assists.any`), we observed exactly 2 page table walks on the data side (`ls_tablewalker.dside`).

Furthermore, we observe a higher timing for page table entries marked as uncachable or write-combining. While this case is not documented for AMD, Intel describes that prefetches to such memory locations are ignored [32]. Since the prefetch instruction serves as a hint to the processor to

Table 3: Average prefetch execution time in cycles for pages with different properties on different microarchitectures (cf. Table 1). For the TLB-hit case in this table, addresses are accessed via a kernel module to ensure that their translation is cached in the TLB. As Zen 2 caches everything in the TLB, leakage is not visible in this table (cf. Table 2). On AMD, pages need to be accessed to be in the TLB. Hence, TLB hits cannot be measured for non-accessed pages (N/A). NX = non-executable, P = present, U = uncachable, D = dirty, G = global, A = accessed.

Permission Bits						**Memory Type**	**Zen** (Ryzen 5 2500U)		**Zen 2** (Ryzen 5 3600)		**Intel** (i7-8565U)	
NX	P	U	D	G	A		TLB-Hit	TLB-Miss	TLB-Hit	TLB-Miss	TLB-Hit	TLB_Miss
●	●	●	●	●	●	Write-Back	75 ($\sigma_{\bar{x}}$ = 0.03)	112 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	○	●	●	●	Write-Back	125 ($\sigma_{\bar{x}}$ = 0.03)	156 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.01)
●	○	●	●	●	●	Write-Back	149 ($\sigma_{\bar{x}}$ = 0.04)	149 ($\sigma_{\bar{x}}$ = 0.04)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.04)
●	○	○	●	●	●	Write-Back	149 ($\sigma_{\bar{x}}$ = 0.04)	149 ($\sigma_{\bar{x}}$ = 0.04)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	○	○	●	●	Write-Back	125 ($\sigma_{\bar{x}}$ = 0.03)	156 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.04)
○	●	○	●	●	●	Write-Back	125 ($\sigma_{\bar{x}}$ = 0.03)	156 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.02)
●	●	●	●	○	●	Write-Back	75 ($\sigma_{\bar{x}}$ = 0.03)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	●	●	●	○	Write-Back	N/A	142 ($\sigma_{\bar{x}}$ = 0.06)	N/A	199 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	○	●	●	○	Write-Back	N/A	155 ($\sigma_{\bar{x}}$ = 0.09)	N/A	220 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.05)
●	○	○	●	●	○	Write-Back	N/A	156 ($\sigma_{\bar{x}}$ = 0.16)	N/A	148 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.05)
●	●	●	●	●	●	Write-Combining	83 ($\sigma_{\bar{x}}$ = 0.06)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	●	●	●	●	Write-Through	75 ($\sigma_{\bar{x}}$ = 0.04)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.02)
●	●	●	●	●	●	Write-Protected	75 ($\sigma_{\bar{x}}$ = 0.04)	113 ($\sigma_{\bar{x}}$ = 0.05)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.04)
●	●	●	●	●	●	Uncacheable	83 ($\sigma_{\bar{x}}$ = 0.02)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.01)	90 ($\sigma_{\bar{x}}$ = 0.02)	119 ($\sigma_{\bar{x}}$ = 0.02)
●	●	●	●	●	●	Uncacheable Minus	86 ($\sigma_{\bar{x}}$ = 0.02)	113 ($\sigma_{\bar{x}}$ = 0.06)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.01)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.03)

load data closer to the executing core, speculatively accessing data, for example, from I/O devices, would break the functionality. We also experimentally confirmed that transient loads, which are similar to the prefetch instruction [2], cannot access uncached memory, as was also shown on Intel CPUs [57]. In contrast to the previous experiment, we did not observe an increased number of retired micro-operations or additional page table walks.

3.3 Difference to Normal Memory Loads

Software-prefetch instructions are a special kind of memory load. In contrast to all other operations that load data from memory, the prefetch instruction does not trigger any exception when specifying an invalid or inaccessible address. This behavior is documented explicitly by AMD: *"If the operand specifies an invalid memory address, no exception occurs, and the instruction has no effect."* [3, §3.9.6.1].

In addition to this well-documented difference, we also experimentally observe other differences.

Retirement. We observe that in contrast to regular data loads, loads triggered by the software-prefetch instructions never stall. Our experiments indicate that prefetch loads are immediately and ready-to-retire in the re-order buffer. Regular loads are only marked as complete and ready-to-retire when the target data is actually returned. On Intel CPUs, it is documented that prefetch instructions do not stall the normal instruction retirement [32] and retire after the address translation is completed [32].

We verified the stalling behavior by measuring the impact of fences on load instructions. AMD documents the load fence as *"All memory loads preceding the LFENCE (in program order) are completed prior to completing memory loads following the LFENCE."* [3, §3.9.2]. Hence, the `lfence` ensures that any load not marked as completed has to be completed first. In contrast, the `cpuid` instruction is fully serializing.

In our experimental setup on an AMD Ryzen 5 3600 (ArchLinux, kernel 5.10.39), we first load from an uncached memory location either via a regular load or a software-prefetch instruction. We then either add an `lfence` or a `cpuid` instruction as the subsequent instruction. We measure the execution time of this code snippet 1 000 000 times and ensure that no reordering of the measured instructions occurs by using memory fences (see Figure 12). For the regular load instruction, we observe that with a fully-serializing `cpuid`, we measure an average execution time of 945.71 cycles. For the prefetch instruction, we measure 842.96 cycles. However, when using an `lfence` instead of `cpuid`, we measure 814.44 cycles for the memory load but only 158.58 cycles for the prefetch instruction. Hence, we conclude that loads triggered by software-prefetch instructions are immediately marked as complete after the address translation and do not stall. In Appendix B, we evaluate an additional experiment measuring the execution time of many memory loads and respectively prefetch instructions that supports this statement. This also confirms the description by AMD that "*[...] a load instruction may cause a subsequent instruction to stall until the load completes, but a prefetch instruction will never cause such a stall.*" [3, §3.9.6.1].

Page-Table Walk. For valid, user-accessible data, both regular loads and prefetches can trigger a page-table walk in case the virtual address is not cached in the TLB. However, for prefetches, the TLB is only populated if the page-table walk does not cause a fault. This is not the case for the Zen 2 microarchitecture, and even invalid translations are cached. For inaccessible pages, *i.e.*, pages where the user-accessible bit is cleared, the fault also causes the prefetch instruction to

be re-issued, which is not the case for regular loads. Table 2 shows the number of data TLB misses when prefetching such an inaccessible address.

3.4 Exploitation

In this section, we define two exploitation techniques based on the leakage analysis primitives described in Section 3.1 and the properties that leak through the prefetch instruction, as shown in Section 3.2.

3.4.1 Prefetch+Time

Prefetch+Time directly exploits the execution time of the software prefetch instruction. This primitive is similar to the prefetch attacks known on Intel CPUs [25]. With Prefetch+Time, it is possible to directly leak the page-table level ($\mathbb{P}1$), as we show in the KASLR case study in Section 4.1. On the Zen architecture, it is furthermore possible to infer the TLB state ($\mathbb{P}2$) with this primitive.

The execution time of the prefetch instruction depends on the page-table level ($\mathbb{P}1$) at which the page-table walk stops and the TLB state ($\mathbb{P}2$) in case the page is present. However, only the page-table level ($\mathbb{P}1$) can be measured by timing a single prefetch instruction. For the other effects, the resolution of the high-resolution timer on AMD is in a similar range as the measured timing differences. Measuring these effects requires the measurement over multiple prefetch instructions.

In the case that prefetching the target does not change the state of the target, e.g., prefetching a non-present address, it is simply possible to measure the time it takes to execute n prefetch instructions to the same page. Alternatively, if the monitored address range spans multiple pages, Prefetch+Time can measure the cumulative execution time of prefetching each page, similar to Multi-Prime+Probe [58]. In other cases, where the TLB state is updated on a prefetch, e.g., when prefetching a kernel page on Zen 2, we have to resort to TLB-Evict+Prefetch (cf. Section 3.4.3).

3.4.2 Prefetch+Power

An alternative to measuring the execution time of the software prefetch instruction is to measure its power consumption via software interfaces. For Prefetch+Power, we rely on the unprivileged RAPL interface (cf. Section 3.1), which provides high-resolution power consumption of CPU cores. As with Prefetch+Time, with Prefetch+Power, it is possible to leak the page-table level ($\mathbb{P}1$), as we show in the KASLR case study in Section 4.1. Prefetch+Power has the advantage that it does not require any high-resolution timer.

As with the execution time, the power consumption of the software prefetch instruction increases with the number of page-table levels. With an update interval of only 976 µs and a resolution of 15.3 µJ, the resolution of the RAPL interface is not sufficient to reliably measure the TLB state ($\mathbb{P}2$). Moreover, as the power consumption cannot be measured for a single instruction but for the entire core activity, the measurement suffers from significant noise.Thus, similarly to Prefetch+Time, using Prefetch+Power to measure $\mathbb{P}1$also requires the measurement of multiple prefetch instructions. Again, this can only be applied if the state of the TLB does not change when prefetching, and thus it is mostly applicable to the Zen microarchitecture as Zen 2 caches invalid translations.

3.4.3 TLB-Evict+Prefetch

When leaking the TLB state ($\mathbb{P}2$) on Zen 2, or monitoring page activity via the TLB, neither Prefetch+Time nor Prefetch+Power are sufficient. In these cases, TLB-Evict+Prefetch can reliably leak the TLB state. As Zen 2 stores valid and invalid virtual to physical translations in the TLB, the measured timing or power differences are only measurable for the first prefetch. However, as the resolution of both primitives is insufficient to measure a single prefetch instruction, TLB-Evict+Prefetch has to first evict the corresponding TLB entry before issuing the prefetch instruction.

For the TLB eviction, we rely on existing eviction strategies [22, 63] for the best possible eviction rate. With this combination of TLB eviction and prefetching, we can reliably leak the page-translation level for inaccessible pages ($\mathbb{P}1$) on Zen 2, as well as the TLB state ($\mathbb{P}2$) on both Zen and Zen 2.

3.5 Covert Channel

In this section, we describe how TLB-Evict+Prefetch can be used to establish a covert channel between two unprivileged processes that either are not allowed to communicate over traditional communication mechanisms, e.g., sockets, or want to hide their communication.

The idea of the covert channel is to encode the information by caching translations of kernel pages in the TLB. While a context switch to a different address space requires addresses from the TLB to be flushed, it is not necessary that the entire TLB has to be flushed. As AMD is not vulnerable to the Meltdown [42] attack, its software-based mitigation kernel page table isolation (KPTI) [18] on Linux is not active. Therefore, kernel pages are still marked with the global bit and, thus, are not affected by the implicit TLB flush. Furthermore, if active, only translations with the same process-context identifier (PCID) are flushed from the TLB. To transmit a 1-bit, the unprivileged sender accesses an inaccessible kernel address (previously fixed on by both, the sender and receiver) using the prefetch instruction. While the actual data is not loaded to the cache, the translation of the address is cached in the TLB. To transmit a 0-bit, the sender idles. Using TLB-Evict+Prefetch, the receiver monitors the same inaccessible kernel address and decodes the transmitted bit by observing the execution time of the translation. Multiple bits are sent in parallel using multiple addresses.

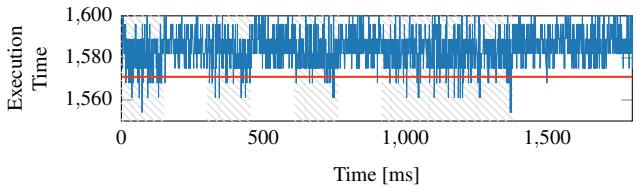

Figure 4: Transmission of bits `101010111000` using the prefetch covert channel. Gray areas highlight the transmission of a 1-bit, the red line is the threshold used by the receiver.

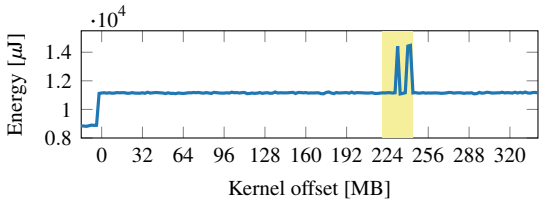

Figure 5: Power consumption when prefetching kernel addresses. The yellow rectangle shows the kernel location. The first spike is the location of the `__do_softirq` function.

Figure 4 illustrates the transmission of the bits `101010111000` over the covert channel. The receiver decodes every bit in each time frame by detecting time measurements under the threshold illustrated as the red line. In our unoptimized proof-of-concept implementation, we achieve a transmission rate of 6.66 bit/s with no transmission errors when using 20 addresses for the covert channel.

In contrast to other state-of-the-art covert channels [26, 43, 45], our covert channel has the limitation that it does not work across cores, likewise to the covert channel utilizing AMD's L1D way predictor [40]. Since the instruction and data TLB are not shared across hardware threads, a covert channel based on the prefetch side channel can not be established.

4 Case Studies

In this section, we present 4 case studies to show how software-prefetch instructions on AMD can be exploited. First, we show that KASLR can be derandomized reliably within seconds (Section 4.1) and that even the new fine-grained KASLR (FGKASLR) of the Linux kernel can be broken (Section 4.2). Furthermore, we show that the TLB leakage allows building direct attacks on the kernel. We demonstrate that it is possible to detect activity in the kernel inferring user activity (Section 4.3) or to leak kernel memory in combination with Spectre (Section 4.4).

4.1 Kernel Address Space Derandomization

In this section, we show that an unprivileged attacker can derandomize the kernel address space layout using RAPL. As there is no distinction between committed and non-committed instructions at the voltage regulator level, the power consumption also changes for transient instructions. Transient instructions are instructions that have been executed by an out-of-order processor but are never committed to the architectural state, e.g., instructions causing a fault [42] or instructions following a misspeculated branch [38]. The general concept of derandomizing the kernel address space is to distinguish between the transient access of mapped and unmapped kernel addresses via differences in power consumption. While on Intel CPUs, transiently accessing mapped and unmapped kernel addresses shows differences in timing [25, 29, 35] and store-forwarding behavior [11, 54], this effect has not been observed on AMD CPUs so far. However, in this section, we show that on AMD CPUs, the same result can be achieved by measuring the *power consumption* of transient kernel accesses.

Figure 5 shows the power consumption when prefetching a kernel address using the `prefetch` instruction 100 000 times. The yellow rectangle marks the location of the kernel. It can be seen that the power consumption differs for several pages that are mapped by the kernel. Specifically, we observed that the first spike in the power consumption reliably appears at the page used by the `__do_softirq` kernel function. By iterating over the page tables of the Linux kernel, we verified that the kernel page of `__do_softirq` function is the first 4 kB page used within the kernel image. As shown in Section 3.2.1, the execution time of the prefetch instruction is higher for a 4 kB page than a 2 MB page. In addition, the translation level for virtual addresses is also visible in the power consumption. In Figure 5, the first increase in the power consumption at offset '0' shows where the page directory for the kernel is mapped. Before that offset, there is no valid page directory.

In our experiments, we successfully used an AMD EPYC 7402P, AMD EPYC 7571, AMD Ryzen 5 3600, and an AMD Ryzen Threadripper. We also verified that the KASLR break works in a virtualized environment by mounting it on a T3a instance (AMD EPYC 7571) on the Amazon AWS cloud. As the kernel is 2 MB aligned with a range of 1 GB, there are 512 possible randomization offsets [54]. For each of these offsets, we measure the power consumption when prefetching the address. The average time to find the KASLR offset is 0.15 s which is similar to state-of-the-art KASLR breaks on Intel CPUs. In contrast to previous microarchitectural KASLR breaks [11, 12, 25, 29, 35, 54, 64], our KASLR break using power consumption is the first microarchitectural KASLR break that does not require any timing primitive.

As discussed in Section 3.2.2, Zen 2 seems to cache pages regardless of their permission bits in the TLB. On the contrary, Intel does not cache address translations for pages that are not present [25]. Thus, measuring over multiple executions of the prefetch instruction yields the same execution time on average and, thus, does not allow derandomizing the kernel offset. Thus, on Zen 2, instead of measuring the energy consumption

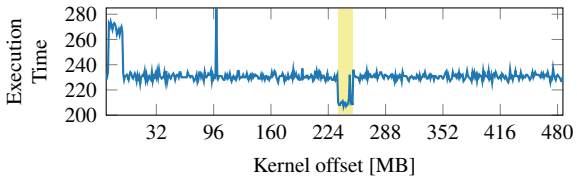

Figure 6: Execution time when prefetching kernel addresses. The yellow rectangle shows the actual location of the kernel, corresponding with the measured negatives spike.

over the prefetch instruction, we measure the execution time of a single prefetch instruction repeatedly. Since the address translation is cached, we invalidate it by evicting the TLB between each measurement. Figure 6 shows the recorded measurements over the possible offsets on an AMD Ryzen 7 3700X running a Linux kernel 5.4. The first negative spike corresponds to the start of the kernel, *i.e.*, `startup_64` located at `0xffff ffff a000 000`. With the same technique, we derandomized the kernel on the A10-7870K, running CentOS 7.8 with a Linux kernel 3.10.

4.2 Breaking Fine-Grained KASLR

In 2020, Intel proposed fine-grained KASLR (FGKASLR) for the Linux kernel. We show that even with this advanced randomization, it is possible to use the prefetch instruction to find the address of a function of interest. At the time of writing, this patch is not in the upstream kernel. However, the patch is stable and can already be applied to kernel 5.9. For the remainder of the section, we use Ubuntu 20.04 with kernel 5.9.0-rc6 and the latest version (version 5) of the FGKASLR patch [1]. With FGKASLR, the kernel image is also relocated to a random start address. In addition, the code of most functions is shuffled within the kernel image. This shuffling is done once at boot time. Hence, to find the address of a target function, it is required to find the base of the kernel image, and additionally, the position of the function within the kernel.

Unlike countermeasures such as KPTI [24], LAZARUS [17], or FLARE [12], FGKASLR does not change anything in the memory mappings surrounding the kernel image. Thus, we can use the same attack as described in Section 4.1 to find the beginning of the kernel image. The only difference is that at this point, we only know the base address of the kernel image but no addresses of any function. One exception is the CPU startup functionality, spanning 2 pages, is still at the base of the kernel image. This might already be sufficient for a return-oriented-programming attack or as a base for continuing with probing the kernel space. Even with FGKASLR, the kernel image does not have holes in the virtual memory. Thus, there is no risk in probing for values as long as the base address is known. Furthermore, an attacker could resort to speculative probing [19] given an exploitable Spectre gadget in the kernel.

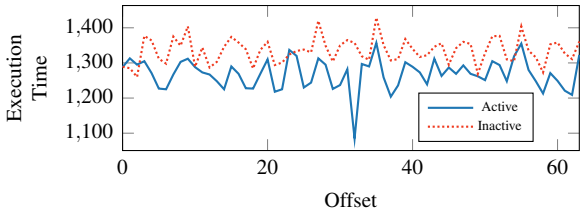

Figure 7: Template attack for page candidates of a kernel driver while triggering a supported `ioctl` syscall. The spike correlates with the kernel page containing the corresponding function, allowing to derandomize FGKASLR.

Function Activity Template Attacks. While we were only able to find the beginning of the kernel image if FGKASLR is active, we now propose a generic approach to find the address of a target function based on the concept of template attacks [10,13,27,56,65]. The principle idea of this technique is to constantly trigger the targeted event while in parallel probing each possible candidate. For instance, Gruss et al. [27] demonstrated cache template attacks by observing cache activity using a cache attack on shared libraries for different key input events. They identified different addresses correlating to keystrokes, enabling inter-keystroke timing attacks.

Similarly, we probe each candidate page within the possible range of the detected kernel image while triggering the execution of the targeted functionality or the access of the targeted data region. We perform the *profiling phase* of each address candidate with the following steps:

1. **Preparation**: In the preparation step, the state of the TLB is reset to a *known* state, *i.e.*, we evict the targeted address translation from the TLB.
2. **Trigger**: In the trigger step, the event that accesses or execute the targeted addresses is induced.
3. **Measurement**: In the last step, the execution time of the address translation is timed.

Each probed address with a low timing measurement is a candidate for the targeted address. If the template attack yields multiple candidate addresses, *i.e.*, if the event triggers accesses to multiple pages, another event could be profiled. This event should then access a subset of addresses as the initial event excluding the targeted address, allowing reducing the number of candidate addresses. However, if the exploit requiring the targeted address allows multiple attempts, the attacker can mount the attack for all candidate addresses.

Evaluation. We mount our attack on a sample kernel driver, as kernel exploits targeting a driver (e.g., the recent CVE-2019-18683) often require the knowledge of the address of the driver. To obtain the address of the kernel driver, we mount a function activity template attack on the located kernel image. We evaluated this attack on an AMD Ryzen 5 2500U running Linux 5.9 with FGKASLR. To reduce measurement noise, we repeat the necessary steps for each address 1500 times.

Figure 7 shows the averaged obtained measurements with TLB-Evict+Prefetch for the 64 candidate addresses for the target function. The spike at offset 32 correlates with the actual page of the targeted function triggered via the `ioctl`. In comparison, the dotted line shows the averaged obtained measurements when the target function is not active. Thus, combining template attacks with TLB-Evict+Prefetch allows us to successfully find kernel functions in a recent Linux kernel protected with FGKASLR.

4.3 Spying on Kernel Activity

In this section, we utilize TLB-Evict+Prefetch to monitor kernel activity. Specifically, we detect active audio transmissions via Bluetooth. For this case study, we assume that we have already de-randomized the kernel address space (cf. Section 4.1).

In contrast to the template attack on the kernel from Section 4.2, we do not trigger the kernel activity from the attacker application. Instead, we rely on an external event, *i.e.*, a Bluetooth communication, that is handled in the kernel on the same core as the attacking application is running. If such an event is triggered, e.g., by the connected Bluetooth device, the kernel interrupts the attacker application and continues executing in the Bluetooth module. For this, the memory management unit (MMU) has to translate the virtual addresses of the accessed or executed pages, transparently caching them in the TLB. By monitoring the address, and thus the TLB state ($\mathbb{P}2$) using TLB-Evict+Prefetch, an attacker can detect if these pages have been accessed. Our attack has the following steps constantly repeated by the attacker:

1. **Evict TLB**: The attacker evicts entries from the TLB by accessing arbitrary user space addresses.
2. **Scheduling/Interrupt**: The attacker process is interrupted to handle an incoming event, e.g., a Bluetooth packet. Alternatively, the attacker relinquishes the core, allowing other processes to execute. If the victim accesses the monitored code, its translation is cached in the TLB.
3. **Time**: The attacker uses Prefetch+Time, *i.e.*, measures the execution time of prefetching the monitored address. If a low timing has been obtained, the attacker can conclude that the monitored address has been accessed ($\mathbb{P}2$) and, thus, an event of interest is observed.

Evaluation. We mounted our proof-of-concept implementation of the attack on an AMD Ryzen 5 2500U CPU, running Ubuntu 20.04 LTS with Linux kernel 5.9. We pair a smartphone with the computer and target the default Bluetooth kernel module present on Ubuntu without any modifications.

Figure 8 shows the obtained measurements by the attacker process via TLB-Evict+Prefetch. In each attack iteration, we attack the same offset in the kernel module. From the plot, we can see that the Bluetooth module always shows a slight activity in the TLB, *i.e.*, the prefetch time is sometimes fast. We

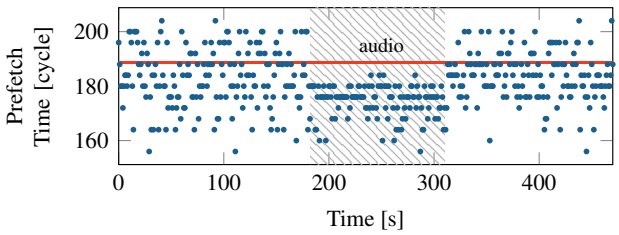

Figure 8: Detecting Bluetooth audio transmission with TLB-Evict+Prefetch. After 180 s, a 2:10 min song is played. Dots below the red line indicate that the page is in the TLB, *i.e.*, active, dots above the line indicate that the page is not in the TLB. During audio transmission, the page is always active.

assume that this is for keeping the connection alive. However, as soon as we start transmitting audio (gray area), the targeted address is always present in the TLB until we stop the audio transmission, leading to always fast prefetch times. Hence, we can introduce a static threshold (red line). If we do not measure any prefetch execution times above this threshold, the Bluetooth module is transmitting an audio stream.

One limitation of the attack on AMD CPUs is that there is no shared TLB between SMT threads. While the instruction and the data TLB are competitively shared, *i.e.*, the entries can be used by both threads, the entries are tagged using the thread ID [4]. Thus, entries can only be accessed by the owning thread of the entry. Therefore, cross-core monitoring is not possible on AMD CPUs.

4.4 Leaking Kernel Memory with Spectre

In this section, we combine Spectre with TLB-Evict+Prefetch to leak kernel memory from a recent Linux kernel. Since AMD CPUs are not affected by Meltdown [42], the kernel is still mapped in user space, as page table isolation [24] is not enabled. During transient execution, the kernel accesses an address based on a secret. Thereby, the address is loaded into the cache, and its translation stored in the TLB.

Most Spectre-type attacks use the cache as a covert channel and, thus, require shared memory between the kernel and user space. If the kernel tries to access a user-space address and SMAP is enabled [3, §5.6.6], the translation raises an exception. However, TLB-Evict+Prefetch can detect if a page translation of a kernel address is cached in the TLB (see Section 3).

To evaluate TLB-Evict+Prefetch for a Spectre-type attack, we implement a custom kernel module to which communication is enabled using `ioctl` from user space. The module contains a Spectre-PHT gadget accepting an `offset` variable as an user-controlled argument:

```
if (offset < data_len) tmp = LUT[data[offset] * 4096];
```

In contrast to the original Spectre attack [38], an adversary requires only full control over the `offset` but no control over the `LUT` address. Thus, our attack works without any shared

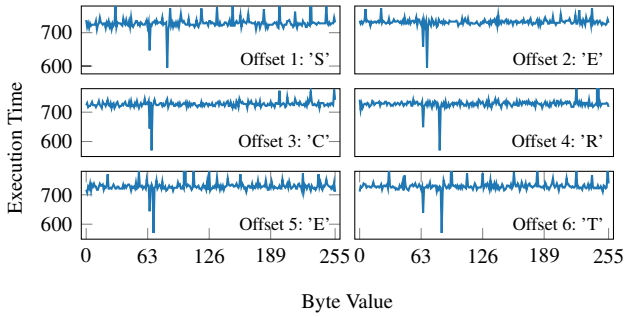

Figure 9: Leaking the secret string `SECRET` byte-by-byte from the Linux kernel. The lowest prefetch measurement for each kernel page correlates to the stored byte at the given offset.

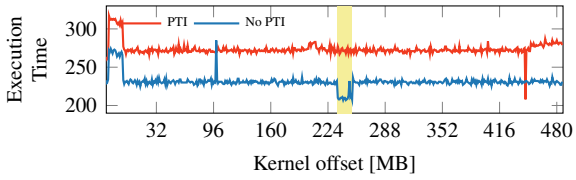

Figure 10: Access times when prefetching kernel addresses. The yellow rectangle shows the location of the kernel. While the KASLR break without KPTI has been successful (blue), activating KPTI on AMD CPUs eliminates the leakage (red).

memory between the kernel and user space. A limiting factor of this attack is the requirement of the knowledge of the kernel address that is accessed by the Spectre gadget if KASLR is active on the attacked system. However, in Section 4.1, we demonstrate that both Prefetch+Time and Prefetch+Power can derandomize the kernel and, thus, obtain the address. In contrast, when using Spectre with Prime+Probe on the kernel, an attacker requires the knowledge of physical addresses to build an efficient eviction set. Unprivileged access to this information is typically prevented [37], and attackers can only resort to less-efficient eviction sets or the use of other side-channel information in combination with huge pages (if available) [55].

In our attack, we first mistrain the branch predictor by repeatedly providing an index that is within the allowed range. The CPU follows the branch to access a fixed kernel-location `LUT` based on the value stored at `data+offset`. Then, we evict the TLB by accessing arbitrary user-space addresses. Next, we provide an out-of-bounds offset, letting the processor speculatively access a memory location based on the sensitive data located at the given offset. Using prefetch, we now measure if the kernel address for each of the 256 possible byte values has been loaded into the TLB. The kernel address with the lowest execution time for the prefetch instruction leaks the actual byte value at the given offset. To increase the likelihood that the processor actually speculatively accessed the address, we repeat this attack step multiple times.

We successfully recovered a secret string of 1000 characters from Linux kernel 5.10.39 running on an AMD Ryzen 5 3600. With our unoptimized proof-of-concept implementation, we can recover the secret bytes with a success rate of 96.7 % (σ = 4.29) and a leakage rate of up to 58.98 B/s (52.85 B/s on average, σ = 2.14, n = 100). In contrast, Lipp et al. [40] achieved a leakage rate of 0.66 B/s exploiting AMD's way prediction in a similar setting. Canella et al. [11] described a similar attack using *Speculative Fetch+Bounce* but did not provide any numbers. For comparison, cache-based covert channels that rely on shared memory between kernel and user space achieve leakage rates of up to 5000 B/s [38].

Figure 9 illustrates the obtained measurements of the execution time of the different kernel addresses for each out-of-bounds offset. One can clearly see that the lower values correlate with the secret byte.

5 Countermeasures

In this section, we discuss different countermeasures and mitigation strategies for the presented attacks.

Page Table Isolation. With KAISER [24], Gruss et al. presented a countermeasure to prevent microarchitectural attacks on the kernel by unmounting the kernel address space while in user space. Canella et al. [12] proposed to map the entire kernel address space using dummy mappings to prevent microarchitectural KASLR breaks. A similar concept was also presented by Gens et al. [17]. While dummy mappings prevent attacks based on the page-table-level leakage, they do not prevent TLB- or cache-line-based leakage. The KAISER technique, implemented as KPTI [18] on Linux and KVAShadow on Windows [36], prevents exploitation of Meltdown on Intel CPUs [23, 42]. With this paper, we show that such a stronger kernel isolation should also be used on AMD CPUs.

To verify if page-table isolation is sufficient on AMD CPUs to mitigate our attacks, we enforced KPTI (`pti=on`) on the Linux kernel 5.4 on an AMD Ryzen 7 3700X CPU. While running our KASLR break (see Section 4.1) was possible without PTI, we can see in Figure 10 that we are not able to locate the kernel image anymore as the kernel addresses are not mapped in user space. The negative spike at the end is the still remaining mapped entry region `__entry_text_start` that is required to handle interrupts and system calls.

FLARE. With FLARE [12], Canella et al. presented a generic mitigation against known microarchitectural KASLR breaks. It uses dummy mappings in the kernel space to eliminate timing differences between mapped and unmapped pages. We used the proof-of-concept implementation [12] to evaluate if it mitigates our KASLR break. Figure 11 shows the

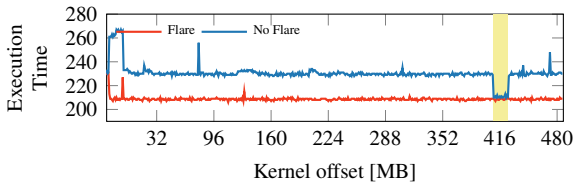

Figure 11: Access times when prefetching kernel addresses. The yellow rectangle shows the location of the kernel. While the KASLR break without FLARE has been successful (blue), activating it on AMD CPUs eliminates the leakage (red).

recorded traces of our KASLR break on a Ryzen 7 3700X. With FLARE activated, we cannot detect the kernel image. Note that FLARE does not prevent other presented attacks.

Prefetch Configuration MSRs. While AMD TLBs do not enable cross-core attacks by design, the timing side-channel leakage of the prefetch instruction is coherent with the caching properties of the TLB. Thus, it can not be resolved by different TLB hardware designs. However, we propose two alternative solutions to mitigate the presented side-channels in future CPUs. As the architectural behavior of the prefetch instruction is described merely as a hint, the processor is not required to execute it [3, §3.9.6.1]. Thus, the first and more aggressive solution is to introduce a bit in an MSR that, if set, treats every prefetch instruction as a NOP. To estimate the impact of this mitigation, we analyzed the prevalence of prefetch instructions. On Ubuntu 20.04, we only found 16 out of 1428 applications and libraries in the system folders that contain the prefetch instruction. Hence, we argue that the performance overhead of this mitigation would likely be not even noticeable.

The second alternative approach is to partially allow prefetching instruction on certain address ranges. Typically, the kernel is either mapped on the top or the bottom of the address space, *i.e.*, the most-significant bit of kernel addresses is either 0 or 1, depending on the operating system. We propose an MSR configuration that disables prefetching instructions for virtual addresses with either the most-significant bit set to 0, set to 1, or to disable it for all addresses. Thus, the operating system can disable prefetching instructions on kernel addresses, mitigating the attacks presented in this paper.

Restricting Access. With the Linux kernel 5.8, the `amd_energy` driver provides unprivileged access to the per-core energy measurements of AMD CPUs. To mitigate Prefetch+Power-type attacks, limiting user-space access is necessary to mitigate at least unprivileged attacks. Hence, as a result of this paper and PLATYPUS [41], AMD first restricted the access to privileged users, and Linux ultimately removed the driver in kernel 5.13 [46]. Further, the `k10temp` exposes the currents and voltages of the AMD CPUs to unprivileged users. While we did not evaluate this interface, we recommend restricting the interface to privileged users as well.

It is possible to prevent unprivileged usage of the high-resolution timers by setting the `CR4.TSD` bit [3, §3.2.5]. While setting this bit prevents an attacker from using `rdtsc`, `rdtscp`, and `rdpru`, it does not disable all timing primitives. A counting thread can be used as a high-resolution timing primitive to distinguish timing differences of a few cycles on AMD CPUs [40]. Moreover, we identified several widespread unprivileged applications that rely on `rdtsc`, including XWayland, adb, cargo, and Docker. Hence, restricting unprivileged access to high-resolution timers is not only ineffective against attacks, but also has adverse effects on benign applications.

6 Related Work & Discussion

In this section, we discuss related and future work.

Prefetcher. Gruss et al. [25] investigated the software prefetch instruction on Intel CPUs. They showed that the software prefetch instructions leak the page-table level of inaccessible virtual addresses and used this observation to build a KASLR break. Their other observation that the prefetch instruction allegedly prefetches inaccessible addresses into the cache, has been shown to be wrongly attributed to the prefetcher [60]. Schwarzl et al. [60] showed that the actual root cause of this observation is speculative execution caused by a Spectre gadget in the kernel. Shin et al. [61] exploited the hardware prefetchers on Intel CPUs to attack OpenSSL. Rohan et al. [52] reverse-engineered the hardware stream prefetcher on Intel CPUs and built a prefetch-based covert channel. In contrast to previous work, we are the first to explore the prefetch side channel on AMD CPUs.

TLB Attacks. Gruss et al. [25] exploited the TLB and address translation caches on Intel CPUs using the prefetch instruction. With TLBleed, Gras et al. [22] reverse-engineered undocumented address functions of TLBs on Intel CPUs and recovered EdDSA keys using a Prime+Probe-style attack on the TLB. Koschel et al. [39] exploit tagged TLBs to break KASLR even in the face of state-of-the-art mitigations. Schwarz et al. [54] leveraged the store buffer in combination with the TLB to break KASLR or to infer control flow of the kernel. However, their necessary attack primitives only apply to Intel CPUs as they were unsuccessful in reproducing them on either ARM or AMD CPUs. In this paper, we demonstrate that similar attacks can be performed on AMD CPUs.

Van Schaik [62, 63] reverse-engineered the MMU to build eviction sets for the TLB and translation caches on Intel, AMD, and ARM CPUs to recover AES keys in OpenSSL's T-Table implementation. Gras et al. [21] exploit the property that page-table pages are stored in the last-level cache of Intel CPUs to break code and heap ASLR from JavaScript.

Power Measurement. While hardware-based power analysis is usually conducted using physical probes [48, 53], software-based power analysis allows performing similar attacks without physical access to the attacked device. Gao et al. [16] used Intel RAPL within containers to co-locate containers on the same host. In 2017, Fusi [15] showed that the Intel RAPL interface can be used to observe whether branches have been taken or if data has been cached but concluded that the sampling rate to recover RSA keys is too low. In 2018, Mantel et al. [44] showed that it can be used to distinguish RSA keys with different Hamming weight using machine learning but did not demonstrate concrete attacks. However, in 2021, Lipp et al. [41] showed that the Intel RAPL interface can indeed be exploited to leak cryptographic key material and to break KASLR. They analyzed the side-channel leakage by showing that one can distinguish between instructions and the Hamming weight of operands and data. On Intel, they demonstrated the recovery of AES-NI and RSA keys in unprivileged and privileged attack scenarios. While they analyzed the leakage behavior of an AMD CPU, they did not demonstrate any attacks. With our KASLR break, we demonstrate the first software-based power side-channel attack on AMD.

In addition to the power measurement capabilities of the CPU, research has been conducted on devices that provide other means of energy measurement of the platform under attack. Qin et al. [51] and Yan et al. [67] monitored the system power information on mobile devices (voltage, current, battery charge) to distinguish different applications and websites or observing keystrokes. O'Flynn [50] demonstrated side-channel attacks using an onboard analog-to-digital converter to recover secrets processed in the secure world on a TrustZone-enabled device.

7 Conclusion

In this paper, we demonstrated that prefetch side channels undermine the isolation between user and kernel space on AMD CPUs. We introduced three exploit primitives, Prefetch+Time, Prefetch+Power, and TLB-Evict+Prefetch, that exploit timing and power variations of the prefetch instructions. We demonstrated the applicability in real-world scenarios to break (fine-grained) KASLR, monitor kernel activity, establish a covert channel, and even leak kernel memory with Spectre gadgets. Finally, we evaluated existing mitigations and discussed possible new mitigations to prevent the presented attacks.

Acknowledgments

We would like to thank our anonymous reviewers and in particular our shepherd, Anil Kurmus, for their feedback that helped improve this paper. 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 a generous gift from Intel. 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] Accardi, Kristen Carlson. Function Granular KASLR, 2020. URL: https://patchwork.kernel.org/project/kernel-hardening/list/?series=354389.

[2] Advanced Micro Devices Inc. *Software Optimization Guide for AMD Family 15h Processors*, January 2014.

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

[4] Advanced Micro Devices Inc. Software Optimization Guide for AMD Family 17h Processors, 2017.

[5] Advanced Micro Devices Inc. *Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh*, 3.03 edition, 7 2018.

[6] Advanced Micro Devices Inc. Software Techniques for Managing Speculation on AMD Processors, 2018. Revison 7.10.18.

[7] Advanced Micro Devices Inc. *AMD uProf User Guide*, 3.2 edition, 2019.

[8] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). *ACM SIGARCH*, 2010.

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandt ner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[10] Billy Brumley and Risto Hakala. Cache-Timing Template Attacks. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, 2009.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[12] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2020.

[13] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[14] Naveen Krishna Chatradhi. hwmon: Add amd_energy driver to report energy counters, 2020. URL: https://patchwork.kernel.org/patch/11496005/.

[15] Matteo Fusi. Information-Leakage Analysis Based on Hardware Performance Counters, 2017. URL: https://www.politesi.polimi.it/handle/10589/137507.

[16] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[17] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2017.

[18] Thomas Gleixner. x86/kpti: Kernel Page Table Isolation (was KAISER), 2017. URL: https://lkml.org/lkml/2017/12/4/709.

[19] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[20] Corey Gough, Ian Steiner, and Winston Saunders. *Energy Efficient Servers*. Apress, 2015.

[21] Ben Gras and Kaveh Razavi. ASLR on the Line: Practical Cache Attacks on the MMU. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[22] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[23] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. *USENIX ;login*, 2018.

[24] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (ESSoS)*, 2017.

[25] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[28] Hegde, Ravi. Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers, 2008.

[29] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[30] Intel. TLBs, Paging-Structure Caches, and Their Invalidation, 2007. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[31] Intel. Intel Analysis of Speculative Execution Side Channels, 2018. Revision 4.0.

[32] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

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

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2016.

[35] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[36] Ken Johnson. KVA Shadow: Mitigating Meltdown on Windows, Mar 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

[37] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace, 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[38] 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. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[39] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[40] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2020.

[41] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[42] 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. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[44] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. How Secure is Green IT? The Case of Software-Based Energy Side Channels. In *European Symposium on Research in Computer Security (ESORICS)*, 2018.

[45] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[46] Michael Larabel. AMD Energy Driver Booted From The Linux 5.13 Kernel, 2021. URL: https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.13-AMD-Energy-Removed.

[47] Mitchell, Kenneth. AMD Ryzen Processor Software Optimization, 2020. URL: http://gpuopen.com/wp-content/uploads/slides/GPUOpen_Let%E2%80%99sBuild2020_AMD%20Ryzen%E2%84%A2%20Processor%20Software%20Optimization.pdf.

[48] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. Precise Instruction-Level Side Channel Profiling of Embedded Processors. In *International Conference on Information Security Practice and Experience*, 2014.

[49] Naveen, Krishna Chatradhi. hwmon: amd_energy: match for supported models, 2020. URL: https://patchwork.kernel.org/patch/11646271/.

[50] Colin O'Flynn and Alex Dewar. On-Device Power Analysis Across Hardware Security Domains. *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2019.

[51] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In *IEEE International Conference on Big Data Science and Engineering (BigDataSE)*, 2018.

[52] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. Reverse engineering the stream prefetcher for profit. In *Security of Software/Hardware Interfaces (SILM) Workshop*, 2020.

[53] Sami Saab, Pankaj Rohatgi, and Craig Hampel. Side-channel protections for cryptographic instruction set extensions. *IACR Cryptology ePrint Archive*, 2016. URL: https://eprint.iacr.org/2016/700.

[54] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*, 2019.

[55] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[56] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[57] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In *Network*

and Distributed System Security Symposium (NDSS), 2020.

[58] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[59] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[60] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. In *Financial Cryptography and Data Security (FC)*, 2021.

[61] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[62] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.

[63] Stephan Van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. Revanc: A framework for reverse engineering hardware page table caches. In *ACM European Conference on Computer Systems (EuroSys)*, 2017.

[64] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium*, 2021.

[65] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security Symposium*, 2018.

[66] Wu, David and Kuepper, Joel. AssemblyLine, 2021. URL: https://github.com/0xADE1A1DE/AssemblyLine.

[67] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A Study on Power Side Channels on Mobile Devices. In *Symposium on Internetware*, 2015.

Table 4: Performance counters leveraged to analyze the prefetch leakage.

Category	Counter	Description
Prefetch	`ls_inef_sw_pref.mab_mch_cnt`	Software prefetches that did not fetch data outside of the processor core
	`ls_pref_instr_disp.load_prefetch_w`	Dispatched PREFETCHT0/1/2 instructions
	`ls_pref_instr_disp.prefetch_nta`	Dispatched PREFETCHNTA instructions
TLB	`ls_l1_d_tlb_miss.all`	L1 DTLB Miss or Reload (all sizes)

A Performance Counters

On AMD, we use the performance counters shown in Table 4 to analyze the behavior of the software prefetch instruction.

B Retirement Experiment

In this section, we discuss the measurement code of the retirement experiment discussed in Section 3.3. To obtain a timestamp, we either use `rdpru` or `rdtscp`. As discussed in Section 3.1, we need to add additional fences if `rdpru` is used. We note that we used `lfence` instructions as `mfence`, contrary to its documentation [33], seems to be ordered with respect to prefetch instructions. Figure 12 shows the measurement code for one of the measured cases. We refer to our code in our GitHub repository[2] for the full implementation. The obtained results, as shown in Table 5, indicate that loads triggered by software prefetch instructions are immediately marked as complete after the address translation. Similar timing differences are visible for other prefetch instructions such as `prefetcht0`.

```
for (size_t i = 0; i < 1000000; i++) {
  flush(address);
  lfence();
  start = get_timestamp();

  prefetch(address);
  lfence();

  measurements[i] = get_timestamp() - start;
}
```

Figure 12: Retirement Experiment

To rule out the possibility that the `prefetch` instruction is not only serialized with respect to `cpuid`, we designed the

[2] https://github.com/amdprefetch/amd-prefetch-attacks

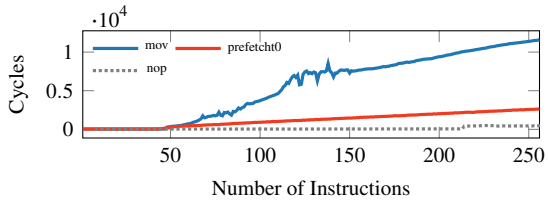

Figure 13: Execution time in cycles of executing `prefetcht0` instructions, memory loads and `nop`s.

Table 5: Measured execution times for memory loads and prefetches.

Instructions	Runtime in cycles
`movq` + `cpuid`	945.71 (n = 1 000 000, $\sigma_{\bar{x}}$ = 229.42)
`movq` + `lfence`	814.44 (n = 1 000 000, $\sigma_{\bar{x}}$ = 241.46)
`prefetchnta` + `cpuid`	842.96 (n = 1 000 000, $\sigma_{\bar{x}}$ = 249.07)
`prefetchnta` + `lfence`	158.58 (n = 1 000 000, $\sigma_{\bar{x}}$ = 124.12)

following additional experiment. Using AssemblyLine [66], we generate code within a single run of our program that performs a different number of memory loads, respectively, a different number of `prefetch` instructions and `nop`s followed by a `cpuid` instruction. Before we emit the first load or prefetch instruction, we issue a `movntdqa` instruction that takes a long time to complete and thus, blocks all subsequently issued instructions from retiring in the reorder buffer.

We record the measured execution time of each run on an AMD Ryzen 5 3600 (Zen 2) CPU and present the average execution time of the performed measurements in Figure 13.

One can clearly see the increase in the execution time of the load instructions starting when the 44 entries of the load queue are exhausted. However, for `prefetch` instructions, the execution time increases linearly, hinting that the `prefetch` instruction does not cause a stall and is immediately marked as completed. This also confirms the description by AMD that "*[...] a load instruction may cause a subsequent instruction to stall until the load completes, but a prefetch instruction will never cause such a stall.*" [3, §3.9.6.1]. For the inserted `nop` instructions, we observe an increase in the execution time after 224 instructions corresponding to the number of entries in the reorder buffer.

C Measured Instruction Sequences

Listing 1 shows the instruction sequences that have been measured for Figure 1. All instructions are executed in an endless loop on one CPU core and measured by a different core using the RAPL interface. Registers are initialized to zero if not specified otherwise. Note that this specific `xor` target leads to a read-after-write hazard, which results in measuring only a single `xor` and not multiple parallel `xor`s [9].

```
1  void target_flush() {
2          asm volatile(
3          "1:\n"
4          "clflush 0(%0)\n"
5          "jmp 1b\n" : : "c"(dst) : "memory");
6  }
7  void target_rdtsc() {
8          asm volatile(
9          "1:\n"
10         "rdtsc\n"
11         "jmp 1b\n" : : : "rax","rdx","memory");
12 }
13 void target_xor() {
14         asm volatile(
15         "1:\n"
16         "xor %%rbx, %%rax\n"
17         "jmp 1b\n" : : : "rbx","rax","memory");
18 }
19 void target_load() {
20         asm volatile(
21         "1:\n"
22         "movq (%0), %%rax\n"
23         "jmp 1b\n" : : "rm"(dst) : "rax","memory");
24 }
```

Listing 1: The measured instruction sequences for Figure 1.