KASLR: Break It, Fix It, Repeat

Claudio Canella
Graz University of Technology

Michael Schwarz
Graz University of Technology

Martin Haubenwallner
Graz University of Technology

Martin Schwarzl
Graz University of Technology

Daniel Gruss
Graz University of Technology

ABSTRACT

In this paper, we analyze the hardware-based Meltdown mitigations in recent Intel microarchitectures, revealing that illegally accessed data is only zeroed out. Hence, while non-present loads stall the CPU, illegal loads are still executed. We present EchoLoad, a novel technique to distinguish load stalls from transiently executed loads. EchoLoad allows detecting physically-backed addresses from unprivileged applications, breaking KASLR in 40 µs on the newest Meltdown- and MDS-resistant Cascade Lake microarchitecture. As EchoLoad only relies on memory loads, it runs in highly-restricted environments, e.g., SGX or JavaScript, making it the first JavaScript-based KASLR break. Based on EchoLoad, we demonstrate the first proof-of-concept Meltdown attack from JavaScript on systems that are still broadly not patched against Meltdown, *i.e.*, 32-bit x86 OSs.

We propose FLARE, a generic mitigation against known microarchitectural KASLR breaks with negligible overhead. By mapping unused kernel addresses to a reserved page and mirroring neighboring permission bits, we make used and unused kernel memory indistinguishable, *i.e.*, a uniform behavior across the entire kernel address space, mitigating the root cause behind microarchitectural KASLR breaks. With incomplete hardware mitigations, we propose to deploy FLARE even on recent CPUs.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

meltdown; side-channel attack; transient execution; kaslr; countermeasure; reverse engineering

ACM Reference Format:
Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3320269.3384747

1 INTRODUCTION

CPUs are optimized for performance and efficiency. Some optimizations are exposed to the user via the instruction-set architecture (ISA), the hardware-software interface, but most are transparent to the developer. CPU vendors can implement any optimizations while still adhering to the ISA, without taking security into account.

As a consequence, many attacks on the microarchitectural CPU state have been published [25]. Most of these are side-channel attacks, including attacks on cryptographic algorithms [4, 45, 46, 60, 69, 71, 96], on user interactions [34, 58, 78], but also covert data transmission [60, 62, 94, 95]. Meltdown [59], Foreshadow [87, 93], RIDL [89], ZombieLoad [79], and Fallout [9] are recent microarchitectural attacks that go beyond side-channel attacks and directly leak (arbitrary) data instead of metadata. These attacks, commonly referred to as Meltdown-type transient-execution attacks [10], exploit the lazy fault handling property of some CPUs. With lazy fault handling, the CPU continues using data in the out-of-order execution even if the loading of the data resulted in a fault, e.g., failed the privilege-level check. While the data never becomes visible on the architectural level, it is encoded in the microarchitectural state, *i.e.*, in the cache. From there, it is made visible on the architectural level using microarchitectural side-channel attacks.

Meltdown-type transient-execution attacks [10] break the hardware enforced-isolation between the trusted kernel and untrusted user programs. These attacks, which are present in most Intel CPUs, showed that it is not always possible to protect an application against side-channel attacks. This is contrary to the belief that side-channel attacks have to be prevented by the application itself [5, 46].

Deeply rooted in the CPU, either close to or in the critical path, most transient-execution attacks cannot be fixed with microcode updates. However, on recent CPUs, Intel introduced hardware mitigations for the first Meltdown-type attacks [17, 39, 40]. On the latest microarchitecture (Cascade Lake), all known Meltdown-type attacks are mitigated in hardware [41]. However, due to their severity and the ease of exploitation, operating systems (OSs) rolled out software-based mitigations to prevent Meltdown [30]. These software mitigations introduce a stricter separation of user and kernel space [31, 32]. This stricter separation does not only prevent Meltdown but also prevents other microarchitectural attacks on the kernel [31], e.g., KASLR (kernel address space layout randomization) breaks [32, 37, 49] which allow an attacker to de-randomize the location of the kernel in the address space. As a drawback, these software mitigations may introduce significant performance overhead. This is especially true for workloads that need frequent switching between user and kernel space [30]. Consequently, hardware manufacturers solved the underlying root cause directly in recent CPUs, making the software mitigations obsolete.

Even though new CPUs are not vulnerable anymore to the original Meltdown attack, we show that they still show signs of Meltdown-type effects. In this paper, we investigate CPUs that have hardware-based Meltdown mitigations. We analyze these fixes and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6750-9/20/10…$15.00
https://doi.org/10.1145/3320269.3384747

develop the hypothesis that they only prevent the data from being used in subsequent operations, not the actual load. We confirm this hypothesis by showing that the fixes introduce new side effects, namely that on illegal accesses to kernel addresses, the CPU zeroes out the data but still performs the load. In contrast, loads from non-present pages stall the CPU. We present a method based on Flush+Reload [96] to distinguish the stalling behavior of loads. With this method, we can exploit the side effects of the Meltdown mitigations to break KASLR reliably. By probing the kernel space for load stalls, we detect whether the probed virtual address is physically backed, revealing the location of the kernel. We demonstrate that these effects can also be exploited on older CPUs, which are affected by Meltdown but protected by software mitigations.

Our KASLR break, EchoLoad, works on all major OSs (Linux, Windows, macOS, and Android x86_64). We tested the KASLR break on Intel microarchitectures from Arrandale (2010) to Cascade Lake (2019) on Atom, Core, and Xeon CPUs. Even on Cascade Lake with fixes for Meltdown and MDS [41], we de-randomize the kernel in 40 µs (F-score 1, $n = 10^9$). Hence, our KASLR break is the fastest and most reliable one published. Moreover, EchoLoad is the only KASLR break that only relies on memory loads and works on Intel microarchitectures since at least 2010. EchoLoad even works on KPTI, the Linux software mitigation for Meltdown.

As EchoLoad does not require anything but memory loads, it works in restricted environments such as SGX and JavaScript. We highlight that EchoLoad can aid kernel exploitation from within SGX enclaves, facilitating SGX malware [81, 82]. In contrast to previous ASLR breaks from JavaScript [7, 29, 76], we are the first to demonstrate a microarchitectural KASLR break from JavaScript on x86 OSs. We also show that on older unpatched x86 OSs, Meltdown can even be exploited from JavaScript. This is particularly dangerous for any Windows XP machines (1–3% of Desktop computers [68]), for which no software patches are available, but which are still running in official, commercial, industrial, or personal environments. Our attack will also soon be possible on unprotected 64-bit systems as WebAssembly plans to extend the size of linear memory indices to 64 bit [91]. We pinpoint the remaining challenges for widely deployable JavaScript-based Meltdown exploits.

To mitigate all microarchitectural attacks on KASLR, including EchoLoad, we present FLARE (Fake Load Address REsponse). The basic idea is to back the entire kernel address space with physical pages. FLARE prevents previous attacks [9, 32, 37, 49, 76] by hiding the kernel within a virtual-address range appearing to be valid. Our proof-of-concept implementation has a memory overhead of only 12 kB, and no measurable runtime overhead.

FLARE is compatible with KPTI on Meltdown-affected CPUs, and forms a low-cost mitigation on CPUs with hardware fixes. We evaluated our open-source proof-of-concept implementation of FLARE[1] for Linux for both cases. Our results show that FLARE indeed prevents all known microarchitectural attacks on KASLR.

We conclude that while the hardware mitigation for Meltdown fixes the problem of Meltdown-US [10], they introduce a new side effect by merely zeroing out data that is illegally accessed, enabling EchoLoad. Based on our analysis of the behavior of AMD and ARM CPUs, we believe that the only complete solution for Meltdown is to treat inaccessible pages the same way as unmapped pages. Furthermore, the software-based isolation of user and kernel space [31] is not sufficient, and we thus suggest to deploy FLARE to prevent microarchitectural attacks on the kernel.

Contributions. The contributions of this work are:

(1) We analyze Meltdown hardware fixes on Intel CPUs and discover a Meltdown-related effect on Meltdown-fixed Intel CPUs.
(2) We present KASLR and ASLR breaks, even from SGX and including the first KASLR break from JavaScript.
(3) We show a JavaScript Meltdown attack on 32-bit x86.
(4) We propose FLARE, a mitigation preventing currently known microarchitectural attacks on KASLR with negligible overhead.

Outline. Section 2 provides background on ASLR and attacks on ASLR. We analyze Meltdown hardware fixes in Section 3. Section 4 presents our new mitigation for microarchitectural KASLR breaks. Section 5 evaluates FLARE's performance and efficacy against attacks. Section 6 discusses related work. Section 7 concludes.

Responsible Disclosure. We responsibly disclosed our findings to Intel on August 5, 2019, and Intel acknowledged them.

2 BACKGROUND

In this section, we provide the background on caches, transient execution and transient-execution attacks, virtual memory, Intel SGX, and address space layout randomization (ASLR).

2.1 Cache Attacks

Caches were designed to hide the latency of memory accesses, creating a timing side channel. Over the past two decades, many different attack techniques have been proposed [5, 33, 54, 69, 96]. Two of these attacks are Prime+Probe [69, 71] and Flush+Reload [96]. In a Prime+Probe attack, an attacker constantly measures how long it takes to fill a cache set with the same set of data. Whenever a victim accesses a cache line mapping to the same cache set, the attacker will measure a higher runtime for the filling. In a Flush+Reload attack, an attacker constantly flushes a cache line and reloads the data. By measuring how long the reload takes, the attacker can infer whether a victim has accessed the data in the meantime. As Flush+Reload exhibits low noise and has a fine granularity, it has been used for attacks on user input [34, 58, 78], cryptographic algorithms [4, 46, 96], and web server function calls [97].

Side channels can also be used to build covert channels. In a covert channel, the attacker controls both the sender and receiver. The goal is then to leak information from one security domain to another, bypassing isolation on both the functional and system level. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [33, 60, 62].

2.2 Transient-execution Attacks

Another optimization is out-of-order execution, avoiding CPU stalls when in-order instructions wait for operands. Instructions are decoded into micro-operations (μOPs) [22] and placed in the Re-Order Buffer (ROB), along with their operands. While waiting for operands, μOPs whose operands are already available are scheduled in the meantime. Results of the out-of-order executed instructions are stored until they can be retired.

[1] https://github.com/IAIK/FLARE

Modern software is rarely linear but contains branches. To avoid pipeline stalls upon unresolved branch conditions, modern CPUs implement speculative execution, predicting the most likely outcome of the branch and starting execution along the predicted path. The results are again placed in the ROB until retirement, *i.e.*, the prediction has been verified. If the prediction was correct, a significant speedup is achieved. Otherwise, the CPU has to revert all results and needs to flush the pipeline and the ROB. Unfortunately, microarchitectural state changes, such as loading data into the cache or TLB, are not reverted. This allows an attacker to use microarchitectural covert channels to exfiltrate the secret data. Speculative or out-of-order executed instructions that were never committed to the architectural state are also referred to as *transient instructions* [10, 53, 59]. Spectre-type attacks exploit transient execution before a misprediction is discovered [10, 36, 51, 53, 55, 61]. Meltdown-type attacks exploit transient execution before a fault or interrupt is handled [3, 9, 10, 39, 40, 51, 59, 79, 85, 87, 89, 93].

Meltdown. Meltdown exploited lazy exception handling in modern CPUs [59]. The attacker triggers a page fault but suppresses it via fault handling, TSX transactions, or misspeculation. While the CPU knows that the access is not allowed, the exception is only raised at the retirement stage. Hence, dependent instructions receive the data and can then, e.g., encode the value in the cache which the attacker can leak using a technique like Flush+Reload.

2.3 Intel SGX

In recent years, software vendors discovered that specific security properties, e.g., for DRM, in theory, are much easier to achieve with trusted-execution mechanisms. Consequently, hardware vendors reacted and developed different trusted-execution environments [1, 42, 50]. Intel developed an instruction-set extension called Software Guard Extension (SGX) [42]. With SGX, an application is split into a trusted and an untrusted part. To protect the former, it is executed within a hardware-backed enclave. In the SGX threat model, neither the OS nor any other application is trusted. Therefore, the CPU guarantees that any memory belonging to the enclave cannot be accessed by anyone else than the enclave. The SGX threat model also allows the remaining hardware to be malicious or compromised. Consequently, the SGX memory is encrypted, protecting it from being directly read from the DRAM module. Additional threats like memory-safety violations [56], side channels [8, 82], or race conditions [77, 92] are considered out of scope and remain an enclave developer's responsibility.

The SGX interface to let the untrusted part enter an enclave conceptually resembles system calls. Once the trusted execution is finished, the result of its computation as well as the control flow is returned to the callee. However, SGX protection mechanisms are one-sided: SGX allows data sharing between the trusted and the untrusted part by giving enclaves full access to the entire host application's address space. Recently, it has been shown that this asymmetric protection gives rise to enclave malware [81].

2.4 Address Translation

Modern OSs rely on memory isolation for security purposes. Hence, CPUs support virtual memory for abstraction and memory isolation. Processes work on virtual addresses and cannot architecturally interfere with each other as the virtual address spaces are non-overlapping and overlapping areas are protected according to the processes' requirements. These virtual addresses have to be translated to physical addresses using multi-level page tables. A dedicated translation-table register indicates the location of the first-level table, e.g., CR3 on Intel architectures. Upon a context switch, the OS updates the translation-table register with the physical address of the first-level page table of the process scheduled next. Page-table entries do not only provide translations but also define properties of memory regions, e.g., executable or not.

2.5 Address Space Layout Randomization

Since the introduction of non-executable (NX) bits, memory corruption attacks have to rely on existing code in the victim process instead of code injection [86]. Shacham et al. [83] generalized the concept of code-reuse attacks, which is now widely known as return-oriented programming (ROP). Subsequently, a variety of code-reuse attack techniques have been described [6, 12, 13, 28, 75].

Code-reuse attacks require knowing addresses of specific code snippets. Similarly, data-only attacks [11, 47] require knowledge of addresses, e.g., of specific data structures. Over the years, many different mitigation techniques have been developed [86], e.g., NX stacks, stack canaries, and ASLR. The idea behind ASLR is to make the addresses of code and data unknown to an attacker by randomizing them. Typically, ASLR randomizes the base address of the executable, stack, heap, and shared libraries. Hence, even if an attacker hijacks the control flow, it is significantly harder to exploit bugs in an application as the location of code snippets usable for code-reuse attacks is unknown. By brute-forcing the location, the chances are high that the process will crash, and any ongoing attack is unsuccessful. Furthermore, an application is re-randomized on every startup, reducing the chances of a successful attack.

General Idea of KASLR. While ASLR initially only protected user-space applications, the kernel space was later on also protected by KASLR [20, 49], e.g., introduced in Windows in 2007 [49], macOS in 2012 [2], and Linux in 2014 [20]. The kernel consists of multiple segments that are individually mapped into the kernel address space. These segments include the code (*i.e.*, text segment), drivers or modules, and data (e.g., stack, heap). The KASLR implementations of the three major OSs (Linux, Windows, macOS) only use coarse-grained randomization, *i.e.*, randomized base address. Fine-grained KASLR implementations using code diversification have been proposed [27, 72] but are not used in practice.

Another property of KASLR implementations is that the kernel is mapped using either 4 kB or 2 MB pages. The mapping is 2 MB-aligned [76], reducing the number of possible offsets. Moreover, the order of the randomized segments is not changed, e.g., in Linux, the text segment always has a lower address than the modules [57]. Consequently, KASLR provides a lower entropy than typical user-space ASLR implementations [20]. However, if an exploit attempt fails, it likely crashes the kernel. Hence, an attacker only has one

shot, and exploitation techniques relying on a large number of retries cannot be used against the kernel if KASLR is active.

Linux. In Linux 5.x, most sections are independently randomized at boot, including the direct-physical map, vmalloc and ioremap space (vmalloc area), virtual-memory map (vmemmap), text segment, and modules [24]. The text segment is mapped between `0xffff ffff 8000 0000` and `0xffff ffff c000 0000` with a maximum size of 1 GB [76]. As the kernel has to be aligned to a 2 MB boundary, the randomization has 9 bits of entropy. Therefore, the kernel is placed at one of 512 possible offsets. Modules are mapped using 4 kB pages in a 1 GB range following the text segment. Unmapped pages follow each module before a new module starts [49].

Start and end addresses for the direct-physical map, the vmalloc area, and the vmemmap are documented [57], but analyzing the start addresses on repeated restarts shows that they are only correct if KASLR is disabled. Therefore, we analyzed the KASLR implementation of Linux kernel version 5.2.9. This analysis showed that the possible start address is indeed `0xffff 8880 0000 0000` for the direct-physical map. It is then placed at a random offset from the start address, aligned to a 1 GB boundary. The vmalloc space is placed at a random offset relative to the end of the direct-physical map with at least 1 GB between them. The vmemmap area is then randomized starting from the end of the vmalloc area, again with at least 1 GB between them. The range of possible addresses is, therefore, from `0xffff 8880 0000 0000` to `0xffff fdff ffff ffff`, always with a 1 GB alignment and the preserved order.

Windows. Windows randomizes almost everything except the HAL heap once at boot [44]. Windows first introduced KASLR with Vista [49] and improved it over time [32]. Windows 7 maps the kernel, followed by the drivers in the same range with the same randomization. The address range of the kernel and drivers is `0xffff f800 0000 0000` to `0xffff f803 ffff ffff` [49]. KASLR on Windows 10 differs from Windows 7 as there is a separate area for the kernel and drivers. The kernel is still mapped in the same virtual address range, but drivers are now mapped in the range of `0xffff f800 0000 0000` to `0xffff f80f ffff ffff` [23]. The kernel is also 2 MB-aligned, resulting in 8192 possible offsets. Drivers are mapped with 4 kB pages with a 16 kB alignment.

macOS. Starting with macOS 10.8 (Mountain Lion), the kernel, `kexts` (kernel modules), and zones are randomized [70]. For instance, the kernel is mapped in the range of `0xffff ff80 0000 0000` to `0xffff ff80 2000 0000` with a 2 MB alignment, resulting in 256 possible offsets. The offset at which the kernel is placed relative to the start of the address range is called *kslide*. According to Chen and He [14], kernel and `kexts` share the same `kslide`.

3 A NOVEL (K)ASLR BREAK

In this section, we first analyze the Meltdown hardware mitigation on new Intel CPUs. We then introduce EchoLoad, an attack primitive that exploits incomplete Meltdown countermeasures to break KASLR. We detail how we can use it to break KASLR from an unprivileged user-space application, JavaScript, and SGX.

3.1 Analyzing the Meltdown Mitigation

The Meltdown vulnerability allowed unprivileged users to leak kernel memory (cf. Section 2.2). The immediate workaround was KAISER [31], a software-only solution to unmap the kernel when running in user space. With the Whiskey Lake microarchitecture, Intel fixed the vulnerability in hardware without providing further details on how their fix works. CPUs with the hardware mitigation indicate that they are not vulnerable by having the `RDCL_NO` bit set in the `IA32_ARCH_CAPABILITIES` model-specific register [42].

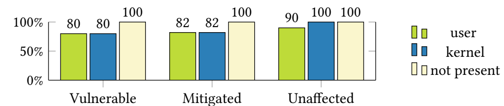

Figure 1: Loads from non-present pages always stall, loads to kernel addresses stall on unaffected AMD CPUs.

Lipp et al. [59] argued that stalling the CPU until the permission check is done might be too costly. We suspect that such a change also requires redesigning a significant part of the CPU's pipeline. As the first CPUs with hardware mitigations already shipped approximately one and a half years after Meltdown was disclosed to Intel, we expect only minor hardware changes as mitigation.

Hypothesis. We hypothesize that instead of stalling on an illegal memory load, the CPU zeroes out the result. Hence, the CPU still loads inaccessible memory locations, but instead of providing the real value to dependent instructions, it always provides '0'.

Verification. We get the first indication that our hypothesis is correct by simply mounting a Meltdown attack. When running the Meltdown attack on a Xeon Silver 4208 CPU which has the `RDCL_NO` bit set, we always get '0's. To verify our hypothesis, we further analyzed performance counters on three different systems: a Meltdown-vulnerable Intel CPU (i7-8650U), an Intel CPU with hardware mitigations (Xeon Silver 4208), and a non-affected AMD CPU (Ryzen Threadripper 1920X). For all systems, we evaluate performance counters when executing the following code 10^4 times: `if (transient_begin()) { *(volatile char*)0; oracle[*address]; }`. The function `transient_begin` either starts a TSX transaction if available, or sets up a signal handler for segmentation faults [59]. The null-pointer access is required to always cause an exception.

The first performance counter of interest is the number of CPU stalls when executing the above code. On Intel CPUs, we use `CYCLE_ACTIVITY.STALLS_MEM_ANY`, and on AMD CPUs the "Dispatch Stalls" counter. We set `address` to a valid kernel address. As baselines, we choose a mapped user address as well as a non-present address for `address`. Figure 1 shows the results of the performance counters for all 3 systems. For comparison, we normalized the values such that the highest value on each system represents 100 %.

All CPUs stall when accessing a non-present virtual address. The AMD CPU also stalls when accessing a kernel address. Both Intel CPUs with and without mitigations show the same stall behavior. Hence, even the Intel CPUs with Meltdown mitigations do not stall when accessing a kernel address. This indicates that the memory load for the kernel address is actually issued.

We substantiate this observation by analyzing another performance counter. With the counters `UOPS_DISPATCHED_PORT.PORT_2` and `UOPS_DISPATCHED_PORT.PORT_3`, we can track the number of μOPs issued on the load ports. The sum of these two counters is the number of all memory loads. Figure 2 shows the number of memory loads when running the code mentioned above with a

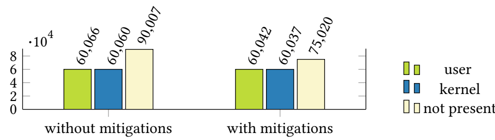

Figure 2: Issued load μOPs for user and kernel addresses (Intel). Only invalid loads from non-present pages are reissued.

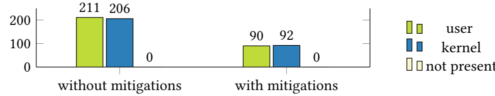

Figure 3: Number of cycles L1D cache misses are pending. User and kernel addresses reach the memory hierarchy, non-present pages do not.

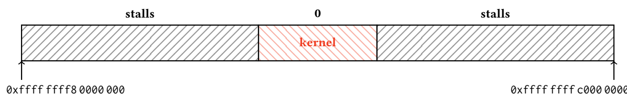

Figure 4: Reading addresses not physically backed stalls the CPU, while kernel addresses return '0' (or the actual data).

user-space, kernel-space, and non-present address both on an Intel CPU with and without hardware mitigation. When trying to load from a non-present page, the load faults and the load instruction is re-issued [79]. The number of issued loads for kernel addresses is the same as for user-space addresses on both CPUs. This indicates that these loads succeed and do not have to be re-issued.

Finally, we show that the issued loads for kernel addresses indeed load data from the memory hierarchy, and not, e.g., from an internal buffer containing '0'. Thus, we monitor the number of cycles that L1 data-cache misses are waiting to be retrieved. Figure 3 shows the values of the performance counter `L1D_PEND_MISS.PENDING_CYCLES` for the previously shown code. While non-present pages do not cause an L1 miss, both user-space and kernel addresses cause L1 misses. This is even the case for CPUs with hardware mitigations against Meltdown, showing that loads to kernel addresses retrieve the actual value, and only later on zero it out.

3.2 Breaking KASLR

EchoLoad is a new microarchitectural KASLR attack exploiting Meltdown-related side effects. EchoLoad reliably breaks KASLR, regardless of OS, software mitigations, and microcode updates. EchoLoad works on all Intel CPUs since 2010, even if they are not affected by Meltdown, e.g., CPUs with the RDCL_NO bit. In contrast to the KASLR break by Schwarz et al. [76], EchoLoad also works on the new Cascade Lake, which is not affected by Meltdown or MDS.

General Idea. The general idea is to distinguish whether accessing a kernel address in the transient-execution domain leads to a stall. We exploit the fact that instructions can only be executed out of order if their data dependencies are fulfilled. Hence, we dereference a user-space memory location where the address is computed based on the value of the kernel address that is being tested.

Listing 1 shows this central part of EchoLoad. First, an attacker induces transient execution by provoking a fault or a misspeculation in Line 1. If the access to `address` in Line 2 stalls, the user

```
1 if (transient_begin()) {
2   *(volatile char*)(mem + *address);
3 }
4 if (flush_reload(mem)) return ADDRESS_MAPPED;
5 else return ADDRESS_NOT_MAPPED;
```

Listing 1: The main part of EchoLoad. The address `mem` is only cached if the access to `address` does not stall.

address cannot be computed before the transient execution aborts. Otherwise, the user address is dereferenced and, thus, cached before the transient execution aborts. After the transient execution, the attacker probes the user- address in Line 4, e.g., using Flush+Reload. If the user address is cached, `address` is valid, *i.e.*, physically backed. Otherwise, `address` is not valid, *i.e.*, not physically backed. Figure 4 illustrates the general idea of EchoLoad.

CPUs with Meltdown Fixes. To break KASLR on CPUs with Meltdown fixes, we run EchoLoad on all 512 possible kernel offsets (cf. Section 2.5). Only where a physical page backs the tested address, we read 0, on other addresses the CPU stalls.

As the CPU stalls on all reads from addresses that the kernel is not mapped to, we observe no false positives. This makes EchoLoad a very reliable attack that even works on Cascade Lake CPUs.

CPUs without Meltdown Fixes. On CPUs without Meltdown fixes, we cannot rely on the CPU returning 0 for reads on kernel pages. Instead, if KPTI is disabled, we read the actual content of the page. As the content of the page is code, there are 256 possible addresses which could be dereferenced.

As the 256 possible addresses are contiguous, and the cache line size is typically 64 byte, they fall into one out of 4 possible cache lines. Testing 4 adjacent cache lines with Flush+Reload triggers the stride prefetcher [38] on Intel CPUs. Instead, we can exploit the L2 adjacent cache line prefetcher (spatial prefetcher) [38], which fetches the sibling cache line whenever a cache miss is handled. Hence, we only have to check 2 cache lines using Flush+Reload, which works without triggering the stride prefetcher. Consider a case with 4 adjacent cache lines. If the data we read falls into line 0 and we check line 1, we observe a hit on line 1 because the prefetcher also loads it into the cache. The same is true if the data falls into line 3, and we check line 4. By merely checking cache lines 1 and 3, we detect all possible accesses.

We can even further increase the performance by only checking one cache line. By using a kernel module, we investigated the beginning of the kernel text segment and determined that it is always the same across kernel versions (*i.e.*, `0x48`).

EchoLoad also works with KPTI [30] as the pages still mapped with KPTI use the same randomization offset as the rest of the kernel code. While the value differs with KPTI (*i.e.*, `0xf`), it is still the same across kernel versions that use it. As we only look for the beginning of the kernel and we know that the value remains constant, we can reduce the number of cache lines we need to check to 1. This further improves the performance of our KASLR break.

EchoLoad and LVI-NULL. On CPUs that have already received fixes for Meltdown, EchoLoad is the inverse of the LVI-NULL attack [88]. While LVI-NULL abuses the fixes to inject a dummy value of zero to dependent transient instructions in a victim, EchoLoad

Table 1: Environments where we evaluated EchoLoad and Data Bounce (KPTI disabled).

CPU	μarch.	EchoLoad	Data Bounce
Intel Atom x5-Z8300	Cherry Trail	✓	✓
Intel Core i5-450M	Arrandale	✓	✓
Intel Core i5-3230M	Ivy Bridge	✓	✓
Intel Core i5-8250U	Kaby Lake R	✓	✓
Intel Core i7-4790	Haswell	✓	✓
Intel Core i7-6700K	Skylake	✓	✓
Intel Core i7-8650U	Kaby Lake R	✓	✓
Intel Core i7-8565U	Whiskey Lake	✓	✓
Intel Core i9-9900K	Coffee Lake	✓	✓
Intel Xeon E5-1630 v4	Broadwell	✓	✓
Intel Xeon Silver 4208	Cascade Lake	✓	✗
Intel Cascade Lake (Google Cloud)	Cascade Lake	✓	✗
AMD Ryzen Threadripper 1920X	Zen	✗	✗
AMD Ryzen 7 3700	Zen 2	✗	✗
ARM Cortex-A57	A57	✗	✗

Table 2: Performance of EchoLoad in terms of runtime and F-score. Each possible offset is tested a single time.

CPU		Speculation	TSX	Segfault
i7-6700K	Time (F-Score)	63 μs (0.999)	48 μs (1.000)	133 μs (1.000)
i9-9900K	Time (F-Score)	33 μs (1.000)	29 μs (1.000)	86 μs (1.000)
Xeon Silver 4208	Time (F-Score)	51 μs (0.994)	40 μs (1.000)	127 μs (1.000)

exploits the inverse effect, *i.e.*, the retrieving of a zero value, to break KASLR.

Evaluation. We evaluated EchoLoad on different Intel microarchitectures running Linux (cf. Table 1). On all CPUs, we evaluated our attack with both KPTI enabled and disabled. These experiments show that KPTI does not prevent EchoLoad. If KPTI is disabled, EchoLoad detects the symbol `startup_64`. With KPTI, it detects the symbol `__entry_text_start`, which is the trampoline required to enter the kernel. As Android is based on the Linux kernel, the behavior on Android is the same. While we evaluate the ability and performance of EchoLoad to leak the kernel code offset, it can equally leak the offsets of all other randomized parts of the kernel.

For the performance evaluation, we used the same setup that Schwarz et al. [76] describe in their paper. We tested 10 different randomizations (*i.e.*, 10 reboots), each 100 times. Using this approach, we have a sample size of 10^3. We evaluated the performance on a selected number of architectures in all three cases, namely misspeculation, TSX, and segfault handling. Table 2 shows the result of this evaluation. In all tested cases, we achieve almost perfect accuracy. On the i9-9900K, we outperform Data Bounce in terms of time required while matching the accuracy.

Similar to Schwarz et al. [76], we tested EchoLoad with TSX on a larger scale. For that, we tested the same offset 100 million times and repeated the experiment 10 times for a total of 1 billion tries. On all three CPUs (cf. Table 2, we achieved an average F-score of 1, giving us perfect accuracy in detecting the KASLR offset.

On Windows 10, we also tested 10 different randomizations (*i.e.*, 10 reboots), each 100 times. In all cases, we successfully found the location of the kernel image. On macOS 10.11.6, instead of 10 randomizations, we repeated the experiment 100 times to verify that the given kernel range is still correct [14]. We then successfully recovered the kernel location in all 100 randomizations.

```
1 if(xbegin() == (~0u)) { *(volatile char*)mem; xend(); }
2 if(flush_reload(mem)) return ROLL_BACK;
3 else return IMMEDIATE_ABORT;
```

Listing 2: Analyzing the behavior of the TSX abort. If the transaction is aborted on `xbegin`, `mem` cannot be cached. If the transaction is just rolled back on `xend`, `mem` is cached.

3.3 Breaking (K)ASLR from SGX

As EchoLoad only requires memory accesses, it also works in restricted environments. We demonstrate EchoLoad in SGX enclaves breaking host ASLR, victim-enclave ASLR, and KASLR.

While it is also possible to use EchoLoad for detecting the location of SGX enclaves from the host application, this is an artificial scenario. First, the host maps the enclave to its location and, thus, knows where the enclave is. Second, on Linux, the host can access this information from the pseudo file `/proc/self/maps`, containing all virtual-address mappings of the current process. Finally, the host can also probe the virtual memory for the enclave, e.g., using a signal handler to catch segmentation faults. If a region returns `0xff`, it is likely to be an EPC page of an enclave.

EchoLoad from Enclave to Host. *TAP* is a method to break host ASLR from an enclave using Intel TSX [81]. It allows scanning the host address space for mapped pages to mount a ROP attack from inside the enclave, impersonating the host application.

While *TAP* only worked for CPUs with TSX, it does not work on CPUs with MDS fixes in microcode at all. With the microcode update, all TSX transactions abort immediately when started inside an SGX enclave [43]. We further analyzed whether the transaction aborts immediately, or is only rolled back in all cases.

Listing 2 shows the code we use to analyze the TSX-transaction aborts. If the transaction aborts already at the `xbegin` instruction, the memory dereference is never executed. If the transaction executes but then rolls back the executed instructions, the dereference of the address still causes the memory location to be in the cache.

Our results show that the transaction is never started as the address `mem` is never cached after the transaction. Hence, we cannot even use TSX to access memory locations transiently. We observe the same behavior outside an SGX enclave when setting the `TSX_FORCE_ABORT` MSR to 1. While this MSR is documented to abort every TSX transaction on *commit* [98], we verified with our test (Listing 2) that the transaction is not even started.

Consequently, even if TSX is re-enabled in SGX via a microcode update, it can be manually disabled with the `TSX_FORCE_ABORT` MSR to protect against attacks such as *TAP*. This is the case on the Amazon EC2 cloud [79]. In contrast to Data Bounce [76], EchoLoad works on the newest CPU generation, as it does not require TSX. Thus, EchoLoad can be used to mount SGX ROP attacks [81] even if TSX is disabled, once more enabling such attacks.

Due to the unavailability of syscalls and the `rdtsc` instruction inside SGX, we mount EchoLoad behind a misspeculated branch and use a counting thread [82] as a timer. We achieve a speed of 388 Mbit/s for scanning the host address space with EchoLoad. Hence, EchoLoad is a viable alternative to *TAP* to de-randomize the host application from an SGX enclave.

EchoLoad from Enclave to Enclave. Enclaves might not only want to de-randomize the host application but also learn information about other enclaves. While enclaves are mutually untrusted and, thus, cannot access each other, EchoLoad can be used to learn the address-space layout of other enclaves. Moreover, assuming that enclaves have unique sizes, an enclave can even detect which other enclaves are used by the host by detecting their size.

We evaluated EchoLoad in the cross-enclave scenario by loading two enclaves in our test application. One enclave is malicious and leverages EchoLoad to learn which other enclaves are used by the host application. We use the same experiment as for de-randomizing the host to scan the address space for other enclaves. We successfully detect the location and the size of the second enclave used by the host application. The speed for scanning the address space is the same as for de-randomizing the host application.

EchoLoad from Enclave to Kernel. Enclaves may foster stealthy exploits [48, 65, 81, 82]. In this work, we add another primitive to malware hidden inside SGX. With EchoLoad, an enclave can de-randomize KASLR, which is a prerequisite for many kernel exploits.

The same code which is used to de-randomize the host application can be used to de-randomize the kernel. We evaluated EchoLoad inside an SGX enclave to find the KASLR offset. Due to the use of misspeculation and a timing thread, the performance is worse than in native code. However, we still detect the KASLR offset with an F-score of 1 ($n = 10^3$).

3.4 Meltdown and KASLR Break in JavaScript

EchoLoad can even be mounted from a JavaScript sandbox. We demonstrate EchoLoad, and as an extension Meltdown, from the Spidermonkey JavaScript engine 60.1.3 used in Firefox.

There are two challenges for mounting EchoLoad in JavaScript. First, both JavaScript and WebAssembly currently only support a 32-bit linear memory index, restricting arrays to 4 GB [19]. While this prevents EchoLoad on a 64-bit OS, it does not prevent it on 32-bit OSs, which only support a 32-bit virtual address space. Hence, we evaluate this attack on Ubuntu 16.04 (Kernel 4.15.0-60) i686 on an Intel i7-4790. While we are currently limited to 32-bit systems, the WebAssembly developers are planning to increase the size of linear memory indices from 32-bit to 64-bit, allowing the attack on all commodity systems that are not patched against Meltdown [91]. Second, the Spectre mitigations do not only reduce the resolution of the high-resolution timer [67], but also harden the bounds check for arrays, preventing speculative out-of-bounds accesses by default [66]. As our focus is not demonstrating a Spectre attack but a Meltdown-related effect, we use a version of the engine that allows speculative out-of-bounds accesses, as in previous work [53]. To develop widely deployable Meltdown and EchoLoad exploits, further research is necessary to investigate whether other misprediction mechanisms may provide a suitable workaround to the hardened out-of-bounds checks. Note that previous work has already shown that some of these mitigations can be circumvented [35].

Building Blocks. An alternative to the high-resolution timer is a counting thread which is commonly used for microarchitectural attacks in JavaScript [29, 53, 80]. Furthermore, as the `clflush` instruction is not available in JavaScript, we resort to Evict+Reload as described in related work [29, 76, 90]. Instead of measuring only one address in our Evict+Reload, we use amplification on multiple cache lines [63]. With amplification, we encode the out-of-bounds access into multiple different cache lines to achieve more reliable results. To access a kernel address during transient execution, we hide an out-of-bounds array access behind a misspeculated branch.

EchoLoad from JavaScript. By combining the building blocks, we can implement EchoLoad in JavaScript. On average, it takes 25.09 ms ($n = 10^3$, $\sigma_{\bar{x}} = 5.92$) to find the start of the kernel image. The detected offset is relative to the base of the array, which is used for the out-of-bounds accesses. However, an attacker can leverage any JavaScript ASLR break [29] to recover the array base address, and from that compute the absolute address of the kernel image.

Meltdown from JavaScript. Contrary to Linux, many 32-bit OSs still in use do not have Meltdown patches (e.g., Windows XP). Hence, we show that with the building blocks, we can mount a Meltdown attack from JavaScript on such systems. Relying on EchoLoad for the KASLR break, we can even target specific locations in the kernel.

To evaluate the attack performance of the proof of concept, we disable KPTI and leak a known value from the kernel. Our JavaScript attack leaks 2 B/s, with an error rate of 0.3 % ($n = 10^3$).

Table 3: We compare microarchitectural attacks on KASLR. EchoLoad outperforms all previous microarchitectural attacks on KASLR while having no requirements.

Attack	Time	Accuracy	Requirements
Hund et al. [37]	17 s	96 %	-
Gruss et al. [32]	500 s	N/A	cache eviction
Jang et al. [49]	5 ms	100 %	Intel TSX
Evtyushkin et al. [21]	60 ms	N/A	BTB reverse engineering
Canella et al. [9]	0.27 s	100 %	MDS vulnerable CPU
Schwarz et al. [76]	42 µs	100 %	Intel CPU before Cascade Lake
EchoLoad (our attack)	29 µs	100 %	-

3.5 Other Side-Channel Attacks on KASLR

Microarchitectural attacks on KASLR so far relied on either branch-predictor states [21], address-translation caches [32, 37, 49], or store-buffer optimizations [9, 76]. We compare EchoLoad to previous attacks on KASLR [21, 32, 37, 49, 76].

Our attack outperforms all state-of-the-art KASLR breaks on Intel x86 CPUs (cf. Table 3). We outperform Data Bounce [76] in terms of speed and match it in accuracy while having lower requirements. Similar to Data Bounce, EchoLoad also has the advantage over previous microarchitectural attacks that it does not require Intel TSX, or knowledge of internal data structures like the branch-target buffer (BTB) or the store buffer.

For instance, Evtyushkin et al. [21] assume an attacker knows how the BTB works internally, which has not yet been reverse-engineered for microarchitectures after Haswell. Moreover, with the widely-deployed Spectre mitigations [10, 39], the BTB is either cleared on context switch or not shared between privilege levels. Hence, this attack does not work on state-of-the-art CPUs anymore.

The double page-fault attack [37] was the first microarchitectural attack on KASLR. By accessing a kernel memory location, an attacker first triggers a page fault. This triggers an interrupt which is handled by the OS. After handling the interrupt, the OS returns control to a pre-installed error handler in the user-space

program. In the error handler, the attacker measures the time it took to handle the fault. The attacker then repeats the attack step, again measuring the time it took to handle the fault. If the kernel address is valid, the first illegal access has created a TLB entry. This speeds up the handling of the second fault, creating a timing side channel. Consequently, a user-space attacker can infer whether a kernel address is valid or not. The requirement for this attack is that the user can install a signal handler to handle segmentation faults. Hence, native code execution is required.

Jang et al. [49] retrofitted the attack by Hund et al. [37] with Intel TSX. TSX is an x86 instruction-set extension introducing hardware transactional memory. If a page fault occurs within a transaction, it is aborted without architecturally raising a fault and, hence, without any OS interaction. This allows the attack to skip the page fault handling of the OS, significantly speeding up the attack and reducing its noise. The approach by Jang et al. [49] only works on CPUs starting from Haswell as it relies on Intel TSX. This extension is not present on low-end CPUs or any CPUs built before 2013 and can be disabled on newer CPUs as well. Intel TSX is, for example, disabled on the Amazon EC2 cloud [79].

Gruss et al. [32] use the software prefetch instruction as a side channel. This side channel exploits that the execution time of the prefetch instruction depends on whether the translation cache holds the correct entry. As the TLB can only hold addresses for which a valid translation, *i.e.*, a physical page is mapped to it, the location of the kernel is revealed due to it consisting of the only valid address mapping within the predefined region. With this attack, the attacker additionally learns the page size that is used for the mapping.

Fallout [9] demonstrates a KASLR break on MDS-vulnerable Intel CPUs. First, they ensure that a user-controlled value is in the store buffer. Then, they attempt to access an address with the same *page offset* ,which is inaccessible. On MDS-vulnerable CPUs, the store-buffer content is transiently forwarded to faulting loads on valid kernel addresses, revealing the location of the kernel. Fallout [9] relies on the opportunistic store-buffer behavior that virtual addresses are likely equivalent if the least-significant 12 bits match. However, this is only the case on MDS-vulnerable Intel CPUs that are not patched. Hence, this KASLR break does not work on CPUs indicating that they are MDS-resistant via the `MDS_NO` flag in the `IA32_ARCH_CAPABILITIES` model-specific register, e.g., on the newest Cascade Lake CPUs.

Data Bounce [76] breaks KASLR by exploiting that the CPU only performs store-to-load forwarding if a physical page backs a virtual address, *i.e.*, the virtual address can be resolved to a physical address. Using this approach, they can break KASLR on all Intel CPUs going back to 2004. They claim that the attack has perfect accuracy and only requires 42 µs to detect the correct kernel location. One of the advantages of this approach over Jang et al. [49] is that it does not require TSX and, hence, is applicable to a broader range of CPUs. However, the behavior of store-to-load forwarding was changed in Cascade Lake CPUs to prevent this attack (cf. Table 1). Hence, while their approach works on microarchitectures starting from the Pentium 4 Prescott to Whiskey Lake and Coffee Lake R, it does not work on the recent Cascade Lake.

EchoLoad relies on the load stalling behavior of the CPU, an effect which has not been exploited so far. As this effect is deeply rooted in the design of the microarchitecture, it cannot easily be fixed (cf. Section 3.1), neither in software nor hardware. Moreover, the attack does not have any requirements as it solely relies on memory loads. As a consequence, even the most recent Cascade Lake is affected by EchoLoad.

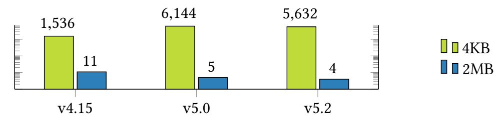

Figure 5: On recent Linux, several 2 MB pages in the kernel text segment have been replaced by 4 kB pages.

4 FLARE: MITIGATING KASLR BREAKS

In this section, we propose FLARE, a defense against KASLR attacks rooted in a CPU's microarchitecture.

FLARE has a negligible memory overhead of only a few kilobytes and next to no runtime overhead. FLARE tackles the root causes of all the microarchitectural KASLR breaks discussed in Section 3.5. It builds on ideas from KAISER [31] and LAZARUS [26] to fix remaining weaknesses efficiently and securely.

The challenge is to fully eliminate differences in:

C1: timing and behavior for mapped and unmapped pages,
C2: timing for different page sizes, and
C3: timing between executable and NX pages.

As we show in this section, FLARE successfully tackles these challenges. However, before we justify these challenges, we briefly introduce a threat model. We then discuss implementation details, corner cases, and pitfalls in Section 4.1.

Threat Model. Our attacker can run unprivileged native code on an up-to-date OS. Furthermore, the attacker knows the exact version of the Linux kernel that the victim uses and, hence, knows the exact structure of the kernel image in memory.

C1: Differences for Mapped and Unmapped Pages. In Section 3.5, we discuss that recent attacks, including EchoLoad, can distinguish mapped from unmapped pages [9, 32, 37, 49, 76]. Therefore, the first challenge is to prevent an attacker from detecting the KASLR offset based on that information.

To tackle this challenge, we map all unmapped virtual addresses in the randomization range to a dummy physical page. Therefore, none of the known attacks that rely on distinguishing mapped from unmapped addresses can de-randomize the kernel anymore.

C2: Timing Differences for Page Sizes. In Section 3.5, we discuss that the attack by Gruss et al. [32] can distinguish different page sizes: Even if the entire kernel space has a valid mapping, different page sizes can create a unique pattern which de-randomizes the kernel. This is especially a problem as the kernel uses different page sizes for its mapping (cf. Figure 5), possibly creating such a unique pattern. We tackle this challenge by avoiding different page sizes in the kernel altogether.

C3: Timing Difference between Executable and NX Pages. Jang et al. [49] showed that there is a timing difference between executable and NX pages. We analyzed the kernel and discovered that executable and NX pages are strictly separated. That is, after the first NX page in the address space there is not a single executable page in the remaining address space. To prevent this straightforward KASLR break, we randomize the executable and

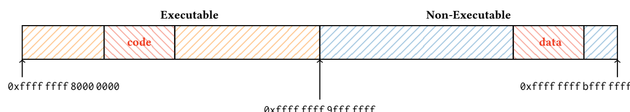

Figure 6: With FLARE, all possible kernel offsets are physically backed, *i.e.*, any potentially read value from this range will be zero. Code and data is independently randomized in 512 MB ranges. This setup allows preventing all currently known microarchitectural attacks on KASLR.

the NX range separately and pad them each with executable and NX pages respectively to the full randomization range.

4.1 Implementation Details

The different Linux kernel regions (cf. Section 2.5) are mapped with different properties, *i.e.*, different page sizes and permissions (e.g., executable and NX). Note that we only need to protect the trampoline code if KPTI is active, while we have to protect the following regions without KPTI.

Text Segment. Figure 5 shows that the text segment is mapped using both 4 kB and 2 MB pages. To address C1, we map the entire range where the text segment can be mapped using 4 kB pages, preventing the attacker from seeing the actual text-segment range. To address C2, we map the text segment only with 4 kB pages, preventing attacks that distinguish page sizes [32]. This is not a large kernel change as this is already an ongoing development (cf. Figure 5). The kernel already supports disabling the use of non-4 kB pages by clearing the CPU capability `X86_FEATURE_PSE`.

Furthermore, to tackle C3, we use the solution shown in Figure 6. We split the randomization range of the text segment in half. We then use one half for the randomization of executable pages, *i.e.*, the kernel code, and the other for NX pages, *i.e.*, the kernel data. Both regions are then randomized independently to not leak their corresponding start and end addresses. This split does not introduce any compatibility issues, even with relative addressing, as we stay within the maximum addressable range of 4 GB.

Modules. Modules already use 4 kB pages only, solving C2. In our proposal, we pad the code and data sections of every module to a multiple of 1 MB, depending on the size of the largest currently loaded module. We then map the remaining offsets in the address range with dummy modules using 4 kB pages that look exactly the same as the actual modules, *i.e.*, same size for code and data sections. Consequently, using the technique by Jang et al. [49] in the memory range for kernel modules, we only see executable and NX regions of all the same size. This mitigates the templating attack by Jang et al. [49] as the attacker cannot infer anymore which module is real and which one is not. With this approach, we solve all three challenges.

Naturally, the privileged user can dynamically load modules which takes the place of a previous dummy module. Likewise, for the unload, a dummy module replaces the kernel module mapping. However, the implementation should be careful not to leave a small time window open for an attack. In our FLARE proof-of-concept, we enable the loading of modules by first removing the dummy mapping by hooking the function *load_module*. Then the module is loaded, and afterward, the mitigation is re-applied. However, a proper implementation should exchange the page-table entries directly instead. This way, it is guaranteed that no time window is left for the attacker to observe the short unmapping from a concurrent microarchitectural attack, as there simply is no short unmapping. Furthermore, the loading and unloading of modules typically does not happen for an average user.

Direct-Physical Map, Vmalloc, Vmemmap. We analyzed how the direct-physical map, vmalloc, and vmemmap are mapped. None of the pages mapped in this region is executable. Thus, we tackle challenges C1 and C3 by mapping all pages in the corresponding randomization regions in our dummy mapping as NX.

Currently, the kernel does not use an explicit randomization range for each of the three regions. Instead, the kernel uses one large range and only guarantees to preserve their order. To mitigate the attack by Gruss et al. [32], all three must use the same page size. We verified that this is already the case when clearing the `X86_FEATURE_PSE` CPU capability at boot. As this causes significant pressure on the TLB, we propose a different approach.

We propose that the kernel uses an explicit randomization range for each of the three regions. Hence, to tackle C2, we can enforce that the kernel consistently uses one page size per region. This mitigates the attack by Gruss et al. [32]. In the analysis for our defense, we empirically determined the page sizes used for each region. On our test machine running Linux kernel 4.15, the kernel indicates during the boot process that 1 GB pages are used for mapping the direct-physical map. However, our analysis revealed that it is mapped using all three page sizes, *i.e.*, 4 kB, 2 MB, and 1 GB. Similarly, the vmalloc area uses both 4 kB and 2 MB pages. The vmemmap area consisted of 2 MB pages only.

Based on this analysis, we propose to consistently use 2 MB pages for the vmemmap region and 4 kB pages for the vmalloc region. For the direct-physical map, we use 1 GB pages. Unfortunately, we cannot use such a huge dummy page for our mapping as we would reduce the available physical memory by 1 GB. Instead, we pick 1 GB of RAM, which is already mapped in the direct-physical map, and map it using a 1 GB page in our dummy mapping. Hence, we avoid the additional memory overhead without increasing the risk for exploitation as we map the page as NX.

5 EVALUATION

In this section, we evaluate the overhead of FLARE in three aspects, namely runtime overhead using the SPEC CPU 2017 benchmarks [16], module loading overhead, as well as the memory overhead. We also evaluate the efficacy of FLARE by analyzing how successful it is in preventing microarchitectural attacks on KASLR.

5.1 Overhead Analysis

Runtime. We create our dummy mapping directly in the *init_mm* struct which is copied into every newly created process. We only have to apply our mapping once, and every new process has the mitigation enabled. Hence, we expect no runtime overhead.

We confirmed this using the LMbench microbenchmark suite [64]. We evaluated process-creation time (fork and exec) and context switches on an Intel i7-8650U (Linux kernel 5.0.0-15). This involves a larger number of TLB invalidations and address resolutions, *i.e.*, the situations that may see a performance penalty. For process creation, we do not encounter any overhead. Both with and without

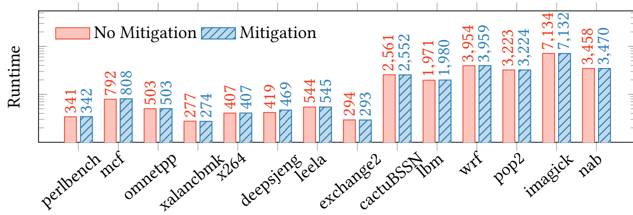

Figure 7: Runtime overhead of FLARE on SPEC CPU 2017.

FLARE, the process creation takes on average 61.14 µs ($n = 10^5$, $\sigma_{\bar{x}} = 0.27$). Similarly, there is no difference in the syscall latency. In both cases, the latency is on average 1.03 µs ($n = 10^5$, $\sigma_{\bar{x}} = 0.006$).

For a real-world workload, we evaluated the runtime overhead using the SPEC CPU 2017 benchmark. We ran the benchmark once with our mitigation and once without it on an Intel Xeon Silver 4208. We excluded some benchmarks in both the *intspeed* and *fpspeed* benchmark as they already crashed or did not compile on our vanilla Linux system. Figure 7 shows the results. As expected by the design of FLARE, we exhibit next to no runtime overhead.

Module Loading. Next, we evaluated the increase in module loading time. We first establish a baseline by loading and unloading a simple test module 10^4 times. We then load the FLARE proof of concept, which requires removing and re-applying the dummy mapping for every module load, thus overapproximating the overhead of our mitigation. We again load and measure the required time 10^4 times. We only observe a 4 % increase from 2.39 ms to 2.48 ms per module load. When implemented in the Linux kernel, the module memory allocation logic is made aware of the dummy mappings so that they are treated like free memory. Thus, overheads are avoided entirely except in cases where the modules have to be re-padded, where we observe the overheads to be negligible.

Memory. Finally, we analyzed the memory overhead of FLARE, which is minimal in our proof of concept. We always map the same dummy page in the paging hierarchy and re-use the same page directory and page table. We do not need a new PDPT, as we are working on existing 1 GB ranges. Therefore, we only require one page each for the new page directory, page table, and one page to point to. As all these pages are 4 kB, the maximum overhead is 12 kB. To map huge dummy pages, the maximum overhead is only increased by 2 MB. The direct-physical map padding with 1 GB pages does not consume additional memory (cf. Section 4).

5.2 Mitigate Microarchitectural KASLR Breaks

In a first step, we evaluated the effectiveness of FLARE in preventing breaking the randomization of the kernel text segment. Using a vanilla Linux 5.0 kernel, we test microarchitectural attacks on KASLR that are not mitigated through orthogonal countermeasures (cf. Section 3.5) with KPTI disabled (cf. Figure 8). In all cases, we first establish a baseline of the attack without FLARE in place, which shows the exact position of the kernel with all attacks.

We then load FLARE and re-evaluate all attacks. We see for each attack that the kernel can no longer be distinguished from other positions. With EchoLoad (Figure 8a), all offsets are backed by a physical page, the load succeeds, but the CPU returns zero for the illegal access. The stall percentage is based on cache hits and misses on the probe array, not performance counter values. With the prefetch side channel (Figure 8b), we see that the prefetch instruction can now also prefetch all other possible locations, mitigating the KASLR break. Data Bounce (Figure 8c) also no longer distinguishes kernel locations from dummy mappings as store-to-load forwarding works for all possible offsets. The double-page fault (Figure 8d) as well as the DrK attack (Figure 8e) also do not work anymore, exhibiting the same timing across the whole address range. With Fallout (Figure 8f), we also see no difference anymore as every page allows to trigger the WTF effect. An attack that tries to detect our dummy mapping based on timing the page-table walk is also not possible. Even though the physical page is shared across all dummy mappings, a TLB entry for one mapping is not shared with another. Hence, each access to a new page requires a full page-table walk. Our dummy mapping can also not be uncoverd via the cache as an access to a privileged address does not load the data into it [42, 76, 79]. Based on the results shown in Figure 8, none of the currently known microarchitectural attacks that are not mitigated through orthogonal countermeasures (cf. Section 3.5) can de-randomize the kernel location despite FLARE. This empirically confirms that we solve challenge C1.

Next, we de-randomize the kernel based on the timing difference between executable and NX pages [49]. We confirm that tackling only C1 and C2 is insufficient (cf. Figure 9). However, full FLARE (cf. Figure 9) separates the regions and the switch from executable to NX is not visible in this region anymore but at the pre-defined start of the randomization range (cf. Figure 6).

Next, we used the prefetch side-channel attack to try to break KASLR based on different page sizes. The different levels visible in the default case of Figure 8b show the different paging levels for the address we test. If nothing is mapped in the PML4, we observe the highest time. There is a drop in the access time for addresses with no PDPT entry, and another drop for addresses that map to an entry in the page table, *i.e.*, a 4 kB page. Thus, the prefetch side channel shows the different paging levels [32]. With FLARE in place, we can no longer see the difference in page sizes as all possible locations as well as the kernel are mapped using 4 kB pages. Thus, we empirically confirmed that our strategy for C3 works, defeating microarchitectural attacks on KASLR based on different page sizes.

6 RELATED WORK

With the advent of KASLR, many different attacks have been proposed to break KASLR. One problem is that the kernels of the major OSs cannot change the randomization at runtime. Hence, if an attacker knows the KASLR offset, it is valid until the next time the OS is rebooted. So far, most of the attacks on KASLR relied either on software vulnerabilities or side-channel attacks on the microarchitecture as discussed in Section 3.5.

6.1 Software-based KASLR Breaks

On Linux, parts of kernel pointers are often disclosed inadvertently through kernel interfaces, e.g., due to uninitialized structure fields or structure padding [84]. There have been many such software vulnerabilities in kernels (e.g., CVE-2012-6138, CVE-2013-1825, 1826, 1827, 1873, 2634, 2635, 2636) that revealed parts of kernel addresses. Similarly, for Windows, multiple methods leak kernel pointers, e.g.,

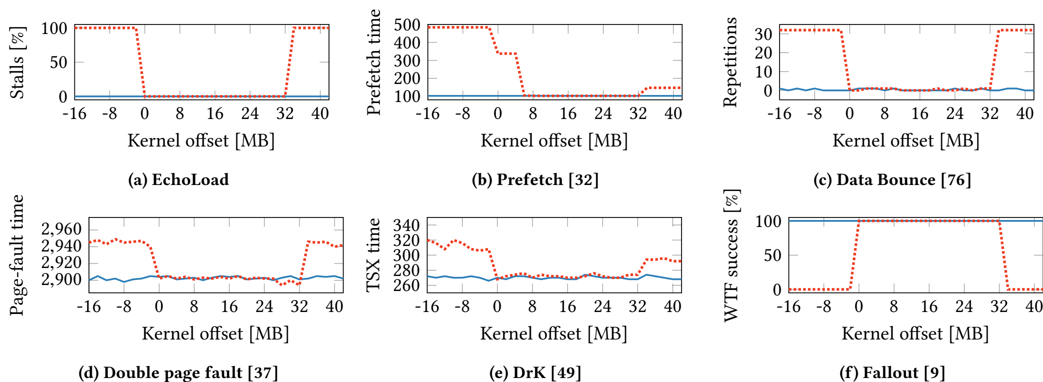

Figure 8: Detecting the kernel (offset 0 to 32 MB) with all known microarchitectural attacks on KASLR without FLARE (⋯⋯) and with FLARE (——). For all attacks, FLARE successfully prevents the detection of the kernel.

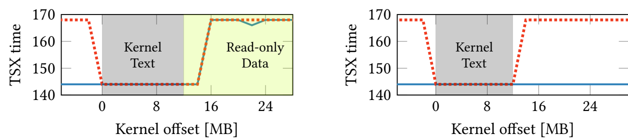

Figure 9: DrK [49] distinguishes executable from NX pages. An attacker can observe the switch to NX pages directly after the executable pages (left), when tackling only C1 and C2 (——) and entirely without (⋯⋯) FLARE. With full FLARE (right), this attack is also fully mitigated.

using the `Win32ThreadInfo` or the Desktop heap [74]. Other attacks on KASLR relied on the fact that kernel addresses were used as unique identifiers [84], or as seed for pseudo-random numbers [52]. For debugging reasons, kernel addresses were often visible in log files or debugging interfaces such as the perf subsystem [18].

6.2 Mitigating Software-based KASLR Attacks

While software bugs causing KASLR breaks can be easily fixed, there are also general concepts for preventing address leakage from the kernel. Linux introduced a setting to mask kernel pointers in log files with a random mask [73]. Thus, a developer sees which pointers are the same, but an attacker cannot learn the actual pointer value and, thus, the KASLR offset. This mitigation reduces the risk of leaking the KASLR offset without impairing the debugging capabilities. The PaX Team proposed STACKLEAK [15], a mechanism to clear kernel-stack memory which is no longer in use. This reduces accidental address leakage from uninitialized stack values.

6.3 Mitigate Microarchitectural KASLR Breaks

While microarchitectural attacks on KASLR cannot be simply fixed in software, there are software-based workarounds. Gruss et al. [31, 32] proposed stronger kernel isolation to prevent microarchitectural attacks on the kernel by unmapping the kernel address space when running in user space. Thus, in theory, there is no valid kernel address in user space, preventing all microarchitectural attacks on the kernel. However, while their proposal is deployed on all major OSs to prevent Meltdown [59], it cannot prevent our KASLR break (cf. Section 3.2). The reason is that x86 requires some kernel pages always to be mapped, even when running in user space [31].

Lazarus [26] proposed a similar approach to KAISER [31]. It is based on fencing the kernel paging entries off from those of the user space by separating user and kernel page tables. Therefore, the Memory Management Unit can no longer use entries pointing to kernel space memory from user space. Contrary to KAISER, Lazarus uses dummy mappings to hide the context switching code while KAISER separates it from the rest of the kernel code section. However, it does not tackle the challenges we identified and does not defeat all known microarchitectural attacks on KASLR.

7 CONCLUSION

In this paper, we analyzed Intel's recent hardware fixes for Meltdown. Our analysis led to the understanding that illegal memory accesses do not lead to a CPU stall, but instead, the illegally loaded data is zeroed-out. With EchoLoad, we presented a novel technique based on Flush+Reload to distinguish stalling loads from transiently executed ones. Hence, EchoLoad enables an attacker to detect physically-backed kernel addresses and break KASLR. Our KASLR break is the fastest and most reliable microarchitectural KASLR break presented so far, taking only 40 μs to de-randomize the kernel. The only requirement for EchoLoad are memory loads, allowing it to be mounted from restricted environments such as SGX and JavaScript. We presented the first JavaScript-based Meltdown attack and KASLR break on the systems that do not receive Meltdown patches, *i.e.*, x86 32-bit OSs.

With FLARE, we proposed a generic approach for protecting the kernel against microarchitectural KASLR breaks. We verified that FLARE mitigates the root cause behind current microarchitectural KASLR breaks and yields a uniform behavior across the kernel address space. Thus, considering the state of the hardware mitigations, we propose to deploy FLARE even on the most recent CPU generations.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their comments and suggestions that helped improving the 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). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO,

which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Additional funding was provided by generous gifts from Intel, ARM, and Cloudflare. 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] Tiago Alves. 2004. TrustZone: Integrated Hardware and Software Security.

[2] Apple Inc. 2012. OS X Mountain Lion Core Technologies Overview. http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf

[3] ARM Limited. 2018. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism.

[4] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. 2014. Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In *CHES*.

[5] Daniel J. Bernstein. 2004. Cache-Timing Attacks on AES.

[6] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *S&P*.

[7] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*.

[9] 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. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*.

[11] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity.. In *USENIX Security Symposium*.

[12] Nicholas Carlini and David A. Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security*.

[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *CCS*.

[14] Liang Chen and Qidan He. 2016. Shooting the OS X El Capitan Kernel Like a Sniper.

[15] Jonathan Corbet. 2018. Preventing kernel-stack leaks. https://lwn.net/Articles/748642/

[16] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[17] Ian Cutress. 2018. Spectre and Meltdown in Hardware: Intel Clarifies Whiskey Lake and Amber Lake. https://www.anandtech.com/show/13301/spectre-and-meltdown-in-hardware-intel-clarifies-whiskey-lake-and-amber-lake

[18] Lizzie Dixon. 2017. Breaking KASLR with perf. https://blog.lizzie.io/kaslr-and-perf.html

[19] ecma international. 2018. ECMAScript 2018 Language Specification. https://www.ecma-international.org/ecma-262/9.0/index.html

[20] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/

[21] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*.

[22] Agner Fog. 2016. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.

[23] Ulf Frisk. 2016. Windows 10 KASLR Recovery with TSX. http://blog.frizk.net/2016/11/windows-10-kaslr-recovery-with-tsx.html

[24] Thomas Garnier. 2016. Kernel memory randomization and trampoline page tables. https://medium.com/@mxatone/kernel-memory-randomization-and-trampoline-page-tables-9f73827270ab

[25] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[26] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. 2017. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *RAID*.

[27] Jason Gionta, William Enck, and Per Larsen. 2016. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *Communications and Network Security (CNS)*.

[28] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *S&P*.

[29] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.

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

[31] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.

[32] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.

[33] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

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

[35] Noam Hadad and Jonathan Afek. 2018. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/

[36] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass.

[37] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.

[38] Intel. [n.d.]. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[39] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf

[40] Intel. 2018. Speculative Execution Side Channel Mitigations. Revision 3.0.

[41] Intel. 2019. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling

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

[43] Intel. 2019. Performance Monitoring Impact of Intel Transactional Synchronization Extension Memory. https://cdrdv2.intel.com/v1/dl/getContent/604224

[44] Alex Ionescu. 2016. Twitter: Windows KASLR. https://twitter.com/aionescu/status/725399988306644992

[45] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P*.

[46] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14*.

[47] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *CCS*.

[48] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*.

[49] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*.

[50] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption.

[51] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[52] Amit Klein and Benny Pinkas. 2019. From IP ID to Device ID and KASLR Bypass. In *USENIX Security*.

[53] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.

[54] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.

[55] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*.

[56] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium*.

[57] Linux. 2019. Complete virtual memory map with 4-level page tables. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

[58] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[60] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[61] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.

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

[63] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178* (2019).

[64] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*.

[65] Andrei Mogage, Rafael Pires, Vlad Crăciun, Pascal Felber, and Emanuel Onica. 2019. Supply chain malware targets SGX: Take care of what you sign (Practical Experience Report). In *SRDS*.

[66] Mozilla. 2019. Index Masking in Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=1430051

[67] Mozilla. 2019. performance.now resolution. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

[68] Net Applications.com. 2019. Desktop Operating System Market Share. http://www.netmarketshare.com/operating-system-market-share.aspx

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.

[70] Matthew Panzarino. 2012. Apple releases OS X 10.8 Mountain Lion Developer Preview 2, lists known issues. https://thenextweb.com/apple/2012/03/16/apple-releases-os-x-10-8-mountain-lion-developer-preview-2-to-mac-developers/

[71] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.

[72] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. kRˆ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *EuroSys*.

[73] Dan Rosenberg. 2010. kptr_restrict for hiding kernel pointers. https://lwn.net/Articles/420403/

[74] Morten Schenk. 2019. Development of a new Windows 10 KASLR Bypass (in One WinDBG Command). https://www.offensive-security.com/vulndev/development-of-a-new-windows-10-kaslr-bypass-in-one-windbg-command/

[75] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*.

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

[77] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2018. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In *AsiaCCS*.

[78] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.

[79] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[80] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*.

[81] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical Enclave Malware with Intel SGX. In *DIMVA*.

[82] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.

[83] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*.

[84] Brad Spengler. 2013. KASLR: An Exercise in Cargo Cult Security. https://grsecurity.net/kaslr_an_exercise_in_cargo_cult_security.php

[85] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

[86] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P*.

[87] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[88] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[89] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[90] Pepe Vila, Boris Köpf, and Jose Morales. 2019. Theory and Practice of Finding Eviction Sets. In *S&P*.

[91] WebAssembly. 2019. *Features to add after the MVP*. https://github.com/WebAssembly/design/blob/master/FutureFeatures.md

[92] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*.

[93] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf.

[94] Zhenyu Wu, Zhang Xu, and Haining Wang. 2014. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. *IEEE/ACM Transactions on Networking* (2014).

[95] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *CCSW'11*.

[96] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.

[97] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*.

[98] Peter Zijlstra. 2019. Implement support for TSX Force Abort. https://lkml.org/lkml/2019/3/12/1352