Computers & Security 134 (2023) 103437

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

AVX-TSCHA: Leaking information through AVX extensions in commercial processors

Suryeon Kim [a], Seungwon Shin [a], Hyunwoo Choi [b,*]

[a] Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea
[b] National Security Research Institute (NSR), Republic of Korea

ARTICLE INFO

Keywords:
Kernel address space layout randomization
KASLR
Side-channel attack
Advanced vector extensions
AVX

ABSTRACT

Modern x86 processors support an AVX instruction set to boost performance. However, this extension set may also cause security issues. We discovered that there are vulnerable properties in the implementation of the masked load/store instructions. First, these instructions can suppress exceptions caused by invalid or inaccessible memory access. Second, the execution time of these instructions leaks the current state of the page mappings, permissions, and TLB states.

Based on this, we present a novel AVX timing side-channel attack that can defeat address space layout randomization. We demonstrate the significance of our side-channel attack by showing User and Kernel ASLR breaks on the recent Intel and AMD processors in various environments, including cloud computing systems (Amazon AWS, Google GCP, and Microsoft Azure), an SGX enclave (a fine-grained ASLR break), and major OSes (Linux, Windows, and macOS). Our attack can identify the Linux kernel's base address in 0.29 *ms* as well as those of loaded kernel modules in 2.24 *ms*, with a near-zero error rate. We further demonstrate that our attack can be used to infer user behavior, such as mouse movements and data transmissions over the network. Our evaluation results on multiple mobile, desktop, and server processors (a total of 26 Intel and AMD CPUs) show that 1) the AVX timing side-channel works on the vast majority of Intel processors (from the Sandy Bridge microarchitecture) as well as AMD processors (from the Zen microarchitecture onward) and 2) our KASLR breaks are very fast and reliable. To the best of our knowledge, our attack is the first to demonstrate a KASLR break on both the recent Intel Alder Lake and AMD Zen 3 CPUs. We highlight that more robust isolation or fine-grained randomization should be adopted to mitigate our presented attacks successfully.

1. Introduction

Modern x86 processors support a Single Instruction Multiple Data (SIMD) instruction set that compilers or programmers can use to boost performance Intel (2021a); AMD (2021). However, this instruction set may also cause security issues. In particular, we discovered that in the Advanced Vector Extensions (AVX), there are vulnerable properties in the implementation of the masked load/store instructions. First, these instructions can suppress exceptions caused by invalid (i.e., unmapped) or inaccessible (e.g., accessing kernel memory from user space) memory access. Second, the execution time of these instructions leaks the current state of the page mappings, permissions, and TLB states.

Based on these vulnerable properties, this paper introduces a novel AVX timing side-channel attack named AVX-TSCHA,[1] which can defeat address space layout randomization (ASLR). Our attack only requires widely supported AVX hardware extension in modern processors, thus significantly lower requirements than the previous attacks. We demonstrate the significance of our attack by showing User and Kernel ASLR breaks on both the most recent Intel (from Sandy Bridge) and AMD (from Zen to Zen 3) processors. To the best of our knowledge, our KASLR break is the first to show a reliable KASLR break on both the recent Intel Alder Lake and AMD Zen 3 processors. Since AVX was introduced on Intel Sandy Bridge and AMD Bulldozer in 2011, our attack can be applied to the vast majority of modern mobile, desktop, server, and cloud systems. We evaluated our attack on a total of 26 Intel and AMD CPUs, and as a result, we verified that our attack works on all processors we tested.

Specifically, we show that our attack reliably retrieves the base address of the Linux kernel text in 0.29 ms with a near-zero error rate.

* Corresponding author.
E-mail address: zemisolsol@nsr.re.kr (H. Choi).

[1] **AVX**-Timing **S**ide-**CH**annel **A**ttack.

https://doi.org/10.1016/j.cose.2023.103437
Received 20 April 2023; Received in revised form 13 August 2023; Accepted 15 August 2023
Available online 30 August 2023

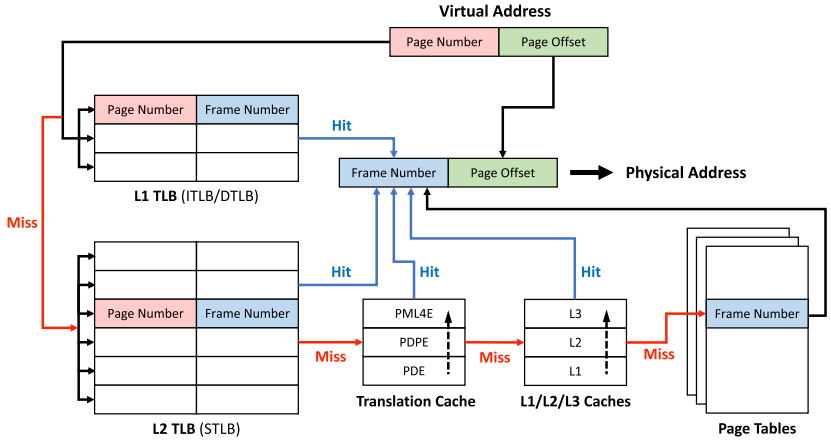


Fig. 1. Caches involved in a TLB hit/miss. The processor retrieves the translation in the order of TLB, second level TLB (STLB), page-translation caches, data caches, and page-tables in the main memory.

For kernel modules, based on a unique size, our attack can identify the currently loaded modules in 2.24 ms with 99.84% accuracy. The attack works even on a kernel page table isolation (KPTI)-enabled kernel. This also shows that our attack can be used to infer user behavior by monitoring kernel activities such as Bluetooth events, mouse movements, and Wi-Fi transmissions. Furthermore, we demonstrate the applicability of our attack by showing KASLR breaks in various environments, including cloud computing systems, a Software Guard Extensions (SGX) enclave, and other major operating systems (OSes). In particular, we show how our attack can be used to mount a fine-grained ASLR break inside an SGX enclave. Our side-channel can be applied to establish a covert-channel between simultaneous multi-threading (SMT) threads and serve as a transmitter for Spectre attacks.

Since our attack only requires AVX instructions, it makes it much more practical compared to known microarchitectural attacks that depend on noise filtering Hund et al. (2013), hardware transactional memory (Intel TSX) Jang et al. (2016), cache eviction Gruss et al. (2016); Lipp et al. (2022), knowledge of the branch target buffer (BTB) hash function Evtyushkin et al. (2016), the TLB addressing Koschel et al. (2020), cache status monitoring Canella et al. (2019, 2020a); Weber et al. (2021), or the energy reporting interface (e.g., RAPL) Lipp et al. (2021, 2022). We highlight that stronger isolation or more fine-grained randomization should be adopted to successfully mitigate our presented attacks.

This paper extends a study that was initially reported in Hyunwoo et al. (2023) and provides the following contributions:

- We conduct an in-depth analysis of the AVX masked operations and discover vulnerable properties in the implementation of masked load/store instructions.
- We introduce a novel AVX timing side-channel named `AVX-TSCHA`, which is capable of defeating address space layout randomization.
- We show that our attack can detect user behavior and is feasible in cloud computing systems (Amazon EC2, Google GCE, and Microsoft Azure), an SGX enclave (a fine-grained ASLR break), and major operating systems (Linux, Windows, and macOS).
- We evaluated our attack on multiple mobile, desktop, and server processors (a total of 26 Intel and AMD CPUs). The results indicate that 1) the AVX timing side-channel attack works on the vast majority of Intel and AMD CPUs and 2) our KASLR breaks are very fast and reliable, with a near-zero error rate.

Responsible disclosure and proof-of-concept code. We responsibly disclosed our findings to Intel on April 20, 2022, and AMD on July 3, 2022. Intel acknowledged our findings on May 10, 2022, and AMD on July 5, 2022. Our attack code is available at https://github.com/zemisolsol/kaslrAVX.

2. Background

2.1. Address translation and translation caches

Address translation. Virtual addresses are translated into physical addresses through multi-level page tables. On Intel x86-64 processors, page tables are comprised of four levels of paging structures: page map level 4 (PML4), page directory pointer table (PDPT), page directory (PD), and page table (PT). Each page table defines the mappings from a virtual address to a physical address, and address translation is performed by indexing certain parts of the virtual address. The virtual address space is divided into user space and kernel space, which serves to provide memory protection. To this end, the page tables contain permission-related information, such as a readable/writable page and a user/kernel accessible page.

Translation lookaside buffer. TLB is a special cache that contains the most recently used page table entries (PTEs). Given a virtual address, the processor examines the TLB. If a PTE is present (i.e., a *TLB hit*), the corresponding page frame number (PFN) is retrieved, and the physical address is formed. On the contrary, if the PFN is not found in the TLB (i.e., a *TLB miss*), the processor starts to walk the page table hierarchy (called a page table walk) to look for the corresponding PFN. On page table walks, a memory management unit (MMU) accesses each page table to find the translation for the virtual address. Once the translation is found, meaning that the page is mapped and no page fault (*#PF*) occurs, the TLB is updated to include the new page entry. Otherwise, a *#PF* is issued, and the operating system handles the *#PF*.

Page-translation caches. The page table can still be cached like any other read from normal memory. However, frequently accessing the PTEs on every page walk will incur a penalty of several tens of cycles per TLB miss, even if all entries are present in the data cache (e.g., L2). For this reason, modern processors may have *page-translation caches* (Intel refers to these as *paging-structure caches* Intel (2016)) to further improve the performance of the TLB miss Barr et al. (2010). Fig. 1 depicts how different caches interact to handle a TLB hit/miss.

2.2. KASLR and kernel page-table isolation

Kernel Address Space Layout Randomization. KASLR is a technique used to randomize the base address of a kernel image and the position of kernel modules at boot or driver load time. Once KASLR is enabled, it defeats code reuse attacks such as return-oriented programming (ROP) Shacham (2007), which rely on knowledge of the

```
#include <immintrin.h>

int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
__m256i msk = _mm256_setr_epi32(-1, -1, -1, -1, 0, 0, 0, 0)
    ;
__m256i res = _mm256_maskload_epi32(arr, msk);
```

Listing 1: An example of Intel intrinsics for an AVX2 masked load. In this code, only four elements (`arr[0]-arr[3]`) are loaded due to the mask values.

absolute address of instructions. In an x86-64 Linux kernel, the kernel image is aligned to a 2 MiB boundary and mapped between `0xffffffff800`00000 and `0xffffffffc00`00000 with a maximum size of 1 GiB (i.e., 512 possible offsets). Although the entropy of the randomization is only 9 bits, brute force attacks against the randomization in the kernel are virtually not feasible due to the high rate of kernel panic. Therefore, attackers must typically de-randomize the kernel address space layout (e.g., via information leaks) before continuing with subsequent attacks.

Kernel Page-Table isolation. KPTI (previously known as KAISER Gruss et al. (2017)) was first introduced to defend against attacks on KASLR. With KPTI, kernel space is isolated from user space; thus, it undermines attacks based on the status of page mappings Hund et al. (2013); Gruss et al. (2016); Jang et al. (2016); Evtyushkin et al. (2016); Canella et al. (2019). Major OSes have adopted this isolation technique as mitigation for a Meltdown attack Lipp et al. (2018) (Linux PTI Corbet (2017), Microsoft Kernel Virtual Address Shadow (KVAS) swiat (2018), and Apple Double Map Ionescu (2018)).

Although the kernel space is isolated from the user space, a few pages (called *KPTI trampoline*) that are used to switch from the user space to the kernel space must still be mapped in the user space. Based on this, prior works Canella et al. (2019); Weber et al. (2021) have demonstrated KASLR breaks in KPTI-enabled kernels.

2.3. Advanced vector extensions

AVX is a SIMD instruction set supported by Intel and AMD processors. With AVX, arithmetic and data transfer operations can be processed simultaneously. Although modern compilers such as GNU GCC and Intel C++ provide automatic vectorization options (e.g., `/Qvec` in the Intel C++ compiler Deilmann et al. (2012)), an advanced user can obtain better performance with AVX programming. As one of the optimizations in AVX, the masked load/store operations (`VMASKMOV` in AVX and `VPMASKMOV` in AVX2) are used to conditionally move packed data elements to/from memory, depending on the mask bits associated with each data element. The mask bit for each data element is the most significant bit in each byte element of the mask. During the execution of the masked load/store instructions, faults can occur when accessing an illegal address. However, if the mask bit is set to "*zero*", it does not issue any exceptions Intel (2021a). In Listing 1, when performing the masked load, only the 1-4 elements of the `arr` array are returned because the corresponding mask bit is set to one. Conversely, since the 5-8 elements' matching mask bits are 0, their returned values will be 0. Assembly codes for all AVX/AVX2 masked operations can be found in Appendix C.

3. In-depth analysis of the AVX timing side-channel

In this section, we first analyze the vulnerable properties of the AVX masked load/store instructions and then present three attack primitives based on the vulnerable properties.

3.1. Fault-resistance

Intel's optimization manual Intel (2021a) describes a fault-resistance property of the AVX masked operations. To verify that a masked load-/store instruction does indeed suppress the exception, we examined

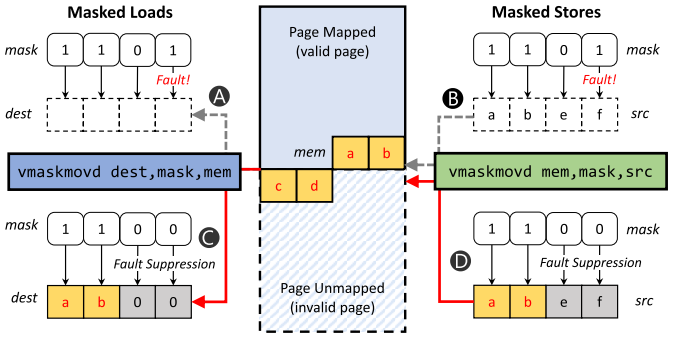


Fig. 2. A fault suppression of the AVX masked load/store instruction. Even on unmapped (or inaccessible) pages, no faults are issued if the mask bits are all set to "*zero*" (C and D).

memory access on both Intel (Ice Lake) and AMD (Zen 3) processors (see Fig. 2). We prepared two adjacent pages using the `mmap/munmap` syscalls; the upper page is mapped (i.e., a valid page), while the lower page is unmapped (i.e., an invalid page). We first executed the masked load/store instructions across the page boundary, where only one element on the low page is masked (**A** and **B** in Fig. 2). We then examined the case where all of the elements on the lower page are masked out (**C** and **D**). From the experiment, we observed that when accessing an unmapped page, no faults occur if the corresponding mask bits are all set to "*zero*". We further tested the masked load/store on kernel memory to determine whether it applies to inaccessible pages. We executed the masked load/store instructions with an arbitrary kernel-mapped address (e.g., the kernel text page). As a result, no faults were observed. Thus, we prove that when executing the masked load/store instruction, masking out does indeed suppress the exceptions, even if the accessed page is invalid or inaccessible.

> $\mathbb{P}1$: The AVX masked operations can suppress the exceptions caused by invalid or inaccessible memory accesses.

3.2. Timing differences

The execution times of the masked operations vary depending on the page-table level, TLB states, page permissions, and instruction types.

Page-Table Level. The masked load and store instructions trigger a *microcode assist* when the address being accessed is invalid or inaccessible Intel (2021a). Considering that microcode assist rectifies the issue by undoing the operation through the purging of pending instructions via a machine clear process Kagan (1997), we speculate that the microcode assist may incur additional cycles because it needs to determine whether the elements have the corresponding mask bits set. To verify this, we measured the execution time of the masked load for different types of pages: USER-M (a page in the user space with Present-bit:1 and User/Supervisor-bit:1), USER-U (P:0), KERNEL-M (a page in the kernel space with P:1 and U/S:0), and KERNEL-U (P:0). We also measured their corresponding microcode assist events (`ASSISTS.ANY`) using a performance counter monitor. Fig. 3 shows the measurement results. On the USER-M page, without issuing a microcode assist, the mean value of an access time is 13 cycles. In contrast, on other pages, the access time is significantly increased due to the microcode assist.

In particular, we observed that the KERNEL-M has a shorter access time (< 14 cycles) than the KERNEL-U. Since the address translation of the unmapped page (P:0) may not be stored in the TLB, we can speculate that the timing difference between the two pages is due to the page table walks. To prove this, when accessing the kernel address, we measured the number of completed page table walks (`DTLB_LOAD_MISSES.WALK_COMPLETED`). As expected, the page table walks were triggered twice in the KERNEL-U but not in the KERNEL-

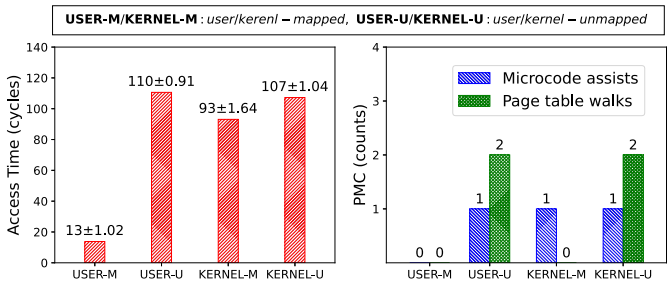


Fig. 3. Execution times of the masked load instructions for different types of pages on an Intel i7-1065G7 (Ice Lake) (on the left side). It also shows the number of corresponding performance counters, *microcode assists*, and *page table walks* (on the right side). USER-M/USER-U refers to a valid or invalid page in the user address space, while KERNEL-M/KERNEL-U refers to a valid or invalid page in the kernel address space.

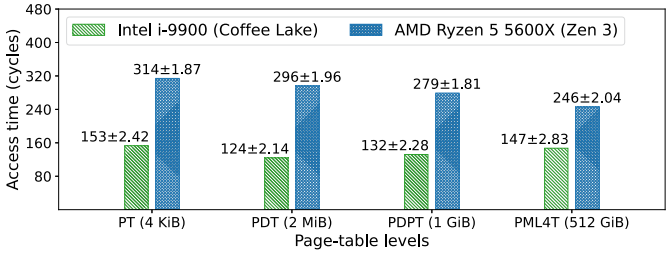


Fig. 4. Execution times of the masked load instructions for different levels of page tables on the Intel i-9900 (Coffee Lake) and AMD Ryzen 5 5600X (Zen 3) processors.

M (right in Fig. 3). Therefore, we prove that the execution time of the masked load/store on kernel-mapped pages is faster than on unmapped pages.

> ℙ2: The masked operations can distinguish between mapped and unmapped pages by measuring execution time.

The execution time of a page table walk varies depending on the level of the page table where the page table walk terminates. Prior works Gruss et al. (2016); Lipp et al. (2022) utilized this to break KASLR by exploiting a software prefetcher. To verify that the masked load/store instruction can also leak information about a page table's level, we measured its execution time on different levels of page tables on the Intel i-9900 (Coffee Lake) and AMD Ryzen 5 5600X (Zen 3) processors. In Ubuntu 20.04.4 (kernel 5.13.0-30), we executed the masked load instruction with four different kernel addresses mapped on PT, PDT, PDPT, and PML4T, respectively. On the Intel processor, since the translation of a valid address is cached in the TLB, we flushed the TLB (using an `INVLPG` instruction in LKM) before the measurement to trigger page table walks. On the AMD processor, however, we observed that the translation of the kernel-mapped address is not stored in the TLB, meaning that it always triggers page table walks. Thus, on AMD, we measured the execution time without considering the TLB state. The detailed experimental results of TLB misses and page table walks on the Intel and AMD CPUs can be found in Appendix A.

Fig. 4 depicts the measured mean value of the masked load execution time on different page table levels. We observe that the AMD processor's execution time increases linearly from the highest level (PML4T) to the lowest level (PT). Conversely, the execution time on the Intel processor exhibits an increase in the opposite direction compared to AMD, with the exception of PT. Notably, on Intel CPUs, the order of the page table lookup in the translation caches is PDE → PDP → PML4 (do not contain PT). Hence, the process of page table walks requires more time when translating a virtual address mapped on a 4 KiB page (PT) compared to huge pages (PDE for a 1 GiB page and PDP for a 2 MiB page).

Fig. 5. A comparison of execution times according to page permissions. A masked load can distinguish two types of pages (`r-/r-x` and `--`), whereas a mask store can distinguish three types of pages (`r-/r-x`, `rw-`, and `--`).

> ℙ3: The masked operations can leak information about the level of the page table where the walk terminates.

TLB state. The execution time of the masked load/store instruction differs depending on the TLB state. If a page table entry is present in the TLB during the address translation (i.e., a TLB miss), it takes less time than a TLB miss. To verify this, we tested memory access on the kernel-mapped page to determine whether the masked operations could distinguish between a TLB hit and miss. On the Intel processor with the KERNEL-M page, we executed the masked load/store instruction twice in a row (first for a TLB miss and then for a TLB hit) and measured each execution time. Before the first access, we evicted the TLB entries Gras et al. (2018); TLB (2022) to ensure the first execution issued a TLB miss. We repeated this test 1000 times on an Intel i9-9900 (Ubuntu 20.04.1 with kernel 5.11.0-27). The experimental results show that the first execution takes an average of 381 cycles for the masked load and 343 for the store, while the second execution takes 147 cycles for the load and 133 for the store, on average. As a result, we confirm that the masked load/store instruction can identify the current TLB state.

> ℙ4: The masked operations can identify the TLB state.

Page permission. The execution time of the masked load/store instruction is affected by page permissions being accessed. To evaluate this, we mapped four pages with different permissions (`read-only`, `read-exec`, `read-write`, and `none`) in the user space and measured each execution time of the masked load/store instructions. The experimental results are shown in Fig. 5. In this figure, the execution time of the masked load differs only in the `none` page permission. However, in the case of the masked store, we observe that write permission (`read-write`) affects the execution time. If the page does not have `write` permission, the masked store triggers a microcode assist, which takes additional cycles. Thus, in the execution of the masked store, the timing differences between read (`read-only` and `read-exec`) and write (`read-write`) permissions are clearly visible. This paper uses this observation to implement a fine-grained ASLR break in user address space (inside an SGX enclave).

> ℙ5: The masked operations can identify page permissions.

Load and store. The masked load and store have most of the same properties discussed above, except for the execution time. On an Intel i7-1065G7 processor, we executed the masked load and store instructions on both KERNEL-M pages and measured each execution time. The masked load takes an average of 92 cycles, while the execution time of the masked store is 76 cycles. The results reveal that the masked store takes roughly 16-18 cycles less time to execute than the masked load (see Table 2).

Table 1
A summary of attack primitives, exploited properties, and target processors.

Attack primitives (case studies)	Properties*	Processors (μ-arch.)
Page-table attack (§4.2, §4.3, §4.4, §4.7, §4.8, §4.9)	ℙ2, ℙ3	Intel, AMD (Zen 2, Zen 3)
TLB attack (§4.5, §4.9, §4.10, §5.1)	ℙ4	Intel, AMD (Zen, Zen+)
Permission attack (§4.6)	ℙ5	Intel

*All attacks suppress *#PFs* based on ℙ1, and ℙ6 is used as an option to improve the runtime.

ℙ6: The masked store executes faster than the masked load.

3.3. Attack primitives

We define three attack primitives based on the vulnerable properties described in §3.2, as summarized in Table 1.

Page-table attack. The page-table attack can distinguish between present and non-present pages (ℙ2) or directly leak the page-table level of the present pages at which the page-table walk terminates (ℙ3). In this paper, we reliably break KASLR on Intel CPUs based on ℙ2 (§4.2, §4.3, §4.4, §4.7, and §4.8) and AMD CPUs based on ℙ3 (§4.9).

TLB attack. The TLB attack can identify current TLB states, i.e., a TLB hit or miss (ℙ4). We use this attack primitive to break KASLR on AMD Zen and Zen+ since the page-table attack does not work reliably on these CPUs (§4.9). However, as a single masked operation is insufficient to measure a timing difference due to the noise, we combine it with a TLB eviction. In addition, with the TLB attack, we can monitor kernel activities by measuring the execution time of the masked operation on the kernel modules (§4.5). We further use this attack primitive to bypass FLARE Canella et al. (2020b), a state-of-the-art defense against currently known KASLR breaks (§5.1). The TLB attack can be used to implement covert-channels (§4.10).

Permission attack. The permission attack can identify the current page permissions (ℙ5). With this attack primitive, we can identify whether the page is readable or writable. Since the Linux kernel adopts strict kernel memory permissions where any area of the kernel with executable memory must not be writable Support strict kernel memory permissions for security (2020), in this paper, we use this attack primitive to implement a fine-grained ASLR break in the user address space (even inside an SGX enclave) (§4.6).

Note that all of our attack primitives suppress page faults caused by invalid or inaccessible memory addresses (ℙ1).

4. AVX timing side-channel attacks

This section shows how the AVX timing side-channel can be used to defeat User and Kernel ASLR on recent Intel CPUs (§4.2, §4.3, and §4.4). Furthermore, we show that our attack can be used to monitor kernel activities (§4.5) and applies to various environments, including an SGX enclave (a fine-grained ASLR break) (§4.6), other major operating systems (§4.7), and cloud computing systems (§4.8). Although this section mainly focuses on Intel processors, attacks on AMD processors are also presented at the end (§4.9). Finally, we discuss a TLB-based covert-channel implementation (§4.10).

4.1. Threat model

Attacker abilities. We assume an attacker can execute arbitrary instructions on the local machine with an unprivileged user account. The attacker can determine the version of an operating system (especially a kernel version), the name of the CPU model, and kernel functions' constant offsets. The process (sections and libraries) and the kernel (kernel text and modules) are randomly placed by the User and Kernel ASLR, and the attacker attempts to exploit memory corruption vulnerabilities (e.g., *control-flow hijacking*) through code reuse attacks (e.g., ROP).

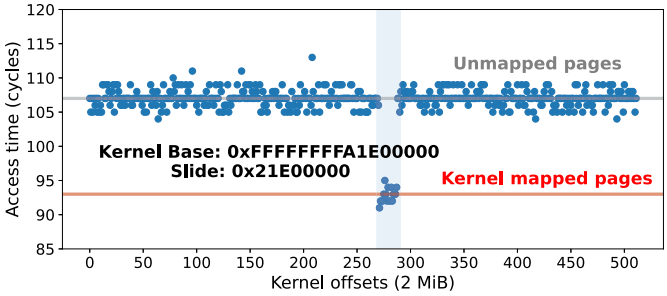


Fig. 6. Measurement results of the probing kernel address range on Linux with an Intel i9-10900 (Comet Lake). The lower plots show the execution times (average: 93 cycles) of kernel-mapped pages.

Thus, the attacker's goal is to know the addresses of the codes to find the necessary gadgets. Even if breaking KASLR could be accomplished through memory leak vulnerabilities, we do not take into account this category of software-based memory leakages.

Hardware assumptions. We assume that the processor supports AVX or AVX2 instructions (especially masked load/store instructions in Appendix C) and is protected by mitigations against the existing side-channel attacks Intel (2021b). Since AVX was introduced with the Intel Sandy Bridge (2011) and AMD Bulldozer (2011) processors, it is reasonable to assume that most modern mobile, desktop, server, and cloud systems support AVX or AVX2 by default. We further assume that the kernel is running with default boot parameters, which means that KPTI is automatically disabled or enabled according to the processor Wiki (2018).

4.2. De-randomizing the kernel base address

We first demonstrate the KASLR break on Ubuntu 20.04.3 (kernel 5.11.0-27) with a Meltdown-resistant Intel i9-10900 (Comet Lake). With KASLR, the base address of the Linux kernel is located at 512 possible offsets (see §2). In our attack, we measure the execution time of the masked load instruction at each possible offset. Algorithm 1 in the Appendix describes our detailed attack procedure. First, we determine a threshold value to distinguish between mapped and unmapped pages. We found that the execution time of the masked store on the user-mapped page with no dirty bit set (D:1) is the same as the execution time on the kernel-mapped page. Thus, we use the average execution time of the masked store instruction on the USER-M page as our threshold value. Next, based on ℙ2, we execute the masked load instruction twice for each of the 512 candidate addresses and measure the execution time of the second execution. Since valid translations of virtual addresses are cached in the TLB on Intel CPUs (Appendix A), the execution time of the kernel-mapped addresses will be faster than others. As a result, we can identify kernel-mapped addresses in the user space.

Evaluation. The execution times of accessing all possible addresses are shown in Fig. 6. From 512 plots, we can clearly distinguish between the execution times for kernel-mapped and unmapped pages. The kernel-mapped pages have a mean execution time of 93 cycles, while the unmapped pages have 107 cycles. Since the lower plots start at offset 271, we can identify the base address of the kernel (`0xffffffffa1e00000`). We rebooted Linux 10 times to verify the result and checked whether the identified base address was correct by

Table 2
An average runtime and accuracy for derandomizing kernel base and module addresses ($n = 10000$).

CPUs (setting)	Attacks	Primitives	Runtime		Accuracy
			Probing	Total	
Xeon Silver 4210 (Server)	Base	Load	96 μs	0.47 ms	99.48%
		Store	92 μs	0.46 ms	99.94%
	Modules	Load	3.43 ms	3.77 ms	99.78%
		Store	3.28 ms	3.67 ms	99.72%
i9-10900 (Desktop)	Base	Load	58 μs	0.29 ms	99.40%
		Store	55 μs	0.29 ms	100.0%
	Modules	Load	2.08 ms	2.35 ms	99.80%
		Store	1.97 ms	2.24 ms	99.85%
i7-1065G7 (Mobile)	Base	Load	0.26 ms	0.57 ms	99.29%
		Store	0.24 ms	0.55 ms	99.33%
	Modules	Load	8.42 ms	8.64 ms	99.72%
		Store	8.24 ms	8.49 ms	99.32%

Table 3
Tested Intel and AMD CPU models.

CPU Models	Types	μArch. (Gen.)	Year	KASLR break
Intel Core i7-2620M	Mobile	Sandy Bridge (2nd)†	Q1 '11	✔
Intel Core i7-3540M	Mobile	Ivy Bridge (3rd)	Q1 '13	✔
Intel Core i5-4200U	Mobile	Haswell (4th)‡	Q3 '13	✔
Intel Xeon E5-2676v3	Cloud (Amazon)	Haswell (4th)	Q3 '14	✔
Intel Core i7-5557U	Mobile	Broadwell (5th)	Q1 '15	✔
Intel Core i7-6700K	Desktop	Skylake (6th)	Q3 '15	✔
Intel Core i7-6920HQ	Mobile	Skylake (6th)	Q3 '15	✔
Intel Xeon W-2191B	Server	Skylake (server)	Q4 '17	✔
Intel Core i7-7700	Desktop	Kaby Lake (7th)	Q1 '17	✔
Intel Core i7-8559U	Mobile	Coffee Lake (8th)	Q2 '18	✔
Intel Core i9-9900	Desktop	Coffee Lake Refresh (9th)	Q2 '19	✔
Intel Core i7-1065G7	Mobile	Ice Lake (10th)	Q3 '19	✔
Intel Xeon Silver 4210	Server	Cascade Lake	Q2 '19	✔
Intel Xeon Scalable	Cloud (Google)	Cascade Lake	'19-'20	✔
Intel Core i9-10885H	Mobile	Comet Lake (10th)	Q2 '20	✔
Intel Core i9-10900	Desktop	Comet Lake (10th)	Q2 '20	✔
Intel Core i7-1165G7	Mobile	Tiger Lake (11th)	Q3 '20	✔
Intel Core i5-12400F	Desktop	Alder Lake (12th)	Q1 '22	✔
AMD EPYC 7601	Server	Zen (1st)	Q2 '17	✔
AMD Ryzen TR 1950X1	Desktop	Zen (1st)	Q3 '17	✔
AMD Ryzen 3 1300X	Desktop	Zen (1st)	Q3 '17	✔
AMD Ryzen 5 2600	Desktop	Zen+ (2nd)	Q2 '18	✔
AMD Ryzen 5 3600	Desktop	Zen 2 (3rd)	Q3 '19	✔
AMD Ryzen 5 5600X	Desktop	Zen 3 (4th)	Q4 '20	✔
AMD Ryzen 9 5950X	Desktop	Zen 3 (4th)	Q4 '20	✔
AMD EPYC 7003	Cloud (Google)	Zen 3 (4th)	Q1 '21	✔

†AVX first introduced. ‡AVX2 first supported.

confirming a `/proc/kallsyms` file. In each attempt, we always found the correct base address of the kernel without any false positives. On the i9-10900 processor, the average runtime of probing the kernel address range is 0.58 μs, while the total average runtime is 0.29 ms. Throughout the evaluation, it's important to note that the term "total runtime" means the duration taken for the execution of our Proof of Concept (PoC). In contrast, the term "probing runtime" denotes the time required for the execution of masked instructions (i.e., the sum of 512 masked load execution times in the case of the base address of the Linux kernel). The accuracy of the attack is 99.4%, on average ($n = 10000$). We also evaluated the runtime and accuracy of the masked store instruction. The average probing runtime is 0.55 μs, the total average runtime is 0.29 ms, and the accuracy is 100% (Intel i9-10900). The results of performance measurements in different settings are shown in Table 2. Notably, as described in §3, the masked store has a shorter runtime (ℙ6); however, it is not significantly different from the masked load. To assess the effectiveness of our attack, we further extended the evaluation of the attack Hyunwoo et al. (2023) on various Intel and AMD CPU models, as shown in Table 3. In our experiments, all identified base addresses of the kernel are correct, and there are no false positives. Table D.1 in Appendix provides the characterization of the masked load and store instructions on different microarchitectures.

4.3. Detecting and identifying kernel modules

In ROP-style attacks, kernel modules are rich sources of ROP gadgets because they contain a significant amount of executable code, which increases the likelihood of finding diverse sequences of instructions suitable for building ROP chains. Thus, we demonstrate how the AVX side channel can be used to identify kernel modules' addresses. In x86-64 Linux, kernel modules (or drivers) are loaded between `0xffffffffc0000000` and `0xffffffffc4000000`, with a 4 KiB alignment. Thus, by probing the address range with 4-KiB offsets (16384 possible addresses), our attack can identify the addresses of the currently loaded modules. For the attack, we first extract all mapped pages in the address range of the kernel modules by measuring timing differences (ℙ2). Then, similar to prior work Canella et al. (2019), we distinguish where a module begins and ends by taking advantage of the fact that the loaded kernel modules are separated by unmapped pages. As a result, we can identify all loaded modules and their sizes. Since Linux's `/proc/modules` file provides module information such

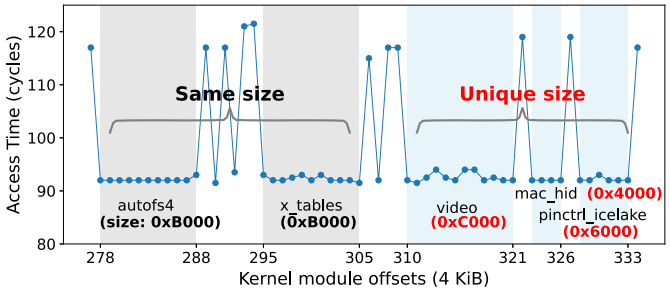


Fig. 7. Identified kernel modules' offsets. The graph shows an example of five modules, along with their names and sizes. `video`, `mac_hid` and `pinctrl_icelake` can be accurately identified by their unique sizes.

as name, size (available without privilege), and loaded address (only available with privilege), we can classify modules by correlating the detected module size with the actual size specified in the `/proc/modules` file.

Evaluation. We conducted the page-table attack (ℙ2) on Ubuntu 18.04.3 LTS (kernel version 5.4.0-81) with an Intel i7-1065G7 (Ice Lake), where the total number of loaded kernel modules is 125, of which 19 have a unique size. Fig. 7 presents an example of the five identified kernel modules along with their names and sizes. In this figure, we cannot distinguish between modules (*autofs4* and *x_tables*) that map with the same amount of pages since the identification of modules is based on the detected size. However, we can identify the three remaining modules (*video*, *mac_hid*, and *pinctrl_icelake*) that have unique sizes. As shown in Table 2, in our mobile setting (an Intel i7-1065G7), the runtime of probing the kernel modules is 8.24-8.42 ms, and the total runtime is 8.49-8.64 ms. Our attack achieved 99.32-99.72% accuracy on average ($n = 10000$). The performance results are greatly improved in a desktop setting. On an Intel i9-10900 processor, the runtime of probing is 1.97-2.08 ms, and the total runtime is 2.24-2.35 ms. The average accuracy is around 99.8%.

4.4. Breaking KASLR with KPTI enabled

In a KPTI-enabled kernel, the kernel pages are not mapped in the user space. However, to provide an entry point into the kernel space, the KPTI leaves a minimal set of kernel pages called *KPTI trampoline* in the user space, which is used to switch between the user and the kernel space. A few examples of trampoline codes include `entry_SYSCALL_64` for a syscall entry point and `swapgs_restore_regs_and_return_to_usermode` for swapping the page tables in the kernel space back to the ones in the user space. With the AVX side-channel, the attacker can identify the kernel-mapped KPTI trampoline region and leverage this to build a ROP chain that bypasses KPTI, as described in Midas (2021). Additionally, the attacker can determine the base address of the kernel using the identified KPTI trampoline since the randomization is performed by shifting the entire kernel image within a given range.

Evaluation. To evaluate our attack on Ubuntu 20.04.3 LTS (kernel 5.11.0-27) with an Intel i7-1065G7 processor, we conducted a comparative experiment as follows. We first fixed the kernel's base address at `0xffffffff81000000` via a boot parameter (*nokaslr*). We then performed the page-table attack (ℙ2) on the KPTI-enabled and -disabled kernels. In repeated experiments, we observed that the KPTI trampoline always begins at `0xffffffff81c00000`, which is the same result as the confirmed constant offset `0xc00000` beforehand. Consequently, our attack can still break KASLR–even on the KPTI-enabled kernel.

4.5. Inferring user behaviors

In general, an attacker can infer a user's activities by monitoring events associated with user interactions, such as keystrokes, mouse movements, and device connections. Recent works Schwarz et al. (2019a); Canella et al. (2019); Lipp et al. (2022) have shown that TLB-based side-channel attacks can monitor kernel activities, allowing an attacker to infer user behaviors. This section demonstrates that our TLB attack can also infer user behaviors by leveraging the same underlying mechanism as prior works Schwarz et al. (2019a); Canella et al. (2019); Lipp et al. (2022). We monitor four types of user activities: Bluetooth audio streaming, mouse movements via the touchpad, Wi-Fi transmissions, and file downloads through a USB Ethernet port. To this end, we target four different kernel modules (`bluetooth`, `psmouse`, `iwlwifi`, and `usbnet`) and keep track of their events by measuring TLB states (ℙ4). When the module is accessed, address translations are cached in the TLB. Consequently, the execution times of the masked operations vary based on whether the module is actively utilized.

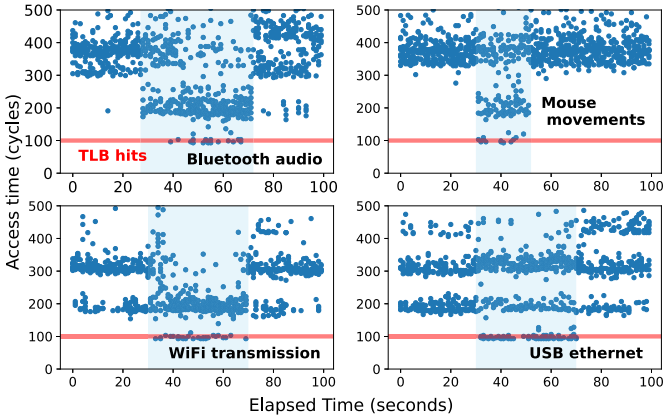


Fig. 8. Monitoring kernel activities by measuring the TLB states of kernel modules (`bluetooth`, `psmouse`, `iwlwifi`, and `usbnet`; clockwise from the top-left corner). The victim's behaviors are clearly observed in all cases.

Evaluation. We performed the attack on Ubuntu 18.04.3 LTS (Linux kernel 5.4.0-81) with an Intel i7-1056G7. In the experiment, once the addresses of the target kernel modules are identified using the kernel module detection in §4.3, a spy process repeats the TLB attack at 1-sec. intervals and lasts for up to 100 sec. Fig. 8 shows the results obtained by the spy process measuring the masked load execution time on the first ten pages of each kernel module. As shown in the graphs, we can observe that the execution times are shorter (blue area) when the kernel modules are accessed. As such, attackers can utilize our attack to infer user behaviors and proceed with further attacks. We believe that our attack will likely be extended to monitor other events or conduct application and website fingerprinting.

4.6. Fine-grained ALSR break inside an SGX enclave

Prior works Lee et al. (2017); Schwarz et al. (2019c) have demonstrated the potential for utilizing an SGX enclave to implant stealthy malware. Since SGX not only prevents enclaves from executing privileged instructions but also syscalls, SGX malware often relies on code reuse attacks, such as ROP, to execute the code of its host application. To achieve this, the malware must first derandomize the host's address space layout to find the necessary gadgets. For instance, to break ASLR, SGX-ROP Schwarz et al. (2019c) leverages an Intel TSX, where a TSX transaction suppresses faults caused by accessing invalid addresses, and its error codes can be used to identify the host process's page mappings. Here, using the AVX side-channel, we demonstrate how to implement a fine-grained ASLR break that works even inside an SGX enclave without using TSX.

Identifying the process's code section. In x86-64 Linux, the ASLR entropy for the process's address space is 28 bits. For example, the process's code text is located within `0x55`**`xxxxxxx`**`000`, and the libraries are loaded within `0x7f`**`xxxxxxx`**`000`. To explore such address spaces,

Table 4
Comparison of KASLR breaks in major operating systems. Our attack successfully breaks KASLR on most major operating systems (except for KPTI-enabled macOS).

Operating system	Processor	Entropy	Time	KASLR break	
				w/o KPTI	w/ KPTI*
Linux (Ubuntu 20.04.04)	i9-10900	9 bits	0.29 ms	✔	✔
Windows 10 (21H1)	i9-9900	18 bits	60 ms	✔	✔
macOS 10.13.1-2	i7-6920HQ	8 bits	0.5 ms	✔	✘

* KPTI impl. is called PTI on Linux, KVAS on Windows, and Double Map on macOS.

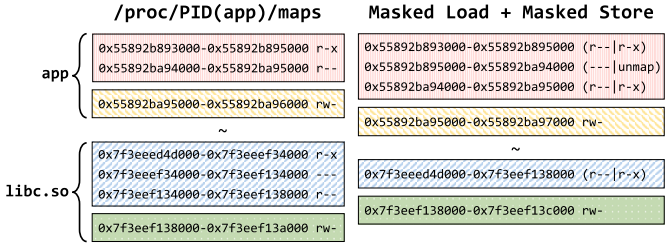


Fig. 9. The process's identified mapped memory regions and their access permissions. The left is the output of the `maps` file, and the right is the result confirmed by our attack. For simplicity, we depict an `libc.so` library only.

we linearly probe the entire virtual address range with a 4-KiB alignment by measuring the execution time of the masked load/store (ℙ2). On Ubuntu 18.04.3 (kernel 5.4.0-81) with an Intel i7-1065G7, our attack successfully identifies the base address of the process's code section inside an SGX enclave. In our unoptimized proof-of-concept implementation, with SGX2 that supports a high-precision timer (`RDTSC` and `RDTSCP`), our attack takes on average 51 sec. (masked load) and 44 sec. (store).

Identifying loaded libraries. To identify the process's loaded libraries, we use a fine-grained methodology based on ℙ5. On Ubuntu 18.04.3, we observed that the loaded libraries (e.g., `libc.so`) consist of consecutive sections and the sections' permissions are in the order of `r-x`, `---`, `r--`, and `rw-`. With this, we used sections' sizes as signatures for detecting libraries. We probed the address space twice by combining the masked load and store to reduce noise and distinguish between readable and writable pages. We first probed the address space using the masked load and filtered out the `none` pages. We then probed again using the masked store to identify the `read-write` pages. Fig. 9 shows the results. Our attack was unable to differentiate between `read-only` and `read-exec`, but it did detect additional pages (`0x55892ba96000` and `0x7f3eef13b000`) that had never been identified with a `/proc/PID/maps` file. We investigated page tables using a custom kernel module and confirmed that all the detected permissions were correct. The average runtime is 95 sec. (51 sec. for the masked load and 44 sec. for the store). The runtime can be significantly improved on the desktop processor.

4.7. Attacks on other popular operating systems

To verify the feasibility of our attack on popular operating systems, we evaluated our attack on both Windows and macOS. On Windows, KPTI is enabled or disabled by default according to the processor. Thus, for Windows, we considered two different types of processors: the Intel i7-6600U (Meltdown-vulnerable) and the i9-9900 (Meltdown-resistant), to evaluate KPTI-enabled or -disabled kernels. In contrast, for macOS, we used one processor, the i7-6920HQ (Meltdown-vulnerable), because KPTI is adjusted according to macOS versions. The results are presented in Table 4.

Attacks on Windows 10. In Windows 10 (ver. 21H1), the kernel and drivers are located between `0xfffff8`**`00000`**`00000-0xfffff8`**`80000`**`00000` with a 2 MiB boundary, which leads to 262144 possible offsets (i.e., 18 bits of entropy) Maisuradze and Rossow (2018). The kernel's entry point is randomized within this address range and can begin at any 4-KiB boundary. We probed the kernel address space on an Intel i9-9900 processor. As a result, we found the kernel address region–which is allocated in five consecutive 2-MiB pages–within 60 ms, on average. In our attack, we were only able to find the base address of the large region containing the kernel image. However, this still derandomizes 18 bits of KASLR entropy and can be used in combination with our TLB attack (ℙ4) to break the remaining 9 bits of entropy.

Additionally, we further conducted our attack on KVAS-enabled Windows. In Windows 10 (ver. 1709), the KVAS code (e.g., `KiSystemCall64Shadow`) is part of the kernel (as in Linux), and the offset from the kernel base address is `0x298000`. On an Intel i7-6600U (Skylake), we probed the kernel address space with a 4-KiB alignment. Consequently, we found the KVAS region consisting of three consecutive 4-KiB pages in 8 sec. with 100% accuracy. After that, we found the kernel base address by subtracting the KVAS offset from the identified address.

Attacks on macOS. For macOS with the Intel i7-6920HQ (Skylake), we considered two macOS versions: 10.13.1 (no KPTI) and 10.13.2 (KPTI first implemented). In macOS, the kernel is mapped between `0xffffff8000000000` and `0xffff ff80`**`200`**`00000` with a 2-MiB alignment, resulting in 256 possible offsets. On macOS 10.13.1, our attack successfully identified the kernel's base address within a millisecond with 100% accuracy. Our attack still found a Double Map region on macOS 10.13.2, where Double Map (the macOS implementation of KPTI) was first implemented. Our initial reverse-engineering of macOS's KPTI implementation revealed that the Double Map is not part of the kernel image and is randomly allocated at boot time Code snippets of XNU's Double Map initialization. This means we cannot calculate other kernel parts using a technique that relies on a fixed offset, like attacks on KVAS-enabled Windows.

4.8. Breaking KASLR in cloud computing systems

To demonstrate the feasibility of our attack in a cloud computing environment, we conducted the attack on global cloud services: Amazon EC2, Google GCE, and Microsoft Azure. The experiment setups are summarized in Table 5.

Evaluation. Like with KASLR breaks before, we mounted our attack by probing all possible kernel addresses in the range of randomization. We successfully identified the kernel's base address and currently loaded modules in all our experiments. On Amazon EC2, since the processor is vulnerable to Meltdown, we found the KPTI trampoline at offset `0xe00000`, from which we were able to calculate the base address of the kernel. The execution time is 0.03 ms for identifying the kernel base and 1.14 ms for the kernel modules. On Google GCE, we identified the kernel base address in 0.08 ms and the kernel modules in 2.7 ms. We derandomized 18 bits of KASLR entropy on Microsoft Azure within 2.06 sec. ($n = 1000$).

4.9. KASLR breaks on AMD processors

We evaluate our attack across four different AMD microarchitectures: Zen, Zen+, Zen 2, and Zen 3. From the experiments, we observed

Table 5
KASLR breaks in cloud computing systems.

Cloud services	OSes	CPUs	Runtime
Amazon EC2	Ubuntu 20.04.3 (5.11.0-1020-aws)	Intel Xeon E5-2676 V3 (Haswell)	0.03 ms
Google GCE	Ubuntu 20.04.1 (5.13.0)	Intel Xeon (Cascade Lake)	2.7 ms
Microsoft Azure*	Windows 10 (21H2)	Intel Xeon Platinum 8171M (Skylake)	2.06 sec.

*The CPU is vulnerable to Meltdown, but KPTI (KVAS in Windows) was not enabled by default.

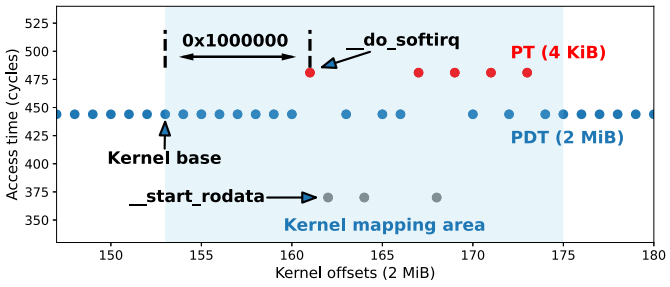


Fig. 10. De-randomizing kernel base address on an AMD Ryzen 5 5600X processor (Zen 3). The blue area shows the kernel mapping area. The first high peak is the location of the `_do_softirq` function, and the first low peak is the location of the `_start_rodata` section.

that the MMU behaves differently depending on the microarchitectures of AMD CPUs. On Zen and Zen+, the kernel-mapped pages are cached in the TLB, but not on Zen 2 and Zen 3 (Table A.1 in the Appendix). Therefore, we used a TLB-based attack (ℙ4) for Zen and Zen+ and a page-table attack (ℙ3) for Zen 2 and Zen 3.

Page-table Attack (on Zen 2 and Zen 3). Our KASLR breaks on AMD Zen 2 and Zen 3 CPUs are based on two observations. First, when the mask load instruction is executed with invalid or inaccessible kernel addresses, AMD processors always perform page table walks. Once the page table walk is triggered, it can leak information about the page table's level on which the page table walk stops (ℙ3). Second, although the kernel is mapped to huge pages (2 MiB), there are 4-KiB pages within the kernel area Lipp et al. (2022). In Ubuntu 20.04.1 (5.13.0-52), we observed five 4K-sized pages aligned to a 2-MiB offset within the kernel area. Thus, if we can locate any of these 4K-sized pages, we can use them to determine the kernel's base address from those pages.

We performed the attack on Ubuntu 20.04.1 (5.13.0-52) with an AMD Ryzen 5 5600X processor. We probed the kernel address range with 512 offsets and measured the execution time of the masked load instruction. Fig. 10 presents the measurement results. The blue area indicates the kernel-mapped address space (the kernel begins at `0xffffffff932`00000). We observed three execution times: 370, 444, and 481 cycles. These timings depend on the level of the page tables the kernel page is mapping. The first high peak (481 cycles) is the location of the `_do_softirq` function, which is mapped to a 4-KiB page and located at 0x100000 from the kernel base. With 10 Linux boots and measurements, we confirmed that the `_do_softirq` function always appears at the first peak with a fixed offset. Therefore, we can reliably calculate the kernel base address from this offset in an average runtime of 2.9 ms with 100% accuracy ($n = 1000$). Other peaks also appeared at fixed offsets (e.g., a `_start_rodata` section with the first low peak at 370 cycles). Thus, we can use these offsets to break KASLR.

TLB attack (on Zen and Zen+). We measured the status of the TLB based on ℙ4. We performed the attack on Ubuntu 20.04.1 (5.15.0-41) with an AMD Ryzen 3 1300 (Zen). We measured the execution time of the masked load instruction over the kernel address range with TLB evictions between each measurement. Although the reduced update interval of the timestamp counter on Zen Lipp et al. (2020) and system noise make our attack challenging, we can reliably break KALSR by measuring over multiple executions.

The evaluation results across various AMD CPUs are depicted in Table 3. Additionally, we verified that the KASLR break works in a virtualized environment by mounting the attack on Google GCE. On the t2d-standard-1 instance with an AMD Zen 3-based EPYC 7B13 processor, we successfully identified the kernel base address within an average execution time of 2.8 ms. This paper mainly focused on Zen microarchitectures; however, our attack will likely be applied to other AMD microarchitectures that support AVX.

4.10. Covert-channel attacks

Similar to prior works Gras et al. (2018); Lipp et al. (2022), our attacks can establish a TLB-based covert-channel between SMT threads. To implement the covert-channel, we can exploit the fact that the address translations of kernel-mapped pages are cached in the TLB. For example, to transmit a 1-bit, the sender evicts the TLB and accesses certain kernel pages that the receiver knows using the masked load instruction. To transmit a 0-bit, the sender does nothing (idle). To receive a bit, the receiver accesses the same kernel pages and decodes the transmitted bit by measuring the execution time of the masked operations. Since the TLB is only shared between SMT threads, our covert-channel does not work across cores.

Similarly, our covert-channel can be used as a transmitter for Spectre attacks Kocher et al. (2019). If the kernel has Spectre-type gadgets where the processor can speculatively access kernel pages based on the user input and the secret, our covert-channel can alternate the existing covert-channels such as cache timing Yarom and Falkner (2014), SIMD utilization Schwarz et al. (2019b), and port contention Bhattacharyya et al. (2019); Aldaya et al. (2019). As described in Lipp et al. (2022), our covert-channel is advantageous because it works without any shared memory between the kernel and user space. We leave the evaluation of the covert-channels for future work.

5. Countermeasures

5.1. Software-based mitigations

On the AMD processor, we utilized the page-table attack (ℙ2) to distinguish the 4 KiB page, and then we could calculate the KASLR address by subtracting a relative offset from the page. We identified five 4 KiB pages from the entire kernel address. This attack primitive can be mitigated by separately randomizing the 4 KiB entry point pages, such as the `_do_softirq` function, from the rest of the kernel.

KPTI significantly improves kernel hardening against attempts to bypass KASLR. However, as described in §4, a few pages for the KPTI trampoline are still mapped into the user space, thereby allowing an attacker to calculate other kernel parts from a fixed trampoline offset in Linux and Windows. As we observed in macOS (since 10.13.2), if the KPTI trampoline is not part of the kernel image and is randomly allocated outside the kernel image at boot time, our attack can be mitigated successfully. We leave a more comprehensive exploration of these ideas for future work.

Function granular KASLR (FGKASLR) Accardi (2020) prevents de-randomizing the kernel address space by implementing finer-grained address space randomization. With FGKASLR, individual kernel functions are reordered so that even if the kernel's base address is revealed, the attacker cannot identify the location of specific kernel functions because relative addresses are different. However, even with FGKASLR

Table 6
Comparison with existing attacks on KASLR.

Attacks	Features exploited (source of side-effect)	Runtime		Accuracy	Requirements
		Kernel base	Kernel modules		
Double Page Fault Hund et al. (2013)	Page fault handling in MMU (TLB)	17.27 - 72.93 s	N/A	94.92 - 99.69%	Noise filtering
DrK Jang et al. (2016)	Page fault handling with Intel TSX (TLB)	5 ms	0.16 - 0.21 s	100%	Intel TSX enabled
Evict+Prefetch Gruss et al. (2016)	Software prefetch (TLB)	N/A	500 s	N/A	Cache eviction
BTB-collision Evtyushkin et al. (2016)	Branch prediction (BTB)	60 ms	N/A	N/A	Knowledge of BTB hash function
Data Bounce Schwarz et al. (2019a); Canella et al. (2019)	Store-to-load forwarding in transient execution (SB)	42 - 72 μs	1.33 - 1.71 s	96 - 100%	Cache state monitoring (Flush+Reload)
TagBleed Koschel et al. (2020)	Tagged TLBs (TLB)	1 s	N/A	94%	Knowledge of TLB addressing
EchoLoad Canella et al. (2020a)	Incomplete Meltdown fix (ROB)	29 μs - 133 μs	N/A	99 - 100%	Cache state monitoring (Flush+Reload)
PLATYPUS Lipp et al. (2021)	Power consumption difference (MMU)	20 s	N/A	100%	Access right to RAPL interface
FlushConflict Weber et al. (2021)	Cache-line conflict of MOVNT (MMU)	5 - 349 ms	N/A	92 - 99%	Cache state monitoring (Flush+Reload)
AMD Prefetch Lipp et al. (2022)	Software prefetch (TLB)	0.15 s	N/A	100%	RAPL interface or TLB eviction
This work	**AVX masked load/store (TLB)**	**55 - 71 μs**	**2.24 - 2.78 ms**	**99.25 - 100%**	**AVX or AVX2 extension**

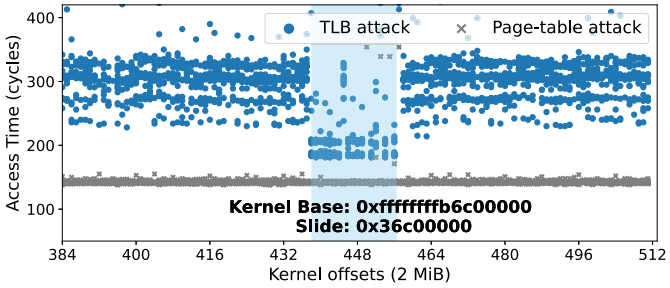


Fig. 11. Results of TLB ($\mathbb{P}4$) and page-table ($\mathbb{P}2$) attacks on the FLARE-enabled kernel. In the TLB attack (blue plots), the fast access times that reveal the kernel-mapped pages are clearly evident.

enabled, our attack can still break the fine-grained KASLR by leveraging the TLB state template attacks, as described in Lipp et al. (2022). Additionally, certain regions in the kernel never get randomized (e.g., `ksymtab` symbols), from which the attacker can obtain the necessary gadgets for a code reuse attack Learning Linux Kernel Exploitation. On Linux with FGKASLR enabled (5.16.0-rc3), we verified that our attack can still find these regions from the beginning of the kernel mapping area.

We further verified whether our attack could bypass FLARE Canella et al. (2020a), a state-of-the-art defense against currently known KASLR breaks Canella et al. (2020a); Gruss et al. (2016); Schwarz et al. (2019a); Canella et al. (2019); Hund et al. (2013); Jang et al. (2016). With FLARE, all unmapped virtual addresses in the randomization range are mapped to dummy physical pages. Thus, the attacks relying on page mappings Lipp et al. (2022) can no longer be used to derandomize the kernel space. On the FLARE-enabled kernel Canella et al. (2020b), we mounted two types of attacks: TLB ($\mathbb{P}4$) and page-table ($\mathbb{P}2$) attacks. Fig. 11 presents the result of our attacks on Ubuntu 20.04.4 (5.11.0-27) with an Intel i7-1065G7 (Ice Lake). We could clearly identify the fast access times that revealed the mapped kernel regions in the TLB attack from the result. While dummy mappings mitigate attacks based on the page table level, they do not prevent our TLB-based KASLR break.

Our attack primitive, AVX, is quite similar to AMD prefetch Lipp et al. (2022) in terms of measuring the execution time of an instruction. However, software prefetch instructions can be easily mitigated through MSR configurations. To the best of our knowledge, there is no way to disable AVX via MSR. We highlight that a stronger fine-grained KASLR, such as address space re-randomization, is needed Giuffrida et al. (2012); Williams-King et al. (2016) to prevent our attack.

5.2. Hardware-based mitigations

Since our attack is based on cached translations in TLBs, a technique that isolates user processes from the TLB sets associated with the kernel space can mitigate our attack. However, this mitigation is not practically possible since the partitioned TLBs do not fully support continuous virtual addresses, and it requires expensive hardware changes Gras et al. (2018); Koschel et al. (2020). Additionally, it is possible to replace the masked load/store instructions with `NOP`s only when the mask bits are all set to zero. To estimate the impact of this mitigation, we assessed the prevalence of masked load/store instructions. On Ubuntu 20.04.3 LTS with the default installation, we found only 6 (compilation-related binaries) out of 4104 executables that contain the masked load/store instructions. Therefore, we believe that the solution of restricting or replacing masked operations has little impact on the system. Since the QEMU Bellard (2005) emulator and Gem5 Binkert et al. (2011) simulator do not yet support AVX masked load/store, we leave the detailed performance evaluation for future work.

6. Related works and comparison

In this section, we compare our attack with previous attacks in terms of exploited features, sources of measurable side effects, performance, and the requirements for a successful attack.

6.1. Microarchitectural attacks on KASLR

Table 6 shows the comparison results for each attack. The runtimes presented in Table 6 reflect the timings mentioned in each respective paper, as their code has not been made publicly available. Consequently, the time units might not be uniform due to the lack of explicit clarification regarding whether the measurements encompass total runtime or probing time. Nevertheless, all attacks, excluding Hund et al. (2013), exhibit a practical level of speed. Notably, our attack ranks among the fastest, comparable to state-of-the-art techniques.

Double Page Fault Hund et al. (2013) introduced the first microarchitectural attack on KASLR break by exploiting TLB states. *DrK*

Jang et al. (2016) greatly improved Hund et al.'s attack Hund et al. (2013) by using an Intel TSX. *Evict+Prefetch* Gruss et al. (2016) and *AMD Prefetch* Lipp et al. (2022) also exploit the timing difference in the TLB; however, no fault suppressions are required because software prefetch does not issue any faults.

Although *BTB-collision* Evtyushkin et al. (2016) and *TagBleed* Koschel et al. (2020) also do not require fault suppressions, they do necessitate additional reverse engineering for BTB or TLB addressing.

Data Bounce Schwarz et al. (2019a) and *Fallout* Canella et al. (2019) exploit the fact that under transient execution, a store-to-load forwarding with an inaccessible kernel address is performed when a full or partial address matches. The attacks were mitigated as a part of the microarchitectural data sampling (MDS) patch Intel (2019).

EchoLoad Canella et al. (2020a) exploited an incomplete hardware fix for Meltdown. *FlushConflict* Weber et al. (2021) exploited a cache line conflict caused by the `MOVNT` instruction. These attacks leverage transient execution and Flush+Reload Yarom and Falkner (2014) to indirectly determine whether the result of kernel memory access is affected by the page mapping state. Although *EchoLoad* is known to be the fastest and most reliable attack, it no longer works in Intel's 10th generation microarchitectures (Ice Lake and Comet Lake).

PLATYPUS Lipp et al. (2021) introduced the first microarchitectural KASLR break that solely uses power consumption differences. Recently, most Linux distros restricted the RAPL to "*root*" only. Similarly, *AMD Prefetch* Lipp et al. (2022) exploits RAPL to break KASLR on the AMD Zen microarchitecture. It further showed that a TLB-based prefetching attack (called *TLB-Evict+Prefetch*) can also be mounted to break KASLR on AMD Zen 2. But for a reliable attack, a TLB eviction is required before issuing the prefetch instruction. Our attack is most similar to software prefetch attacks Gruss et al. (2016); Lipp et al. (2022) in terms of the exploited feature and fault suppression. However, our KASLR break reliably works on AMD Zen 2 and Zen 3 without relying on TLB eviction.

6.2. AVX side-channel attacks

Schwarz et al. Schwarz et al. (2019b) introduced the first AVX-based covert-channel, which is based on timing differences in the AVX2 power saving feature. Ragab et al. Ragab et al. (2021) discovered that `VMASKMOV` instructions with all-zero masks and invalid addresses issue a machine clear. Once the machine clear is triggered, the processor flushes the entire processor pipeline and restarts execution from the last retired instruction. Although this side-effect incurs a transient execution, it cannot be used due to the speculation window being too short. Weber et al. Weber et al. (2021) discovered a timing side-channel that consists of `VDMADD132PD` and `FISTP` instructions, as revealed by their side-channel fuzzing framework. In this paper, we exploited the fault resistance and timing difference properties of the AVX masked operations.

7. Conclusion

This paper introduced a novel AVX timing side-channel attack that can defeat User and Kernel ASLR. We demonstrated USER and Kernel ASLR breaks on popular operating systems, cloud computing systems, and even within an SGX enclave. We also showed that our attack can effectively infer user behaviors by monitoring kernel activities. Our attack is very fast, reliable, and works on the vast majority of modern processors. We highlight that stronger isolation or re-randomization should be implemented to mitigate our presented attack successfully.

CRediT authorship contribution statement

Suryeon Kim: Conceptualization, Methodology, Software, Writing – original draft. **Seungwon Shin:** Formal analysis, Writing – review & editing. **Hyunwoo Choi:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: co-author currently editorial board member of elsevier computer and security.

Data availability

I have shared the link of proof-of-concept code of our attack in the manuscript.

Appendix A. Results of dTLB misses and page table walks for an inaccessible kernel-mapped address

Table A.1 presents the number of both dTLB misses and page table walks when executing the masked load instruction 1000 times with a user-inaccessible kernel-mapped address. On the Intel processors, we observe that there is only one TLB miss on the first access, implying that the inaccessible address is cached in the TLB. On the AMD processors, it shows different behaviors according to the microarchitectures. On Zen 3, the MMU triggers two page table walks per single dTLB miss, whereas on Zen 2, each memory access results in two triggers for the dTLB miss and page table walks. In either case, we confirm that the inaccessible address is not stored in the TLB. Interestingly, on Zen and Zen+, the measurement results are the same as on Intel. On AMD CPUs, we demonstrated KASLR breaks based on the level of page tables ($\mathbb{P}3$ for Zen 2 and Zen 3) and the status of the TLB ($\mathbb{P}4$ for Zen and Zen+).

Appendix B. Algorithm for probing the base address of the Linux kernel

Algorithm 1 depicts an algorithm for probing the kernel address range using a masked load instruction. Since the Linux kernel is mapped with a 2-MiB alignment, we measured the execution time of the masked load with the 512 possible kernel addresses. In the page-table attack ($\mathbb{P}2$ and $\mathbb{P}3$), we executed the masked load instruction twice (for caching the translation of the first access) and took the execution time of the second execution as a timing value. In the TLB-based attack ($\mathbb{P}4$), we evicted the TLB before the measurement and executed the masked load instruction only once. Notably, on Intel CPUs, our algorithm can reliably measure timing differences without serializing operations, whereas on AMD CPUs, a serializing instruction such as `MFENCE` or `LFENCE` is required to measure the timings.

Algorithm 1: De-randomizing the base address of the Linux kernel using the AVX timing side-channel attack.

```
   /* Identifying kernel-mapped pages */
1  Procedure ProbeAddr()
2      start ← 0xFFFFFFFF80000000;
3      end ← 0xFFFFFFFFC0000000;
4      mask ← {0, 0, 0, 0, 0, 0, 0, 0}
5      while start ≤ end do
           // Repeat 2 times for caching valid addresses
6          for i = 0; i ≤ 2; i++ do
7              if AccessTime(start, mask) < Threshold then
8                  Kernel-mapped page.
9              else
10                 Unmapped page.
11         start ← start + 0x200000; // 2 MiB boundary
   /* Measure access time for candidate addresses */
12 Procedure AccessTime(addr, mask)
13     mfence;
14     past ← rdtscp();
15     vmaskmovp (addr), mask, dest;
16     now ← rdtscp();
17     return now - past;
```

Table A.1
Number of dTLB misses and page table walks of the masked load instruction for an inaccessible kernel-mapped address ($n = 1000$). On AMD CPUs, we used `ls_l1_d_tlb_miss.all` and `ls_tablewalker.dside` to observe dTLB misses and page table walks. On Intel CPUs, we used `dtlb_load_misses.walk_completed` for both dTLB misses and page table walks.

Processors	dTLB misses	page table walks
Intel Core i7-2620M (Sandy Bridge)	1	1
Intel Core i7-6600U (Skylake)	1	1
Intel Core i9-9900 (Coffee Lake)	1	1
Intel Core i5-12400F (Alder Lake)	1	1
AMD Ryzen 3 1300X (Zen)	1	1
AMD Ryzen 5 2600 (Zen+)	1	1
AMD Ryzen 5 3600 (Zen 2)	2000	2000
AMD Ryzen 5 5600X (Zen 3)	1000	2000

- Performance counters used for measurements:
`dtlb_load_misses.walk_completed` (dTLB miss and page table walks on Intel)
`ls_l1_d_tlb_miss.all` (dTLB miss on AMD)
`ls_tablewalker.dside` (page table walks on AMD)

Table D.1
Characterization of AVX masked operations on different μArchs.

Microarchitecture (gen.)	AVX (`vmaskmov`)	AVX2 (`vpmaskmov`)	Load	Store
Intel Sandy Bridge (2nd)	✔	N/A	✔	✘
Intel Ivy Bridge (3rd)	✔	N/A	✔	✘
Intel Haswell (4th)	✔	✔	✔	✔
Intel Broadwell (5th)	✔	✔	✔	✔
Intel Skylake (6th)	✔	✔	✔	✔
Intel Kaby Lake (7th)	✔	✔	✔	✔
Intel Coffee Lake (8th)	✔	✔	✔	✔
Intel Coffee Lake Refresh (9th)	✔	✔	✔	✔
Intel Cascade Lake (10th)	✔	✔	✔	✔
Intel Tiger Lake (11th)	✔	✔	✔	✔
Intel Alder Lake (12th)	✔	✔	✔	✔
AMD Zen (1st)	✔	✔	✔	✘
AMD Zen+ (2nd)	✔	✔	✔	✘
AMD Zen 2 (3rd)	✔	✔	✔	✘
AMD Zen 3 (4th)	✔	✔	✔	✘

```
/* AVX masked load */
vxorpd %ymm0, %ymm0, %ymm0
vmaskmovpd (%rdi), %ymm0, %ymm0

/* AVX masked store */
vxorpd %ymm0, %ymm0, %ymm0
vmaskmovpd %ymm0, %ymm0, (%rdi)

/* AVX2 masked load */
vpxor %ymm0, %ymm0, %ymm0
vpmaskmovd (%rdi), %ymm0, %ymm0

/* AVX2 masked store */
vpxor %ymm0, %ymm0, %ymm0
vpmaskmovd %ymm0, %ymm0, (%rdi)
```

Listing C.1: AVX/AVX2 masked load/store instructions used in this study.

Appendix C. Masked load and store instructions

The masked load/store instructions for AVX and AVX2 are presented in Listing C.1. The `%ymm0` register is a mask value of "*zero*," and the `%rdi` is the kernel address being accessed by the masked load and store instructions.

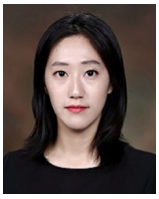
Appendix D. Characterization of AVX masked operations

As demonstrated in §4, the AVX timing side-channel attack uses either a masked load or a store instruction to identify the status of page mappings, page permissions, or TLB. In our attack, we observed that the masked load and store instructions behave differently depending on the microarchitectures. Table D.1 presents the characterization of the masked load and store instructions on various microarchitectures. As shown in the table, our attack has several distinct characteristics. First, on Intel Sandy Bridge and Ivy Bridge microarchitectures (AVX2 is not supported), our attack can only use the masked load instruction to distinguish the status of page mappings. Second, on most of the Intel microarchitectures (from Haswell to Alder Lake) that support both AVX (`VMASKMOV`) and AVX2 (`VPMASKMOV`), our attack can use both masked load and store instructions to identify the status and page mappings. Finally, on the AMD microarchitectures (from Zen to Zen 3), our attack can only work with the masked load instruction. Although the masked load and store instructions behave differently depending on the microarchitectures, our attack can reliably defeat KASLR on most Intel and AMD processors.

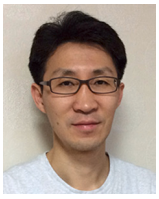
References

Accardi, K.C., 2020. Function granular KASLR. https://lwn.net/Articles/824307/.
Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N., 2019. Port contention for fun and profit. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 870–887.
AMD, 2021. AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions. AMD64 Technology.
Barr, T.W., Cox, A.L., Rixner, S., 2010. Translation caching: skip, don't walk (the page table). Comput. Archit. News 38 (3), 48–59.
Bellard, F., 2005. QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track. California, USA, vol. 41, p. 46.
Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A., 2019. Smotherspectre: exploiting speculative execution through port

contention. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 785–800.
Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al., 2011. The gem5 simulator. Comput. Archit. News 39 (2), 1–7.
Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., et al., 2019. Fallout: leaking data on meltdown-resistant CPUs. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 769–784.
Canella, C., Schwarz, M., Haubenwallner, M., Schwarzl, M., Gruss, D., 2020a. KASLR: break it, fix it, repeat. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, pp. 481–493.
Canella, C., Schwarz, M., Haubenwallner, M., Schwarzl, M., Gruss, D., 2020b. PoC implementation for FLARE. https://github.com/IAIK/FLARE.
Code snippets of XNU's Double Map initialization. https://github.com/apple/darwin-xnu/blob/main/osfmk/i386/i386_init.c.
Corbet, J., 2017. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/.
Deilmann, M., et al., 2012. A Guide to Vectorization with Intel® C++ Compilers. Intel Corporation, pp. 20–21.
Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N., 2016. Jump over ASLR: attacking branch predictors to bypass ASLR. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp. 1–13.
Giuffrida, C., Kuijsten, A., Tanenbaum, A.S., 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In: 21st {USENIX} Security Symposium ({USENIX} Security 12), pp. 475–490.
Gras, B., Razavi, K., Bos, H., Giuffrida, C., 2018. Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 955–972.
Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S., 2016. Prefetch side-channel attacks: bypassing SMAP and kernel ASLR. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 368–379.
Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S., 2017. KASLR is dead: long live KASLR. In: International Symposium on Engineering Secure Software and Systems. Springer, pp. 161–176.
Hund, R., Willems, C., Holz, T., 2013. Practical timing side channel attacks against kernel space ASLR. In: 2013 IEEE Symposium on Security and Privacy. IEEE, pp. 191–205.
Hyunwoo, C., Suryeon, K., Seungwon, S., 2023. AVX timing side-channel attacks against address space layout randomization. In: Proceedings of the 60th Annual Design Automation Conference 2023.
Intel, 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Intel Corporation.
Intel, 2019. Microarchitectural data sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html.
Intel, 2021a. Intel® 64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation.
Intel, 2021b. Speculative execution side channel mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html.
Ionescu, A., 2018. Apple Double Map, https://twitter.com/aionescu/status/948609809540046849.
Jang, Y., Lee, S., Kim, T., 2016. Breaking kernel address space layout randomization with Intel TSX. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 380–392.
Kagan, M., 1997. MMX™ Microarchitecture of Pentium® Processors with MMX Technology and Pentium® II Microprocessors.
Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al., 2019. Spectre attacks: exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1–19.
Koschel, J., Giuffrida, C., Bos, H., Razavi, K., 2020. TagBleed: breaking KASLR on the isolated kernel address space using tagged TLBs. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 309–321.
Learning Linux kernel exploitation - part 3. https://lkmidas.github.io/posts/20210205-linux-kernel-pwn-part-3/.
Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B., 2017. Hacking in darkness: return-oriented programming against secure enclaves. In: 26th {USENIX} Security Symposium ({USENIX} Security 17), pp. 523–539.
Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al., 2018. Meltdown: reading kernel memory from user space. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). USENIX Association, pp. 973–990.
Lipp, M., Hadžić, V., Schwarz, M., Perais, A., Maurice, C., Gruss, D., 2020. Take a way: exploring the security implications of amd's cache way predictors. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, pp. 813–825.
Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., Gruss, D., 2021. PLATYPUS: software-based power side-channel attacks on x86. In: IEEE Symposium on Security and Privacy (SP).
Lipp, M., Gruss, D., Schwarz, M., 2022. AMD prefetch attacks through power and time. In: USENIX Security Symposium.
Maisuradze, G., Rossow, C., 2018. Speculose: analyzing the security implications of speculative execution in cpus. arXiv preprint arXiv:1801.04084.
Midas, 2021. Learning Linux kernel exploitation - part 2. https://lkmidas.github.io/posts/20210128-linux-kernel-pwn-part-2/.
Ragab, H., Barberis, E., Bos, H., Giuffrida, C., 2021. Rage against the machine clear: a systematic analysis of machine clears and their implications for transient execution attacks. In: 30th {USENIX} Security Symposium ({USENIX} Security 21). USENIX Association, pp. 1451–1468.
Schwarz, M., Canella, C., Giner, L., Gruss, D., 2019a. Store-to-leak forwarding: leaking data on meltdown-resistant CPUs (updated and extended version). arXiv preprint arXiv:1905.05725.
Schwarz, M., Schwarzl, M., Lipp, M., Masters, J., Gruss, D., 2019b. NetSpectre: read arbitrary memory over network. In: European Symposium on Research in Computer Security. Springer, pp. 279–299.
Schwarz, M., Weiser, S., Gruss, D., 2019c. Practical Enclave Malware with Intel SGX, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 177–196.
Shacham, H., 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552–561.
Support strict kernel memory permissions for security. https://lwn.net/Articles/812633/.
swiat, 2018. KVA shadow: mitigating meltdown on windows. https://msrc-blog.microsoft.com/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.
TLB, 2022. DR: enhancing TLB-based attacks with TLB desynchronized reverse engineering. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/tatar.
Weber, D., Ibrahim, A., Nemati, H., Schwarz, M., Rossow, C., 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. 30th USENIX Security Symposium (USENIX Security, vol. 21. USENIX Association, pp. 1415–1432.
Wiki, U., 2018. MitigationControls. https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/SpectreAndMeltdown/MitigationControls.
Williams-King, D., Gobieski, G., Williams-King, K., Blake, J.P., Yuan, X., Colp, P., Zheng, M., Kemerlis, V.P., Yang, J., Aiello, W., 2016. Shuffler: fast and deployable continuous code re-randomization. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 367–382.
Yarom, Y., Falkner, K., 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14), pp. 719–732.



Suryeon Kim is currently pursuing her Ph.D. degree in School of Computing at KAIST, from March 2022. She received her B.S. degree in Information Science at Korea Military Academy, Republic of Korea, in 2011, and her M.S. degree in Information Security at KAIST, Republic of Korea, in 2017. Her research interests include computer architecture, hardware security.



Seungwon Shin is an associate professor in the School of Electrical Engineering at KAIST. He received his Ph.D. degree in Computer Engineering from the Electrical and Computer Engineering Department, Texas A&M University, and his M.S. degree and B.S. degree from KAIST, both in Electrical and Computer Engineering. He is currently a corporate vice president at Samsung Electronics, leading the security team in the IT & Mobile Communications Devision. His research interests span the areas of Software-defined networking security, IoT security, Botnet analysis/detection, DarkWeb analysis and cyber threat intelligence.

Hyunwoo Choi is currently a senior researcher at National Security Research Institute. He received his Ph.D. degree in Information Security at KAIST, Republic of Korea, in 2017. His research interest lies in the areas of system and hardware security, with a focus on software vulnerabilities and microarchitectural attacks.