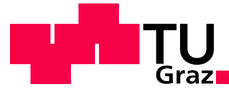


Thomas Schuster, BSc

Speculative Dereferencing of Registers

MASTER'S THESIS

to achieve the university degree of

Dipl. Ing.

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Martin Schwarzl, Dipl.-Ing. BSc
Daniel Gruss, Ass.Prof. Priv.-Doz. Dipl.-Ing. Dr.techn. BSc

Institute of Applied Information Processing and Communications

Graz, March, 2021

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Abstract

Many modern operating systems' kernels hide information about virtual to physical mapping information from user programs. This is due to security reasons, as virtual to physical mapping information enabling an attacker to bypass vital kernel security measures, for example, kernel address layout randomization (KASLR) and enabling hardware-fault attacks such as Rowhammer. As this information is therefore usually hidden, an attacker has to use techniques such as the address-translation attack to learn which virtual address is mapped to which physical address, using missing privilege checks of software prefetch instructions. In order to prevent these types of attacks, KAISER (KTPI) was introduced, adding a stricter separation between the kernel address space and the user address space.

In this thesis, we will show that KAISER never entirely prevented address-translation attacks. This is due to prefetch instruction not being the real cause of leakage. We will first analyze the original address-translation attack and uncover the real root cause of the prefetching effect. Based on this analysis, we will show that Spectre gadgets in the kernel code of syscall and interrupt handlers are the real causes of leakage. Thus, speculative execution leads to speculative dereferencing of unclear general-purpose registers in kernel space. Furthermore, we will locate one such gadget causing leakage in the Linux syscall handler of the `sched_yield` syscall. We will analyze the influence of various software and hardware Spectre countermeasures on this speculative dereferencing attack. Furthermore, we will show that even modern Linux kernel versions and Intel CPUs are susceptible. We will conduct the attack on various systems, using different CPUs (Intel, ARM, and AMD), kernel versions, and Linux distributions. Furthermore, we demonstrate several attacks based on this discovery. First, we will build a covert channel that does not depend on shared memory. Second, we will show that an attacker with sufficient address space can directly leak values from user programs, kernel space, and even SGX. The content of this thesis will be presented as a talk at Financial Cryptography and Data Security 2021.

Keywords: operating systems, transient execution, branch prediction, CPU cache

Kurzfassung

Aus sicherheitstechnischen Gründen verstecken viele moderne Betriebssystem Information über den Zusammenhang von virtuellen und physikalischen Adressen von Anwenderprogrammen. Informationen über diesen Zusammenhang ermöglicht es Angreifern, wichtige Kernel-Sicherheitsmechanismen, wie zum Beispiel KASLR zu umgehen und Attacken wie Rowhammer zu ermöglichen. Angreifer müssen daher auf Techniken wie die Address-Translation-Attacke zurückgreifen, die mithilfe Software-Prefetch-Instruktionen ermöglicht, den Zusammenhang zwischen virtuellen Adressen und physikalischen Adressen zu lernen. Um solche Attacken zu verhindern, wurde die KAISER-Technik (KPTI) entwickelt, die für eine striktere Aufteilung zwischen dem Adressraum für Anwenderprogrammen und dem vom Kernel genutzten Adressraum sorgt. In dieser Masterarbeit zeigen wir jedoch, dass KAISER nie wirklich Address-Translation-Attacken verhindert hat, da, statt wie zunächst angenommen, Software-Prefetch-Instruktionen nicht die eigentliche Lücke war, die die Attacke ermöglicht hat. Stattdessen werden wir zeigen, dass spekulativ ausgeführter Kernel Code, auch Spectre Gadgets genannt, in den Systemaufruf und Interrupt Handler des Linux Kernel die Grundlage der Address-Translation-Attacke war. Dabei werden frei verwendbare Register im Kernel spekulativ dereferenziert, die noch Informationen von Anwenderprogrammen beinhalten. Wir werden ein solches Spectre Gadget im Handler des `sched_yield` Systemaufrufes lokalisieren. Basierend auf dieser Erkenntnis werden wir testen, wie sich bereits existierende Gegenmaßnamen gegen Spectre auf diese Attacke auswirken und werden zeigen, dass selbst aktuelle Intel CPUs und aktuelle Linux-Kernel-Versionen betroffen sind. Dabei werden wir die Attacke auf den verschiedensten Systemen mit unterschiedlichen Linux-Kernel-Versionen, Linux-Distributionen und CPU-Typen (Intel, AMD und ARM) laufen lassen. Des Weiteren werden wir zwei praktische Attacken vorstellen. Zuerst werden wir einen verdeckten Kanal (Covert-Channel) entwerfen, und ihn mit anderen versteckten Kanälen vergleichen. Weiters werden wir zeigen, dass, wenn einem Angreifer genug Adressraum zu Verfügung steht, er in der Lage ist, Variablen direkt aus Nutzerprogrammen, Kernel oder sogar SGX zu lernen. Der Inhalt dieser Masterarbeit wird bei der Financial Cryptography and Data Security 2021 als Talk präsentiert.

Keywords: Betriebssysteme, Transient Execution, Branch Prediction, CPU-cache

Acknowledgements

I want to thank my advisors Martin Schwarzl and Daniel Gruss, for their constant support, meaningful discussions, tips, and feedback for this thesis.

Furthermore, I want to thank my friends and family for keeping me motivated and supporting me during my studies. Especially, I want to thank my partner for supporting me and proofreading my English writing.

Thomas Schuster

Contents

1 Introduction **1**
1.1 Structure of this document 2

2 Background **3**
2.1 Virtual Address Space 3
2.2 CPU Caches 6
2.3 Cache Attacks 8
2.4 KAISER (KPTI) 12
2.5 Transient Execution 13
2.6 Transient Execution Attacks 16
2.7 Transient Execution Defense 21
2.8 Covert Channels 26

3 Speculative Dereferencing Analysis **29**
3.1 Address-Translation Attack 29
3.2 Locating the leakage source 32
3.3 Kernel Spectre Gadgets 35
3.4 Speculative Dereferencing using Spectre 37

4 Improving the number of fetches **40**
4.1 Measuring the leakage 40
4.2 Improving the leakage 49

5 Attack Case Studies **51**
5.1 Covert Channel 51
5.2 Dereference Trap (Value Leak) 56

6 Additional Work **61**
6.1 Speculative Dereferencing in Virtual Machines 61
6.2 Speculative Dereferencing inside SGX enclaves 62
6.3 Speculative Dereferencing in Javascript 62

7 Conclusion **64**

Chapter 1

Introduction

Information about physical addresses and how they are mapped to virtual addresses are usually made unavailable to user programs by an operating systems' kernel [30, 31, 80]. This is due to security reasons, as information about physical addresses and virtual addresses enables an attacker to bypass vital kernel security measures, including KASLR [19, 30]. Kernel address space layout randomization (KASLR) is used by an operating system in order to make kernel addresses unpredictable, hardening the exploitation of kernel bugs [19, 30, 31, 39, 96]. Additionally, an attacker can utilize knowledge about physical addresses to run hardware-fault attacks such as Rowhammer [8, 48, 64, 86, 95, 115], which enables an attacker to leak confidential information by inducing bit flips in RAM [30].

To harden the operating system's kernel against these types of attacks, information about the mapping of virtual addresses and physical addresses is hidden to user programs [30, 31, 94]. Therefore, in order to learn this information, an attacker first has to leak it [31, 94]. For that purpose Gruss et al. [31], in 2016, introduced the *address-translation attack*. The address-translation attack enables an attacker to find the physical address to any arbitrary virtual address by allowing fetching of arbitrary kernel addresses into the cache [31]. The attack thus exploits missing privilege checks of software prefetch instruction [31]. In order to prevent these types of attacks, in 2017, Gruss et al. [30] presented the KAISER technique, which introduces a stricter separation between the kernel address space and user address space, thus harding prefetching kernel addresses via a user program [30]. However, KAISER never fully prevented prefetch attacks such as the address-translation attack.

In this master thesis, we will show that the original analysis of the address-translation attack was erroneous. Thus, we will analyze the root cause of the prefetching effect [13, 31, 65]. We will show that missing privilege flags of software prefetch instruction are not causing the kernel addresses to be fetched into the cache [31]. Instead, speculative execution [13, 65, 72] in the kernel leads to *speculative dereferencing* of user-space addresses stored in general-purpose registers [94]. Many possible sources of speculative

dereferencing might exit in kernel code. However, this thesis's focus will be on Spectre gadgets located in syscall handles and interrupt routines [13, 65, 94]. We will locate an actual Spectre-BTB [13,65] gadget in the syscall handler of the `sched_yield` syscall that can be triggered to fetch arbitrary addresses stored in registers.

Based on these findings, we will show that KAISER never fully mitigated address-translation attacks [30,31]. We will analyze how various software and hardware Spectre countermeasures influence the attack and show that the attack can even be conducted on current Linux kernel versions and the most recent Intel CPUs [5,13,30,43–45,57–59, 78,85,87,92,107,110,113,118]. The attack will be conducted on various systems using various syscalls, and the number of cache fetches caused by speculative dereferencing will be recorded. Intel CPUs, as well as ARM CPUs and AMD CPUs, will be tested [94]. Based on this data, we will optimize the rate of cache fetches for practical attacks.

Finally, we will conduct two practical attacks. First, we will construct a speculative dereferencing based covert channel in order to compare its performance to a covert channel based on other hardware vulnerabilities [11, 25, 32, 37, 69, 73, 76, 77, 79, 83, 88, 97, 114, 117, 119]. Furthermore, we will present dereference trap, a technique that can be utilized to leak data directly from registers using speculative dereferencing. In this manner, attacks using this technique do not need any further encoding steps and can leak data from user programs, from kernel space, and even from SGX [94].

1.1 Structure of this document

In Chapter 2, we will provide background information necessary for this thesis. We will examine CPU caches, cache side-channel attacks, and transient execution. The background chapter will be followed by the systematic analysis of speculative dereferencing in Chapter 3. We will analyze the address-translation attack by Gruss et al. [30] and locate a Spectre-BTB gadget [13,65] causing leakage in a syscall handler. In Chapter 4, we will evaluate speculative dereferencing on various systems and using various syscalls. Based on this information, we will optimize the number of fetches during a speculative dereferencing attack. Two case studies will be conducted in Chapter 5. We will build and benchmark a cache-based covert channel [32, 37, 73, 76, 77, 83, 88, 114, 117], as well as introducing the Dereference Trap technique. In Chapter 6, we will give a short overview of additional work and experiments conducted by Schwarzl et al. [94] based on this thesis's findings. Finally, we will summarize the thesis in Chapter 7.

Chapter 2

Background

In this chapter, we will give an overview of relevant topics. Therefore, we will provide an introduction to address translation, kernel protection mechanisms, CPU caches, cache attacks, and transient execution. Furthermore, we will discuss three major transient execution attacks, namely Meltdown, Spectre, and Foreshadow.

2.1 Virtual Address Space

Virtual addressing is a crucial part of memory isolation in modern operating systems. For many modern architectures, the operating system assigns each process its own virtual address space that can not be accessed by other processes [104]. Furthermore, virtual memory prevents direct access to physical memory by user-space processes. When a virtual address is accessed, address translation is used to find the corresponding physical address. Address translation uses multi-level page tables, which are isolated between different processes by the operating system's kernel. The translation between virtual and physical addresses is usually performed by the memory management unit (MMU), which is often a part of the CPU.

To protect the kernel from access by user-space processes, the virtual address space of a user process is further divided into two sections, as illustrated in Figure 2.1 [104]. The user address space is mapped as user-accessible and can be accessed by the process at any time [104]. The kernel address space, however, is only accessible for a process when the CPU with the privileged bit set, for example, during the execution of a syscall. This separation is a cornerstone of kernel security, which is based on preventing illegitimate access to kernel information from user-space. However, in recent years more and more attacks have shown that these security measures can be bypassed by, for example, using hardware side-channel attacks [30, 31].

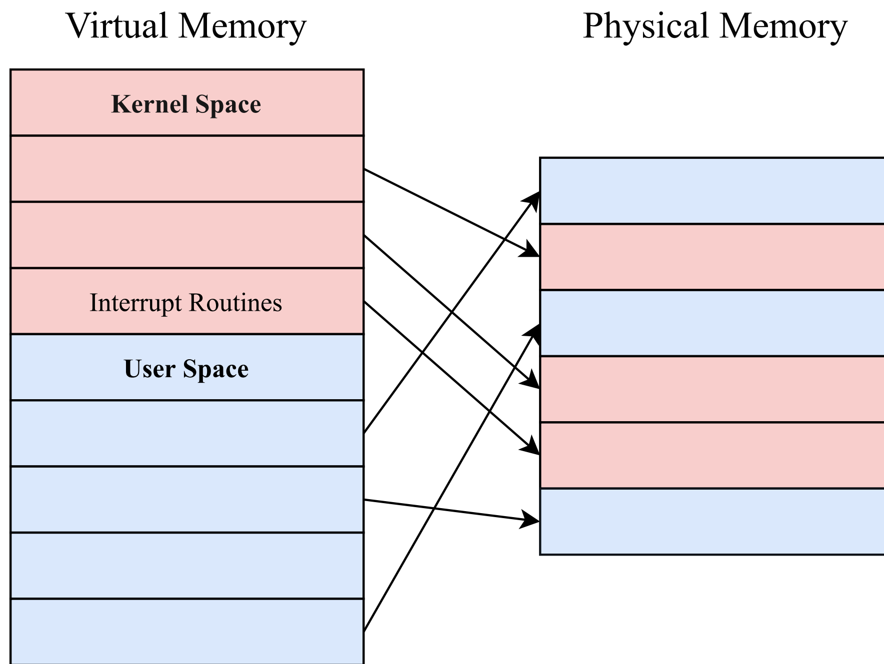


Figure 2.1: The virtual address space of a process. [104]

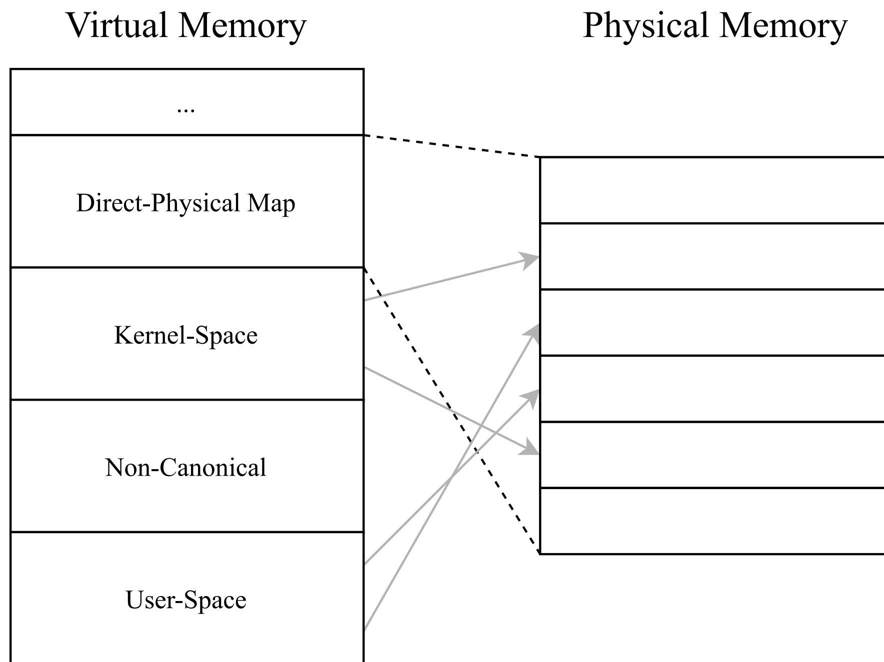


Figure 2.2: On Linux and OSX, physical memory is mapped twice, once as a kernel or a user page and once as a 1:1 mapping [61, 71].

In Linux and OS X, there often exists a direct memory mapping of all physical memory in the kernels' virtual address space [61, 71], as illustrated by Figure 2.2. On Windows, memory pools residing in the kernel address space include a huge fraction of directly mapped physical memory [72]. Due to the vast amount of available virtual address space for 64bit systems, enough virtual addresses are available to map a machine's entire physical memory [61]. Mapping all physical memory directly enables more comfortable and quicker access to physical memory for a kernel driver or the kernel itself [71]. In some operating systems like Linux, information about virtual-to-physical address mappings is available [62]. However, the mapping information can usually not be access by non-privileged programs in order to prevent attacks [98]. In 2016 Gruss et al. [31] proposed a prefetch side-channel attack that can be used to obtain the physical address for any mapped virtual address in user-space.

2.1.1 Kernel Protection Mechanisms

There exist a variety of memory safety violations like buffer overflows, enabling control-flow hijacking attacks [103]. Attackers usually exploit these memory safety violations to attack user-space applications [103]. However, control-flow hijacking attacks are not limited to programs running in user-space but can also be used for attacking the kernel of an operating system [38]. Therefore, modern operating systems use a multitude of hardware and software security mechanisms to protect the kernel from malicious user-space applications [31, 80].

For hardware countermeasures against code injection attacks, most modern CPUs support supervisor mode execution protection (SMEP) and supervisor mode access protection (SMAP) [80]. SMEP is used to prevent the execution of code residing in user-space memory in privileged mode [80]. This countermeasure prevents an attacker from tricking the kernel into executing malicious code while running in kernel mode. SMAP, on the other hand, prevents data access to user-space memory in privileged mode [80].

However, while these countermeasures prevent attackers from tricking the kernel into executing malicious code or accessing malicious data, they do not prevent the kernel against code-reuse attacks like Return-Oriented Programming (ROP) [10, 15, 38, 51, 89]. In a code-reuse attack, the attacker searches for existing code gadgets in already executable memory regions [89]. The address of these code gadgets is then injected into the stack and chained together to execute malicious code [89]. The code gadgets often consist of a number of useful instructions combined with a return instruction [89]. By injecting the address of the next gadget into the stack, multiple gadgets can be combined to run nearly arbitrary code [89]. While these gadgets are often found in user-space libraries, code-reuse attacks can also be used to attack the kernel, bypassing countermeasures like SMEP and SMAP [38].

To harden the execution of code-reuse attacks, various countermeasures were intro-

duced [3, 17, 31, 68, 103]. These countermeasures include Control-Flow Integrity protection (CFI) [3, 68]. CFI uses techniques like shadow stacks [17] or stack canaries [18] in order to prevent an attacker from redirecting the flow of a program's execution. Furthermore, address space layout randomization (ASLR) [31, 103] can be used in order to harden a system against code-reuse attacks. When using ASLR, the virtual memory layout is randomized for every process started [31, 103]. There exists coarse-grained ASLR and fine-grained ASLR. Coarse-grained ASLR randomizes the location of memory regions on process start [31, 103]. These memory regions include the code, heap, data, and stack regions [31, 103]. Additionally, in order to prevent ROP-like attacks, the memory locations of libraries are randomized [31,103]. Coarse-grained ASLR prevents an attacker from predicting the memory location of possible code gadgets needed for ROP, as well as the memory location of already injected code and malicious data [31, 103]. When using fine-grained ASLR, however, even the memory locations for functions and variables are randomized. Fine-grained ASLR, however, usually has a negative performance impact and is therefore rarely used [31, 103].

While ASLR is widely used for protecting user-space processes from code injection attacks, many operating systems additionally utilize kernel address space layout randomization (KASLR) to protect the kernel from ROP attacks [19, 31]. On Linux, KASLR randomizes the kernel virtual address space at boot time [19]. This randomization includes the area where the kernel image is loaded [19]. As this randomizes the address of possible code gadgets in the Linux kernel and kernel libraries, KASLR protects the kernel against ROP attacks [19, 31].

2.2 CPU Caches

Access time to physical memory is high [35]. Thus, small and fast memory is used as a buffer to store recently-used data expecting it to be reaccessed in the near future. Caching frequently accessed memory locations, therefore, leads to significant time-saving. A cache is organized in multiple cache sets with `n` cache lines each, called an `n`-way cache. The size of one cache line is typically 64 bytes. In modern processors, caches are usually *n-way set-associative*, where n is the number of cache lines per cache set, as Figure 2.3 illustrates. Which of the sets is used depends on the address accessed. To find out if the data is cached in a set, the tag part of the address is then compared to the tags of all the cache lines in the set. Besides regular caches, special caches such as the translation-lookaside buffer exist. This buffer stores recent traversed page table entries for faster CPU access.

In x86 and other modern architectures usually multiple cache levels are used, as Figure 2.4 illustrates [35]. These levels typically differ in size and speed [35]. The smallest cache is usually the fastest. In such multi-level cache designs, usually, cache inclusion policies are used [99]. The cache levels can either be inclusive, exclusive, or non-inclusive/non-

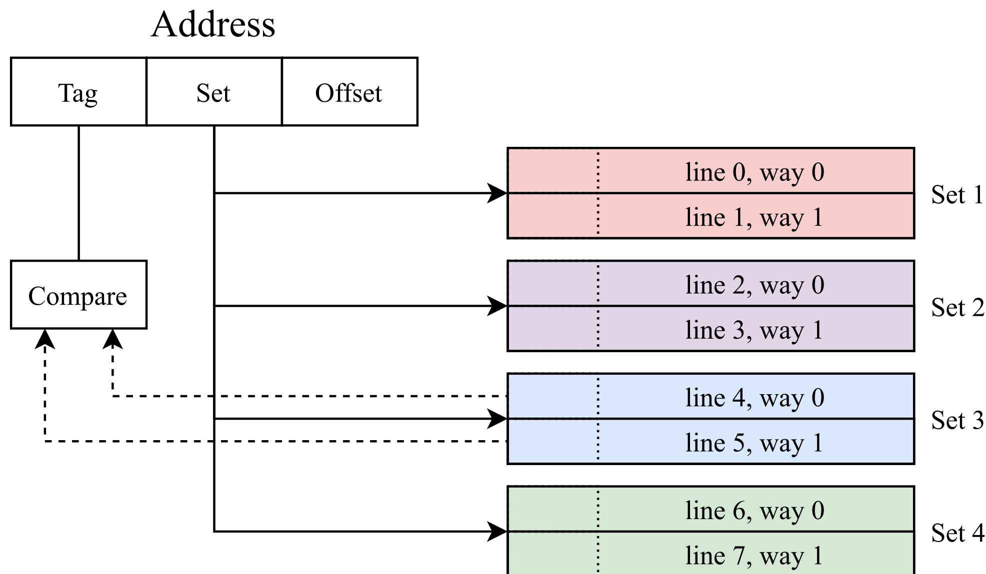


Figure 2.3: 2-way set-associative cache with 8 cache lines in 4 sets, 2 lines per set. On access, the set is chosen, and the tags of the lines are compared with the tag of the address. The same memory location is always cached in the same cache set. [35]

exclusive (NINE). In the case of two inclusive caches, each entry of the lower-level cache is additionally added to the higher-level cache. Eviction, however, only additionally removes the entry from lower-level caches. In an exclusive policy, the higher-level cache is only filled with entries previously evicted from the lower-level cache. A NINE policy is similar to an inclusive cache, however, eviction only removes the entry from one cache level.

In modern processors, usually 3 cache levels are used, denoted L1, L2, and L3 [46]:

- Level 1 Cache [46]: There exists one level 1 cache per CPU core. The level 1 cache is the fastest yet smallest cache. It is usually separated into a data cache and an instruction cache. It is only accessed by virtual addresses, not physical addresses.
- Level 2 Cache [46]: As with the level 1 cache, every CPU core has its own level 2 cache, which is exclusive to the level 1 cache. The level 2 cache is bigger compared to the level 1 cache; however, access is slower. Furthermore, it is not separated into a data cache and an instruction cache.
- Level 3 Cache (LLC) [46]: The level 3 cache, or Last Level Cache, is the biggest yet slowest cache. It is shared between all CPU cores and split up in slices. Furthermore, it contains all the data from all level 1 and level 2 caches, making it a shared exclusive cache [99]. On AMD CPUs, however, a NINE policy is often used [99].

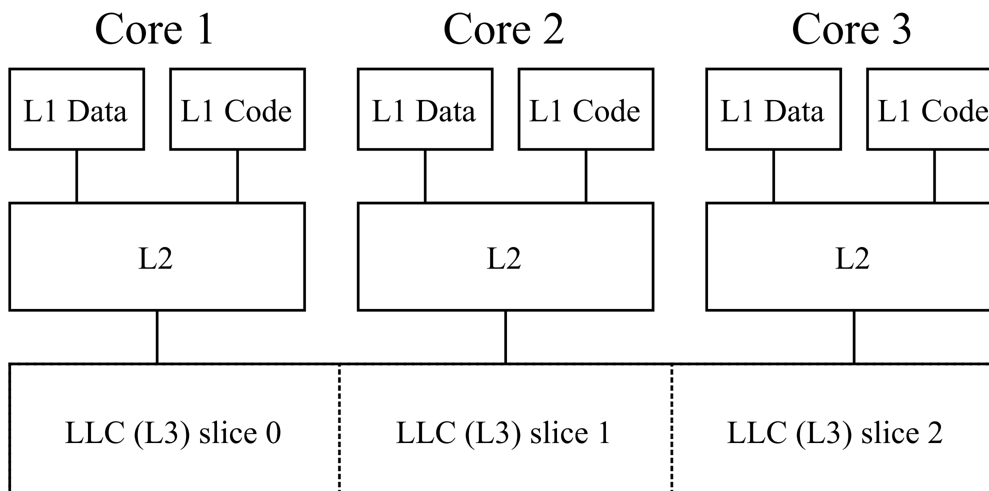


Figure 2.4: Illustration of the level 1, level 2, and level 3 cache on a multi-core processor [46].

When accessing data, a cache hit or cache miss can occur [35]. In the case of a cache hit, the data resides in one of the cache levels and can be accessed quickly. In the case of a cache miss, data will be loaded from slow physical memory and saved into the cache, overwriting a previous cache line entry chosen by a replacement policy like *Least Recently Used (LRU)*. Therefore, cache entries can be evicted by accessing a certain amount of data with addresses leading to the same cache line. Alternatively, the unprivileged `cflush` instruction [21] can flush the cache entry for the data at a given address. Flushing data will evict it from all cache levels.

A program can provide a hint to the processor on which data to fetch and put into the cache using software prefetching instructions [4, 41]. The program can use these instructions to tell the processor to cache an address prior to usage. Intel and AMD CPUs have multiple instructions for software prefetching, including `prefetcht0`, `prefetcht1`, `prefetcht2`, and `prefetchnta` [4, 41]. The result of these instructions, however, is uncertain, as processors might ignore these hints [46].

2.3 Cache Attacks

Cache attacks are side-channel attacks that allow an attacker to collect information about a victim's programs' behavior by determining which data is used and cached during execution [32, 37, 50, 66]. Hu [37] first mentioned the idea of leaking information cross-process using the cache in 1992. Kocher [66] and Kelsey et al. [50] in 1996 and 2000 describe the theoretical usage of cache timing attacks in cryptoanalysis to attack

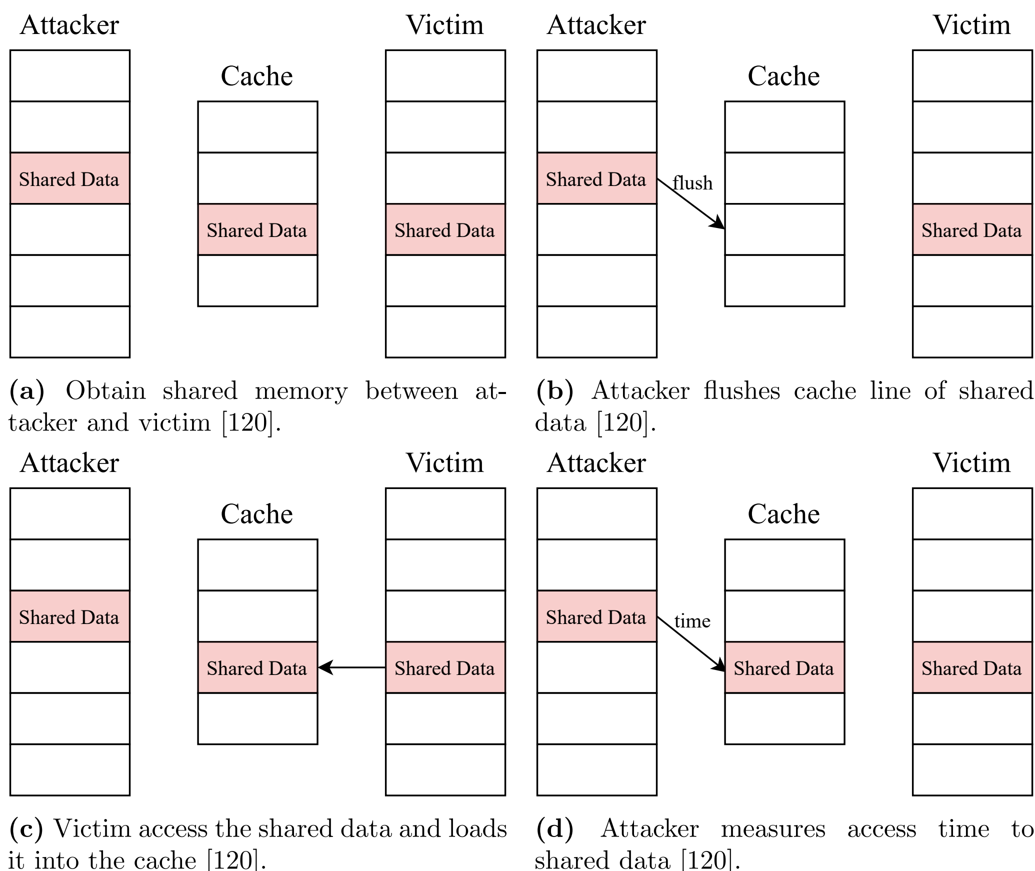
cryptosystems implementations of, among others, DES, RSA, DSS, and Diffie-Hellmann. Cache Attacks exploit the difference in the fast access time of cached data compared to the long access time of uncached data [82, 106]. They can therefore be classified as timing side-channel attacks [82,106]. Practical cache timing attacks were first discussed by Page [82] and Tsunoo et al. [106] in 2002 and 2003 to attack DES implementations. In 2004 the first attack on AES was published [6]. Percival [83] suggested an attack that determines which cache sets are occupied by a victim program by measuring access time to cache ways. Based on this, Osvik et al. [81] and Tromer et al. [105] suggested various attack techniques on AES.

In the last two decades, several techniques were introduced that allow an attacker to utilize the cache to collect information about a victim [32,34,81,120]. The most important techniques are:

- Evict+Time [81]: This technique was introduced by Osvik et al. [81] in 2006 and consists of three steps in order to learn which cache sets are accessed by a program. At first, the victim program is executed and the execution time is measured [81]. Second, the attacker evicts a certain cache set from the cache by accessing certain addresses [81]. Finally, the victim program is timed again. If the execution time increases, the attacker learns that the evicted cache set was probably accessed by the victim [81].
- Prime+Probe [81]: First, an attacker occupies several cache sets and runs the victim program (Prime) [81]. Second, the attacker probes which cache sets are still occupied and learns information about which data was accessed by the victim [81]. This information can be learned by observing a slow access time for evicted cache sets in comparison to fast access times for cache sets not accessed by the victim. Osvik et al. [81] proposed this technique in 2006.
- Flush+Reload [34, 120]: This technique was presented by Gullasch et al. in 2011 [34] and Yarom and Falkner in 2014 [120]. It utilizes the `cflush` instruction and shared memory in order to determine which addresses were accessed by the victim [120]. We will discuss this technique in more detail in the next section [120].
- Flush+Flush [32]: Flush+Flush is a stealthy cache attack technique introduced by Gruss et al. in 2016 [32]. The technique is similar to Flush+Reload [34, 120], however it solely uses the `cflush` instruction, as the execution time of `cflush` is faster for cached data compared to uncached data.

2.3.1 Flush+Reload

Flush+Reload is a cache-based side-channel attack that utilizes the fact that memory shared between two processes is cached in the same cache sets [34,120]. The idea behind Flush+Reload was first proposed by Gullasch et al. [34], attacking AES on the L1 cache



(a) Obtain shared memory between attacker and victim [120].

(b) Attacker flushes cache line of shared data [120].

(c) Victim access the shared data and loads it into the cache [120].

(d) Attacker measures access time to shared data [120].

Figure 2.5: A Flush+Reload attack illustrated. After measuring the access time, the attacker learns if the shared data was accessed [34, 120].

utilizing shared memory. Yarom and Falkner [120] improved this idea and introduced the Flush+Reload technique targeting the L3 cache in 2014.

Flush+Reload consists of four steps, as illustrated in Figure 2.5 [120]:

1. Obtain a shared memory region with a victim program [120].
2. Choose an address from the shared memory region and flush the cache line from the cache using the `cflush` instruction [120].
3. Wait for the victim process to run [120].
4. Measure the access time to the shared memory address [120].

A shared memory region between an attacker and a victim can be obtained in multiple ways [102]. Dedicated shared memory, shared binaries, shared libraries, and, if activated, memory optimizing algorithms like page deduplication can provide an attacker with a shared memory region [102].

By measuring the access time, one can infer whether the victim program accessed the shared memory region under attack [120]. In the case of a cache hit, we can observe a fast access time [120]. In the case of a cache miss, the access time is significantly longer [120]. This difference enables an attacker to search for memory regions accessed by arbitrary algorithms [120]. Yarom and Falkner [120] presented an attack using Flush+Reload that extracts parts of the private key of the RSA implementation in GnuPGP.

Flush+Reload is considered a low noise cache side-channel attack [120], making it feasible for a broad range of applications. Gruss et al. [32] showed that the probability of false positives using the Flush+Reload technique is very low. In their experiments, they were able to observe an accuracy between 96% and up to nearly 100% for correctly monitoring keystrokes [32].

2.3.2 Cache Template Attacks

The Cache Template Attacks is a two-phase cache attack technique introduced by Gruss et al. [33] in 2015. It is a generic attack, enabling an attacker to automatically conduct cache-based attacks on any program [33]. Information about Program versions or system-specific information is not required by a Cache Template Attack [33]. Furthermore, remote systems can be attacked without prior offline measurements. The technique uses Flush+Reload [120] as an underlying attack [33].

Cache Template Attacks are conducted in two phases [33]. At first, a profiling tool determines and collects information about the connection between secret information and certain memory areas being accessed and cached [33]. This secret information can, for example, be keystrokes or private keys used in cryptographic primitives [33]. In the second phase, the exploration phase, the attacker then deduces details about the secret information by observing the cache [33]. As an example, Gruss et al. [33] showed an attack detecting keystrokes on specific keys using a Cache Template Attack.

Gruss et al. [33] provide a collection of public domain tools to perform Cache Template Attacks [27]. These tools can be used on Windows and Linux [27]. It contains programs for profiling and exploitation, as well as a calibration tool and a C header file providing functions for Flush+Reload [27]. The calibration tool can be used to obtain a histogram of multiple cache hit and cache miss time measurements on the current system [27]. Figure 2.6 shows a visualization of the calibration tool histogram. The histogram can be used to learn the optimal threshold to differentiate between a cache hit and a cache miss [27,33]. The optimal threshold is the highest timing of a cache hit to minimize false negatives [27, 33]. However, the threshold has to be lower than the lowest timing of all cache misses to prevent false-positive results [27, 33].

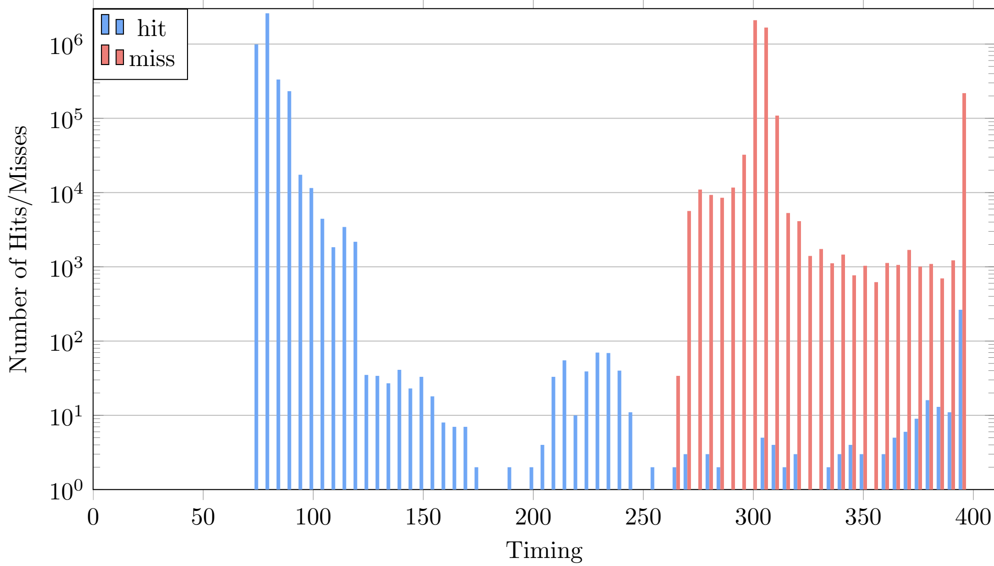
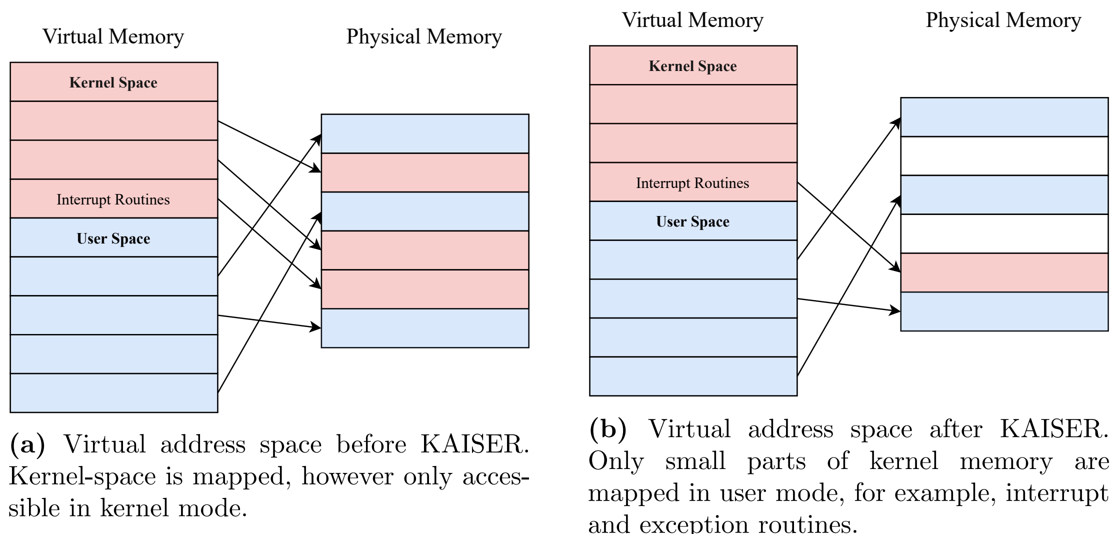


Figure 2.6: This diagram is a visualization of the histogram provided by the Cache Template Attack calibration tool [33]. The optimal threshold in this example would be around 250. High access time for cache hits might be caused by scheduling.

2.4 KAISER (KPTI)

The KAISER technique [29, 30] strengthens kernel security and prevents many side-channel attacks by enforcing strict isolation between user-space and kernel-space. By using this technique, almost no kernel pages are mapped while a user process is running in user mode [30]. The kernel uses two separate sets of page tables, one for user-space and one for kernel-space [30]. The full set of page tables includes all user-space and kernel-space mappings and is only used while the CPU runs in kernel mode [30]. The second set of page tables is restricted to user-space addresses and only a small number of kernel-space addresses, as illustrated in Figure 2.7 [30]. These kernel-space addresses contain information needed for entering and exiting syscall, interrupt, and exception routines [30].

Kernel features based on the KAISER technique [29, 30] were implemented under the name *Kernel Page-Table Isolation* (KPTI) [16] for Linux, on MAC as Double Map [47], and *Kernel Virtual Address Shadowing* (KVAS) [49] for Windows as a mitigation for the Meltdown attack. KPTI might have a negative impact on performance [13, 26, 29]. For processors with PCID support, overheads were reported as negligible (0-2.6%), while for systems without PCID, in the worst-case, overhead went all the way up to 800% while executing a considerable number of syscalls [13, 26, 29].



(a) Virtual address space before KAISER. Kernel-space is mapped, however only accessible in kernel mode.

(b) Virtual address space after KAISER. Only small parts of kernel memory are mapped in user mode, for example, interrupt and exception routines.

Figure 2.7: The virtual address space of a process before and after applying the KAISER patch. [104]

2.5 Transient Execution

Many modern CPUs do not work with the instruction set directly [23]. Instructions are further split up and translated into micro-operations (µOPs) by newer processors [23]. These µOPs can, for example, be reading from memory into a register, execute calculations using data in registers, and writing from registers into memory [23]. Instructions that only use registers like `ADD RAX, RBX` are translated into only one µOP [23]. For `ADD [MEM1], RBX`, the processor will generate three µOPs, read into a register, execute addition, and write back into memory [23].

Spreading an introduction onto one or more µOPs enables the processor to use out-of-order execution [23]. Out-of-order execution improves the performance of the processor by minimizing the number of otherwise wasted instruction cycles [23]. This performance improvement is achieved using the time a processor has to wait for an instruction to complete, for example, while loading data from memory [23]. While the processor is waiting for the delayed instruction to complete, following instructions with already available inputs are antedated [23]. However, this is only possible if there is no dependency between the delayed and the following instructions [23].

Figure 2.8a shows an example of assembler code [23]. The `mov eax, [mem1]` will split up into fetching the value `[mem1]` and moving it into `eax` [23]. As `imul eax, 5` is dependent on the not yet available new value for `eax`, it can not be antedated [23]. However, `add eax, [mem2]` will be split up into fetching and adding the value to `eax` [23]. As loading `[mem2]` is not dependent on the value from `[mem1]`, the processor can start fetching

```
1 mov eax, [mem1]
2 imul eax, 5
3 add eax, [mem2]
4 push eax
```

(a) Simple assembler code example.

```
1 load: [mem1]
2 mov: [mem1] into eax
3 mul: eax with 5
4 load: [mem2]
5 add: [mem2] to eax
6 sub: 4 from esp
7 mov: eax into [esp]
```

(b) Code split up into example µOPs.

Figure 2.8: Code example for instructions being split up into µOPs [23]. Code adapted from *The microarchitecture of Intel, AMD, and VIA CPUs* by Agner Fog [23].

`[mem2]` prior to the availability of `[mem1]` [23]. Furthermore, subtracting from the `esp` for the `push` instruction can be executed out of order, as no other µOP is dependent on the stack pointer [23].

Branch prediction is a technique that tries to predict which path will be used after a conditional jump [23]. By determining which path will be used for conditional branches and where the branch is going for unconditional and conditional branches, the processor can fetch instruction from memory prior to their actual usage [23]. For example, the Intel processors' branch prediction unit uses multiple different branch prediction structures [46]:

- Branch Target Buffer (BTB) [20, 65, 70]: The BTB is a cache-like structure that saves information about previously taken branches [20]. The information usually includes the target of the jump and whether the branch was taken or not [20]. This buffer is then used by a branch target predictor to predict the target of a conditional or unconditional branch without the need to decode and compute the real target address [20].
- Branch History Buffer (BHB) [7,65]: The BHB is a table that stores branch instructions and a bit that indicates whether this branch was recently taken or not [7,65]. A branch predictor can use this information to speculatively execute the branch that will probably be taken based on the information saved in the BHB [7, 65]. In the case of a correct prediction, the processor has already executed the branch instruction [7, 65]. Otherwise, the processor has to flush the wrongly executed instructions out of the pipeline and execute the correct branch [7, 65].
- Pattern History Table (PHT) [23, 65]: The PHT improves the prediction based on the BHB by saving the history of a branch being taken in two bits instead of one [23]. Additionally, the PHT saves four counters, indexed by the two bit history [23]. Every time a certain 2-bit history leads to a branch taken, the cor-

responding counter is incremented [23]. This enables a branch predictor to detect patterns and correctly predict branches based on multiple previous executions [23].

- Return Stack Buffer (RSB) [23, 67, 75]: RSBs are small and fast buffers that store return address of recently executed call instructions [23]. Every time a call or return instruction is executed, the return address is pushed onto a stack or popped from the stack [23]. RSB utilizes the fact that call instructions and return instructions are often executed in pairs [23]. On return, the processor can predict the presumed return address by taking it from the RSB [23]. This circumvents the loading time needed to access the main memory to get the return pointer from the stack [23].

Modern processors utilize speculative execution to further optimize performance [23]. This optimization is used as a countermeasure in order to tackle the problem of the growing gap between processor speed and memory access speed [23]. In speculative execution, a predictor assumes which path will be executed or which value will be loaded and speculatively continues execution [23]. The prediction mechanism can base this assumption using either control-flow prediction or data-flow prediction [23]. In the case of a correct prediction, the processor can use the result of the already executed instructions [23]. However, in the case of a wrong prediction, the result of the speculatively executed instruction has to be discarded [23]. Furthermore, the pipeline has to be flushed, and the correct path has to be executed [23]. As these discarded instructions are executed in a transient way (*transient instructions* [65, 72]), this is also called *transient execution* [13, 65].

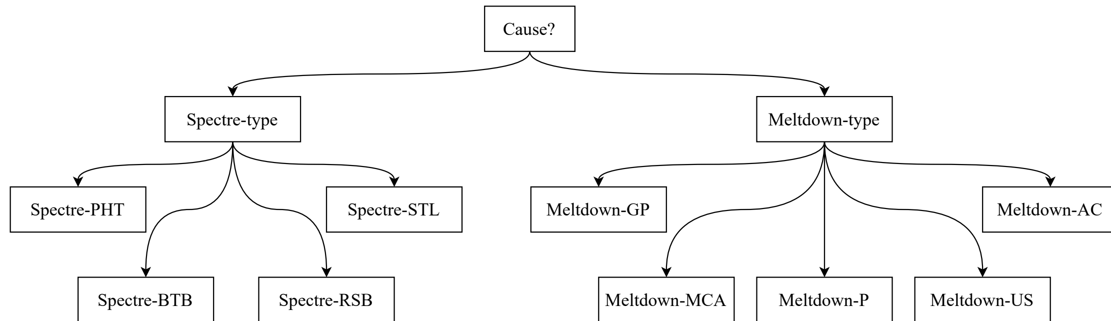


Figure 2.9: Classification tree of transient execution attacks [13, 28]. Split up into Meltdown-type and Spectre-type attacks [13, 28]. Based on a graph by Canella et al. [13, 28].

```
1 char data = *(char*)0xffffffff81a000e0;
2 array[data * 4096] = 0;
```

Figure 2.10: Toy example of the Meltdown attack [72]. A kernel address is dereferenced, and the result is used to index an array [72]. Due to speculative execution, the indexed part of the array might be prefeched before the memory access permission check can detect invalid access to a kernel address [72].

2.6 Transient Execution Attacks

Transient execution enables the processor to optimize performance [13,65]. On the other hand, however, it can be used by an attacker to leak secret information [13,65]. When using out-of-order execution, as a side effect, the microarchitectural state of the processor is changed [13,65]. These changes, for example, cached memory, are not reverted in the case of a wrongly executed branch, enabling an attacker to learn sensitive information by exploiting this side effect [13, 65]. Instructions that are out-of-order executed and produce such side effects are called transient instructions [72].

Attacks that exploit side effects of transient execution are called transient execution attacks [13, 65, 72]. Transient execution attacks can be separated into two groups [13], Spectre-type attacks [65], and Meltdown-type attacks [72]. The first group is Spectre-type attacks, which exploit misprediction of control flow or data flow [13, 28], caused, for example, by branch prediction. The second group is Meltdown-type attacks which exploit transient execution on an instruction that will raise a CPU exception [13, 28]. While Spectre-type attacks affect Intel, ARM, and AMD processors, Meltdown-type attacks can not be executed on AMD processors [72]. An overview of the classification of some known transient execution attacks is illustrated in figure 2.9. The first attacks utilizing transient execution were found in 2017 and published later in 2018 [65, 72]. In the following, we will explain several transient execution attacks in more detail.

2.6.1 Meltdown

The original Meltdown attack (also referred to as Meltdown-US-L1 [13]) was published by Lipp et al. [72] in the first quarter of 2018. Meltdown exploits a race condition between memory access and memory access permission check [72]. When a program accesses a memory location, the CPU will usually check the user/supervisor attribute of a pagetable [72, 104]. This attribute indicates if a page was mapped and can only be accessed by the kernel [104]. If a user program attempts to access virtual addresses that

point to a kernel-owned virtual memory page, an exception is raised, and the accessing program is terminated [104]. Meltdown exploits the fact that, when utilizing out-of-order execution, the microarchitectural state can be modified, regardless of the exception being raised by the processor when accessing a kernel address [72].

The attack consists of three steps [72]:

1. Reading the secret [72]: An attacker loads the virtual address of a memory location into a register [72]. The content of the memory location is then used as an index to access an array in user-space, as described in the next step [72]. During loading, the CPU will translate the virtual address into a physical address [72]. Furthermore, the processor will check the permission bit in the pagetable and raise an execution in the case of illegal access [72]. The instruction sequence reading the secret and accessing the array must therefore be implemented in a way that it becomes a transient instruction sequence that will be executed out-of-order [72].

2. Transmitting the secret [72]: A sufficient-sized array is allocated in order to transmit the secret, working as a lookup table [72]. First, it is ensured that no part of the array is cached [72]. Second, the array is accessed at an offset based on the secret value read in step one [72]. Due to transient execution, a race condition between the CPU exception raised and the access to the array will occur [72]. This might lead to the array being cached at the secret-based offset [72].

3. Receiving the secret [72]: The attacker now utilizes the cache-based side-channel attack Flush+Reload [120] in order to learn which offset at the array was being cached [72]. Using this offset, an attacker is now able to deduce the secret read in step one, regardless of the fact that the virtual address only accessible in kernel mode [72].

A toy example code for this can be seen in Figure 2.10. In this example, the dereferenced address is a kernel address. The `array` might be speculatively accessed before a page-fault can occur [72]. In this case, the `array` will be cached at the value of `data` times 4096, revealing the content of `data` by using cache attacks [72].

Meltdown enables an attacker to read the entire physical memory [72]. The attack described above can be repeated for multiple different memory locations, dumping the entirety of the kernel memory [72]. Linux and OS X usually have the entire physical memory mapped in the kernel virtual address space [61, 71], while Windows typically maps a large fraction of the physical memory in the kernel address space in the form of memory pools [72]. Therefore, physical memory content will most likely be part of the resulting kernel memory dump [72].

```
1 if (x < array1_size)
2   y = array2[array1[x] * 4096];
```

Figure 2.11: Toy example of the Spectre attack [65].

2.6.2 Spectre

The original Spectre attack was published by Kocher et al. [65] together with Meltdown [72] in the first quarter of 2018. Spectre attacks use speculative execution to trick the processor into executing an instruction sequence that would not be executed during a strictly serialized program run [65]. First, an attacker has to mistrain a prediction mechanism of the processor, for example, a branch predictor [65]. Second, an instruction set containing a non-reachable branch in which confidential information is used will be executed [65]. Due to the mistraining of the prediction mechanism, the processor will speculatively execute the branch, normally not being reached [65]. As this leaves changes in the microarchitectural state, side-channel attacks can be used to extract confidential information [65].

A toy example for one of the first Spectre attacks, known as Spectre-PHT or Spectre v1, can be seen in Figure 2.11 [13, 65]. Due to branch prediction, the access to `array1` might be executed before `x` was validated in the branch condition, enabling `x` to be out of bounds [65]. `array2` will be cached at the value of `array1[x]` times 4096, revealing the content of `array1[x]` by using cache attacks [65].

Spectre can exploit multiple possible prediction mechanism [13, 28]:

- Spectre-PHT [13, 65]: One of the versions of the Spectre attack described in the first Spectre paper by Kocher et al. [65]. In this attack, the Pattern History Table (PHT) and the Branch History Buffer (BHB) are mistrained in order to mispredict the outcome of a conditional branch [65].
- Spectre-BTB [13, 65]: This version of the Spectre attack mistrains the Branch Target Buffer (BTB) in order to wrongly predict the destination address of a conditional branch [65]. Compared to Spectre-PHT an attacker can speculatively execute arbitrary instructions sets [65]. The attack is therefore not restricted to a certain conditional branch contained in the executing path like in Spectre-PHT [65].
- Spectre-RSB [13,67,75]: Spectre-RSB, also known as ret2spec, is a version of Spectre that exploits the Return Stack Buffer (RSB) [67,75]. By mistraining the Return Stack Buffer, arbitrary code across processes can be speculatively executed [67,75].
- Spectre-STL [13,36]: Store To Load (STL) dependencies require a memory location to be free of pending store instructions before being loaded [13, 36]. However, the

processor might speculatively predict which memory loads can already be executed speculatively [13, 36]. In Spectre-STL, an attacker can mistrain this prediction mechanism and speculatively bypass store instructions [13, 36].

For mistraining the prediction mechanisms, Canella et al. [13] describe four mistraining strategies:

1. Executing the victim branch in the victim process (sameaddress-space in-place) [13]
2. Executing a congruent branch in the victim process (sameaddress-space out-of-place) [13]
3. Executing a shadow branch in a different process (crossaddress-space in-place) [13]
4. Executing a congruent branch in a different process (crossaddress-space out-of-place) [13]

Additionally to the above-mentioned cache-based side-channel Spectre attacks, various Spectre variants using other side channels were found [9,93]. For example, SMoTherSpectre [9] utilizes the port-contention of simultaneous multithreading (SMT) architectures (multiple logical cores share one physical core) as a side-channel [9]. Two hardware threads running on the same physical core contend for the same ports, where each port is responsible for a specific type of execution (e.g., loads or stores) [9]. This leads to a measurable slowdown, enabling an attacker to learn information about the execution sequence run by the victim running on the same physical core [9]. This is done by the attacker using an instruction sequence utilizing the same ports as the victim [9]. Spectre can then be used in order to make the victim execute certain instruction sequences, e.g., using the BTB [9]. NetSpectre by Schwarz et al. [93], on the other hand, uses the execution time difference of AVX2 instructions to enable remote Spectre attacks, called an AVX side-channel. On the victim machine, a Spectre-PHT gadget, as well as a so-called transmit gadget, have to be present [93]. The transmit gadget performs activities based on the microarchitectural state changed by the Spectre gadget, leading to different execution times and, therefore, measurable network latency [93].

2.6.3 Foreshadow

Foreshadow, also known as Meltdown-P-L1 [13], is a Meltdown-type attack [109, 112]. Contrary to the previously described Meltdown variant, Foreshadow is not aimed at bypassing the memory protection provided by the Supervisor/User attribute of a page table [109, 112]. Instead, Foreshadow utilizes page-faults while accessing unmapped pages, pages with the present bit cleared, or the reserved bit set [109, 112]. When accessing such an unmapped page, the processor immediately aborts address translation, referred to as a terminal fault [109, 112]. However, Foreshadow uses the fact that when accessing an unmapped page, in parallel to the address translation, the processor checks

whether the memory location of the physical address of the faulting page table entry is cached in the L1 cache [109,112]. In the case of a cache hit, the data will be immediately used by transient instructions before the processor raises a page-fault, modifying the microarchitectural state [109, 112]. As with Meltdown-US-L1, cache-based side-channel attacks can then be used in order to extract data from the microarchitectural state change [13, 109, 112].

Foreshadow was originally used to extract data out of Intel SGX enclaves [41, 109]. However, it can further be used to bypass operating system protection or hypervisor protection [112]. An attacker can use Foreshadow in order to extract any physical memory cached in the L1 cache from within a guest virtual machine [112]. This includes memory belonging to other guest virtual machines on the same system as well as memory owned by the hypervisor [112].

2.6.4 MDS Attacks

In this section, we will discuss Meltdown-type attacks utilizing Microarchitectural Data Sampling (MDS) side-channels [40, 93]. Therefore, we discuss three such attacks: Fallout [12], RIDL [110], and ZombieLoad [92].

Fallout

Fallout is a Meltdown-type attack utilizing the Microarchitectural Store Buffers Data Sampling vulnerability, which can be used as an MDS side-channel [12]. Store buffers are used by the CPU pipeline in order to reduce latency for data storage when storing any type of data [12]. When loading data, however, these buffers have to be searched for the loading address in the case of yet unwritten addresses and are directly read in the case of matching addresses, called store-to-load forwarding [12]. Fallout enables an unprivileged attacker to leak data from these store buffers, using the so-called Wire Transient Forwarding (WTF) shortcut [12]. The WTF shortcut leaks values from memory writes by using faulting load instructions, abusing store-to-load forwarding [12].

Fallout has been shown to be able to break Kernel Address Space Layout Randomization (KASLR), even recovering address space information from Javascript [12]. Additionally, sensitive data written into memory by the kernel was able to be leaked [12]. Fallout is not affected by recently introduced Meltdown hardware mitigations, showing that even recent processor generations are affected [12].

Rogue In-Flight Data Load (RIDL)

The Rogue In-Flight Data Load (RIDL) attack is a Meltdown-type attack exploiting multiple MDS vulnerabilities [110]. These vulnerabilities include Microarchitectural Load Port Data Sampling, exploiting the CPU's load ports, and Microarchitectural Fill Buffer Data Sampling, exploiting the CPU's line-fill buffer (LFB) [110]. These CPU buffers are being used by the CPU while loading and storing data from and into memory [110]. As an example, the LFB is used by the CPU in order to optimize outstanding memory requests by speculatively loading data into the buffer [110]. The RIDL attack can be used in order to leak sensitive data from other applications running on the same Intel processor [110]. These include the operating system's kernel, VMs (for example, in the cloud), or Intel SGX enclaves [110]. For example, arbitrary kernel memory can be leaked by speculatively loading data previously stored in the LFB by mistraining [110].

ZombieLoad

ZombieLoad is a Meltdown-type attack utilizing the fill buffer structure of modern CPUs [92]. ZombieLoad exploits a vulnerability usually referred to as Microarchitectural Fill Buffer Data Sampling (MFBDS) [40] by Intel. The fill buffer is a buffer allocated and used to gather data in the case of a miss on the first level data cache [92]. The buffer holds data used in load operations or data returned by memory operations to be written into the L1 data cache [92]. Once data is written into the cache, the fill buffer entry is deallocated to be reused by future memory operations on the same physical core [93]. Under certain conditions, the stale data of previous memory operations may be speculatively forwarded during memory operations, causing a fault, referred to as a zombie load [92]. As with other Meltdown-type attacks, the speculative loaded value can then be recovered from the microarchitectural state using established techniques, e.g., cache-based side-channel attacks [92].

ZombieLoad allows leaking data across all privilege boundaries [92]. This includes leaking data from other user processes, the kernel, Intel SGX, and virtual machines [92]. However, compared to other Meltdown-type attacks, ZombieLoad gives an attacker less control over which data is leaked, as only the least-significant 6 bits of the virtual address can be used to address data in the fill-buffer entry [92].

2.7 Transient Execution Defense

In this section, we will give a short overview on some proposed hardware and software defenses for various Spectre and Meltdown-type attacks. Based on the classification by

Canella et al. [13], we will consider Meltdown and Spectre as two separate problems with different causes. Mitigations for Spectre-type attacks can be split up into three categories [13]:

1. Prevent covert channels: Defense approaches that make the usage of certain covert channels infeasible (e.g., by reducing accuracy) or that mitigate certain covert channels entirely [13].
2. Prevent speculation: Abort or mitigate speculative execution in the case of data possibly being accessible during transient executions [13].
3. Isolate secret data: Make secret data unreachable to potential attackers [13].

Mitigations for Meltdown-type attacks can be split up into two categories [13]:

1. Protect data from attacks on a microarchitectural level: Make architecturally inaccessible data inaccessible on a microarchitectural level as well [13].
2. Prevent faults: Prevent faults by making them valid accesses without leaking secret data [13].

2.7.1 Spectre Defense: Prevent covert channels

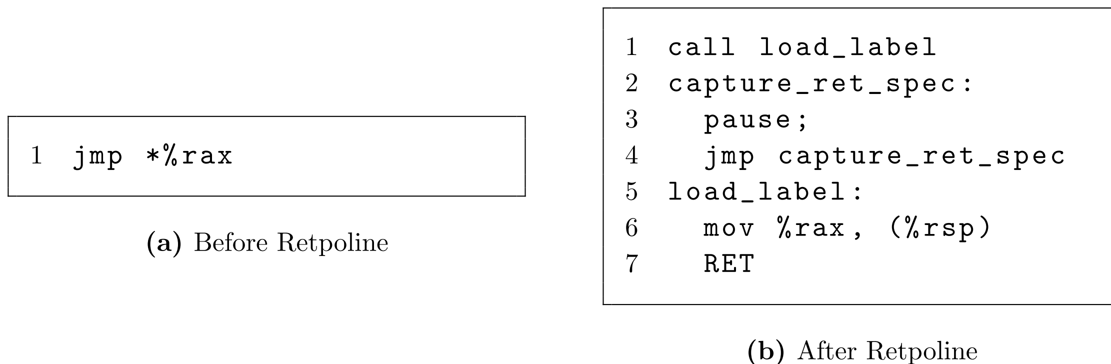
Transient execution attacks usually utilize a covert channel in order to learn information from microarchitectural changes [13]. Preventing covert channel approaches or reducing their accuracy can be used to prevent Spectre-type attacks [13]. Possible hardware countermeasures would use a separate speculative buffer instead of the data cache for all speculatively executed loads, proposed by Yan et al. [118] as InvisiSpec. In the case of a correct prediction, the content is copied into the cache, visible to the rest of the system. In the case of a wrong prediction, the content of the buffer is invalidated [118].

Other hardware-based countermeasures use shadow hardware structures (SafeSpec) [63] or prevent the usage of data loaded during transient execution by subsequent instructions [65]. One example of software countermeasures would be reducing a covert channel's accuracy by removing access to an accurate timer [13].

2.7.2 Spectre Defense: Prevent speculation

An effective way of preventing Spectre-type attacks would be deactivating speculative execution altogether [13, 65]. However, as the performance loss would be too high, deactivating speculation while working with sensitive data is an option [13].

In order to prevent Spectre-BTB (also known as Spectre v2), Intel and AMD introduced several related hardware countermeasures [5, 13, 45]. These countermeasures include



(a) Before Retpoline

(b) After Retpoline

Figure 2.12: Retpoline exchanges the jump instruction of (a) to the sequence seen in (b). First, there is a direct call to `load_label` [107]. The RSB entry after that call leads to `capture_ret_spec`. In `load_label`, the target is pushed onto the stack and returned to using `ret`, while speculative execution based on the RSB is trapped inside the `capture_ret_spec` loop [107].

barriers like the Indirect Branch Predictor Barrier (IBPB) that flushes the BTB [58, 87]. IBPB prevents code executed before the barrier to affect the prediction of the code executed after the barrier [58, 87]. Related to IBPB, Indirect Branch Restricted Speculation (IBRS) flushes the BTB on kernel entry to prevent speculative execution in the kernel by mistraining in user-space [58,87]. Single Thread Indirect Branch Predictors (STIBP), on the other hand, prevent branch prediction-based mistraining caused by sibling CPU threads using hyperthreading [58, 87].

`retpoline` is a software countermeasure introduced by Google for Spectre-BTB (e.g., Spectre v2) as well as Spectre-RSB (e.g., ret2spec) attacks [13, 43, 78, 107]. With `retpoline`, the target of indirect branches is pushed onto the stack and returned to using the `ret` instruction, as shown in Figure 2.12b [43, 107]. This prevents speculative attacks based on indirect branch prediction, as prediction of `ret` only relies on the RSB [43, 107]. Moreover, `retpoline` adds an entry to the RSB, leading to an endless loop when speculative predicting the `ret` instruction [43,107]. Performance overhead for retopline was reported between 5–10% on servers [14, 78].

RSB stuffing is a software countermeasure by Intel available for Skylake and newer architectures, mitigating Spectre-RSB (e.g., ret2spec) [13,43]. When using RSB stuffing, the RSB is filled with the address of a harmless function, for example, during a context switch into the kernel [13,43]. This countermeasure prevents speculative execution based on the RSB, for example, to avoid the execution of unwanted user-space code in kernel mode [13, 43].

Software-based countermeasures also include the `lfence` instruction [5,44] for Intel and AMD processors. It prevents the execution of code after the `lfence` instruction unless all prior instructions are completed, for example, mitigating Spectre-PHT and Spectre-

BTB [13, 78].

For Spectre-STL, Mcilroy et al. [78] conclude mitigation can not be effectively achieved in software [28]. This is due to years of work that would theoretically be required in order to mitigate attack vectors for Spectre-STL, including the redesign of compiler optimization and applying possible software countermeasures to a huge amount of codebases [78]. Moreover, architectural changes would be required in order to prevent reads on speculative writes [78].

2.7.3 Spectre Defense: Isolate secret data

One example of this category is `site isolation` proposed by Google for Google Chrome and Chromium [85]. Site isolation ensures that every site is rendered in its own process, minimizing the amount of data that can be gathered using speculative side-channel attacks [85]. However, enabling site isolation might cause a memory overhead of up to 13%, depending on the number of open tabs [85].

2.7.4 Meltdown Defense: Protect data from attacks on a microarchitectural level

Meltdown-type attacks use transient execution to access architecturally inaccessible values. By preventing access to unauthorized values by design, directly in silicon, Meltdown attacks can be mitigated [13]. While AMD enforced this design in all available processors [5], only recent Intel processors with RDCL_NO support have Meltdown hardware mitigations [45]. For some versions of Meltdown, Intel released microcode updates [44] mitigating attacks.

An example of a software-based Meltdown defense is the KAISER technique, which was explained in Section 2.4 [29, 30]. As Meltdown-US attacks require the secret to being mapped, KAISER prevents attacks on kernel memory locations. Countermeasures based on KAISER were implemented on Linux as *Kernel Page-Table Isolation* (KPTI) [16], on MAC as Double Map [47], and Windows as *Kernel Virtual Address Shadowing* (KVAS) [49].

2.7.5 Meltdown Defense: Prevent faults

Faults are crucial for Meltdown-type attacks, as they utilize the delayed exception handling of the CPU [13]. Therefore, preventing faults will mitigate Meltdown-type attacks. One example of this approach is the countermeasure against Meltdown-NM [101].

Meltdown-NM is a Meltdown-type attack targeting the content of the Floating Point Unit (FPU) registers from other processes by exploiting the delay of the "device-not-available" exception [101]. This exception is raised while accessing the FPU the first time after a context switch, leading to the previous FPU state being saved and the FPU being made available to the current process [101]. The countermeasure proposed for preventing Meltdown-NM prevents the "device-not-available" exception by making the FPU available during a context switch [74]. The countermeasure was implemented in Linux [74].

2.7.6 Transient Execution Defense on Linux

Linux implements several of the above-mentioned Meltdown and Spectre countermeasures [58, 60]. Additionally, many of these countermeasures can be controlled by several kernel boot parameters [58, 60]. This enables a user to disable or enable certain countermeasures [58, 60]:

- `nopti` - Disables Page-Table Isolation (PTI), proposed as KAISER by Gruss et al. [30, 57]. By introducing two separate kernel- and user-space page tables, PTI prevents leaking of kernel memory by user-space applications [30, 57].
- `nospectre_v1` - Deactivates Linux kernel countermeasures against Spectre-PHT (Spectre Variant 1) [13, 28, 65]. The countermeasures include barriers (e.g., the `LFENCE` instruction) for code that is possibly vulnerable to Spectre-PHT type attacks [58]. `LFENCE` barriers prevent transient execution and therefore prevent bounds-check bypass [58]. For Linux, barriers are placed in kernel entry code for interrupts and exceptions, as well as kernel code working with user-space memory [58].
- `nospectre_v2` - Disables the Linux kernel countermeasures against Spectre-BTB (Spectre Variant 2 [13, 28, 65]). Other countermeasures include Indirect Branch Restricted Speculation (IBRS), resetting trained BTB predictions on kernel entry, protecting the kernel from user-space mistraining [58, 87]. Another countermeasure, Indirect Branch Predictor Barrier (IBPB), prevents branch predictions from earlier executions using barriers [58, 87]. Furthermore, Single Thread Indirect Branch Predictors (STIBP), which prevents branch prediction mistraining when using hyperthreading between two sibling CPU threads, is deactivated by this `nospectre_v2` [58, 87]. The last two countermeasures included for this kernel boot parameter are `retpoline`, a software countermeasure that replaces indirect branches like `jmp *%rax` with return trampoline code that traps transient execution, and RSB filling [43]. RSB filling fills the RSB with an address to trampoline code in order to prevent speculative execution on return [43].
- `spectre_v2_user=off` - Similar to `nospectre_v2`, however, this kernel boot pa-

rameter deactivates retpoline, STIBP, and IBPB for code compiled and run in user-space [58].

- `spec_store_bypass_disable=off` - Disables kernel countermeasures against Spectre-STL (Spectre variant 4) [13,28,36], for example, Speculative Store Bypass Disable (SSBD). SSBD prevents speculative loads while stores are still in progress, preventing speculative loading of already invalid data [45].
- `l1tf=off` - Disables kernel countermeasures against L1TF, also known as Foreshadow and Meltdown-P-L1. [13,28,109,112], including flushing the L1 data cache on `VMENTER` [54].
- `mds=off` - Disables mitigations against Micro-architectural Data Sampling (MDS) attacks [55]. Examples of MDS attacks are Fallout and RIDL [12, 110]. The countermeasures include clearing the CPU buffers affected by MDS on user-space or a VM entry [55].
- `tsx_async_abort=off` - Disables countermeasures against the TSX Async Abort (TAA) vulnerability. Attacks exploiting this vulnerability include the ZombieLoad and RIDL attacks [92, 110]. The countermeasures included for this kernel parameter include clearing the affected CPU buffers on ring transition [59].
- `kvm.nx_huge_pages=off` - Disables countermeasures for iTLB multihit-based attacks, including marking huge pages as non-executable when used by KVM [53].
- `dis_ucode_ldr` - In contrast to the previous kernel boot parameter, `dis_ucode_ldr` disables dynamic loading of microcode updates provided by the CPU vendors [113]. Microcode updates are not loaded by the corresponding loader on system start [113]. As many of the above-mentioned transient execution defenses depend on microcode updates, this parameter deactivates many countermeasures [113].

2.8 Covert Channels

Covert channels are used to allow communication between processes that should typically not be allowed to communicate with each other [79]. They do not use the legitimate data transfer mechanisms of a system [79]. Therefore, they can usually not be detected by the security mechanism of a system. Covert channels were first defined by Butler W. Lampson [69] in 1973 as a communication channel that is not intended to transfer information. As with other communication channels, a covert channel usually includes a sender and a receiver [79]. These, for example, can be two malicious processes secretly communicating with each other using a shared physical resource [119]. Shared resources that can be used for building a covert channel include file system objects [69], input devices [97], network stacks/channels [11,25] and caches [32,37,73,76,77,83,88,114,117].

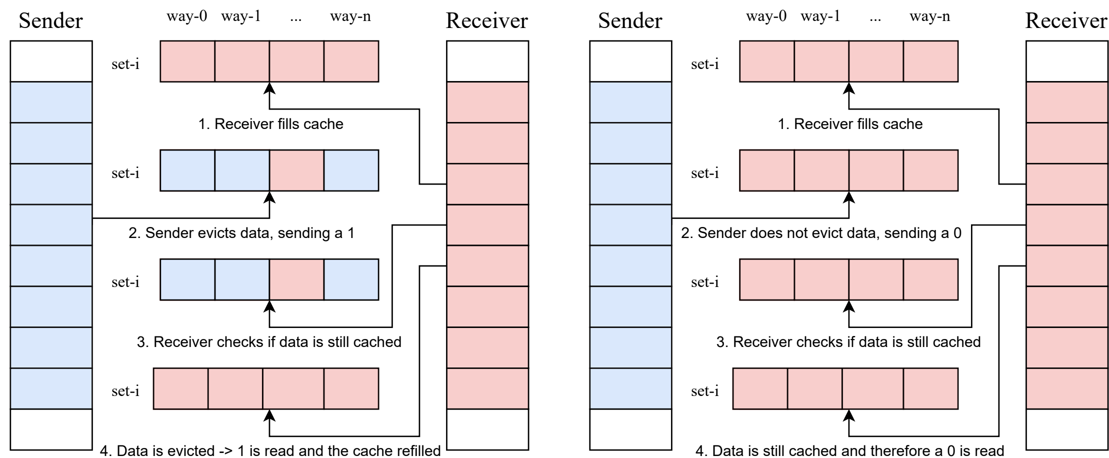
2.8.1 Cache Covert Channels

Cache covert channels can be used to let two processes, a sender and a receiver, communicate over the CPU cache [37]. This enables the communication between two isolated processes, not indended to communicate by the system [88]. As cache covert channels are usually noisy, some error detection and correction should be applied [77]. In the following, we will explain a simple 1-bit cache-based covert channel, using the Prime+Probe technique [76], illustrated in Figure 2.13:

1. Both the sender and the receiver agree on the cache sets used by the covert channel [76].
2. A timing protocol has to be used in order to coordinate writing by the sender and reading by the receiver [76].
3. The receiver fills the cache set [76]. (Prime)
4. If the sender wants to send a 1, the cache set is filled with data by the sender [76]. Filling the cache set evicts the data of the receiver out of the cache [76].
5. The receiver then probes the cache set [76]. If more cache hits than misses occur, the receiver deduces a 0 [76]. If more cache misses are recorded, the receiver assumes a 1 [76]. (Probe)
6. The receiver fills up the cache again, waiting for the sender to send the next bit [76]. (Prime)

Cache covert channels were first mentioned in 1992 by Hu [37], theoretically describing the transmission of data over a covert channel via cache. In 2005 Percival [83] introduced the first Prime+Probe-based cache covert channel on the L1 cache. Percival [83] estimated a capacity of 400 kilobytes per second using an "appropriate error-correcting code". Wang et al. [111] showed the possibility of cache covert channel between two virtual machines, despite state-of-the-art security measures. Ristenpart et al. [88] presented the first cache-based covert channels in a cloud environment in 2009. The reported bandwidth was approximately 0.2 bits per second between two virtual machines running on the same physical CPU on the Amazon EC2 cloud service [88]. Xu et al. [117] enhanced the cache covert channel approach by Ristenpart et al. [88], switching from the L1 cache to the L2 cache. They reported a capacity of 215 bits per second [117]. As these attacks utilize the L1 cache or L2 cache, sender and receiver were required to run on the same core [117].

The first cross-core covert channel was introduced by Wu et al. [114], using Prime+Probe. Different from previous cache covert channel approaches, the covert channel by Wu et al. [114] was built on the last-level cache instead of the L1 cache. Switching to the last level cache allows the sender and receiver of the covert channel to run on different cores, as long as they share the same CPU [114]. In 2015 Maurice et al. [76] improved the method proposed by Ristenpart et al. [88] by switching to the last-level cache, still using



(a) The sender sends a 1 by evicting data from the agreed cache set.

(b) The sender does not evict data and therefore sends a 0.

Figure 2.13: Illustration of a 1-bit covert channel using the Prime+Probe technique [77]. Sender and receiver agree on using a certain cache set `i` for communication.

Prime+Probe. They achieved a capacity of 1291 bits per second on a native setup and 751 bits per second between virtual machines [76]. In the same year, Lui et al. [73] showed that LLC-based covert channels could reach a capacity of up to 1000 kilobits per second using the Prime+Probe technique. In 2016, Gruss et al. [32] demonstrated the first covert channel utilizing the Flush+Reload and Flush+Flush techniques. The reported capacity for cross-core transmissions was 496 kilobyte per second [32]. However, the usage of Flush+Reload or Flush+Flush assumes shared memory between the sender and the receiver [32]. Later, Maurice et al. [77] presented an error-free Prime+Probe covert channel used to build an SSH connection between two virtual machines on Amazon EC2. The capacity achieved exceeded 360 kilobits per second [77].

Preventing cache covert channels can be achieved by preventing underlying cache side-channel attacks [77]. As cache side-channel attacks depend on an accurate timer, removing timing mechanisms or making them coarser are considered countermeasures [77]. Many countermeasures are based on adding noise, making it more difficult to achieve robust covert channels [24, 90, 121].

Chapter 3

Speculative Dereferencing Analysis

In this chapter, we will give an overview on speculative dereferencing and will analyze its properties. Therefore, we will first discuss and analyze the address-translation attack first introduced by Gruss et al. [31] in 2016. We will discuss the original attack explanation and show why the original attack description is erroneous. We will show that instead of the `prefetch` instruction [22] cited in the original paper [31], Spectre gadgets in the kernel are the root cause of the leakage. We will therefore call this speculative dereferencing. Based on these findings, we will locate an actual Spectre-BTB gadget in the kernel and identify it as the primary source of leakage on our test system. Based on this, we will discuss kernel Spectre gadgets in general and discuss preconditions for speculative execution.

For additional analysis on speculative dereferencing, we would refer to *Speculative Dereferencing: Reviving Foreshadow* by Schwarzl et al. [94], which was based on the findings of this thesis.

3.1 Address-Translation Attack

The address-translation attack by Gruss et al. [31] can be used by an attacker to find the direct-physical address $\overline{p}$ for an arbitrary virtual address p. As operating systems like Linux have a direct mapping of all physical addresses in the kernel virtual memory space [61], the address-translation attack can help an attacker to learn which virtual address is mapped to which physical address [31]. Furthermore, the attack can be used by an attacker to check if the virtual addresses p and a different virtual address q map to the same physical address $\overline{p}$ [31]. Information gathered by this attack, for example,

```
1  for (size_t i = 0; i < NUMBER_OF_TRIES; i++) {
2    // Step 1
3    flush(virtual addr);
4    // Step 2
5    for (size_t i = 0; i < 3; i++) {
6      prefetch(direct_phys_map_addr);
7      sched_yield()
8    }
9    // Step 3
10   access_time = reload(virtual addr);
11   if(access_time < CACHE_HIT_THRESHOLD) {
12     print("Cache␣hit");
13   else
14     print("Cache␣miss");
15 }
```

Figure 3.1: Example code for the 3 steps of an address-translation attack [31]. The prefetch step is repeated several times in order to increase the chance of the address being cached [31].

enables an attacker to bypass kernel and CPU security mechanisms like SMAP, SMEP, and KASLR [31].

In the original description of the address-translation attack, the attack works in 3 steps, based on a Flush+Reload attack [31]:

1. Flush user-space address p using the x86 `CLFLUSH` instruction [31].
2. Prefetch the inaccessible kernel-space address $\overline{p}$ [31].
3. Reload p and check if the data was cached by measuring the access time [31].

The attack assumes knowledge of the user-space virtual address p and the corresponding kernel-space virtual address of the direct mapping $\overline{p}$ [31]. Alternatively, an attacker can guess the direct mapping address $\overline{p}$ [31]. Figure 3.1 shows a small code example for the above-mentioned 3 steps of the address translation attack.

If the user-space address p and the kernel-space address $\overline{p}$ share the same physical address, step 2 will lead to the shared physical address of these two addresses being cached [31]. Thus, reload in step 3 will lead to a fast access time and, therefore, a cache hit with a high probability [31]. Usually, these 3 steps are repeated several times in order to maximize the possibility of detecting a cache hit [31].

The address-translation attack and the learning physical address information learned through it enable an attacker to defeat SMAP, SMEP, and KASLR [31,51]. Additionally,

```
1  ;%r14 contains the
      direct-physical
      address
2  callq 1080 <
      sched_yield@plt>
3  prefetchnta (%r14)
4  prefetcht2 (%r14)
5  callq 1080 <
      sched_yield@plt>
6  prefetchnta (%r14)
7  prefetcht2 (%r14)
8  callq 1080 <
      sched_yield@plt>
9  prefetchnta (%r14)
10 prefetcht2 (%r14)
```

(a) Disassembly of the prefetching loop of the original address-translation attack.

```
1  ;%r14 contains the
      direct-physical
      address
2  callq 1080 <
      sched_yield@plt>
3  nop
4  nop
5  callq 1080 <
      sched_yield@plt>
6  nop
7  nop
8  callq 1080 <
      sched_yield@plt>
9  nop
10 nop
```

(b) Disassembly of the prefetching loop with replaced prefetch instructions.

Figure 3.2: The assembler code of the prefetching component of the prefetch address-translation attack showed in Figure 3.1 [94]. Version (a) shows the original disassembly. In version (b), the prefetch instructions were replaced by `nop` [94].

Rowhammer attacks [64, 95] and side-channel attacks [77, 84] are re-enabled, reopening various attack vectors. Additionally, Gruss et al. [31] introduced the translation-level attack using the `prefetch` instruction for breaking KASLR by learning virtual address information [39,96]. In order to prevent the attack, the KAISER technique was introduced by Gruss et al. in 2007 [29, 30]. This mitigation was later implemented as KTPI [16] for Linux and KVAS [49] for Windows, theoretically making address-translation attacks impossible. However, due to an erroneous assumption in the original attack description, address-translation attacks are still possible with KAISER mitigations activated.

Gruss et al. [31] erroneously assumed the leakage was due to missing privilege checks of the `prefetch` instruction [22], therefore fetching arbitrary normally inaccessible privileged memory into the CPU cache. However, replacing the `prefetch` instructions with `nops` still shows leakage up to 10 cache fetches per second for a machine using an Intel i7-6500U running Linux Mint 19, kernel version `4.15.0-52-generic`. On an i7-8700K running Ubuntu 18.10, kernel version `4.16.0-55`, approximately 60 cache fetches per second were measured [94]. This shows that replacing the `prefetch` instructions from the prefetching loop with `nop`, as shown in Figure 3.2a, does not prevent the address-translation attack. We can therefore conclude, the leakage is not caused by missing privilege checks of the `prefetch` instructions.

```
1 while(1)
2 {
3   set_registers(address);
4   sched_yield();
5 }
```

Figure 3.3: First proof-of-concept code showing prefetching of addresses stored in registers. `set_registers` fills all general-purpose registers with the given address.

3.2 Locating the leakage source

In Section 3.1, we were able to show that the `prefetch` instructions are not the source of leakage used by the address-translation attack. However, we were able to observe that writing a kernel address into general-purpose registers sometimes leads to this address being cached, measuring up to 8 cache fetches a second, without explicitly accessing the chosen address. Additionally, we detected that newer Linux kernel versions do not show this behavior on our first testing system, running an Intel i7-6500U (Linux Mint 18, kernel version `4.18.0`) with all Meltdown and Spectre related countermeasures deactivated. Figure 3.3 shows one of the first PoC program where this behavior was observed. The PoC first writes a kernel address into all available general-purpose registers, followed by a call to the `sched_yield` syscall. First, we assumed the cause was some unknown optimization technology, prefetching registers. However, no indication for such an optimization mechanism was found in the Intel documentation.

We detected that a commit making small changes to the `do_syscall_64` general Linux syscall handler function almost eliminated cache hits on the attacked address [1]. First, we observed that, on our Intel i7-6500U system (Linux Mint 18), the number of cache fetches drastically decreases between the two major Linux kernel versions `v4.16` and `v4.17`. While on kernel versions `v4.16`, usually up to 8 fetches per second were measured, `v4.17` reduced the number of cache fetches down to about 1 fetch every 5 seconds. To narrow down possible code sections causing the prefetching, `git bisect` was used in order to detect commits that significantly reduce the number of cache fetches. The most significant drop in cache fetches was measured for a commit that changes the number of arguments of the `do_syscall_64` function, now directly passing the syscall number [1] additionally to the register data structure. Therefore, we were able to conclude that the leakage is caused somewhere in the syscall handler function and therefore caused by the `sched_yield` syscall in our proof-of-concept program, seen in Figure 3.3.

Putting the `lfence` instruction right before the actual syscall handler inside the `do_syscall_64` function reduces the number of cache fetches on our Intel i7-6500U system (kernel versions `v4.16`) from on average 8 cache fetches per second down to one

cache fetch per second. Therefore, we assumed that the leakage is likely to be caused by speculative execution. The instruction trace of the `do_syscall_64` and following functions further strengthens this assumption, as they show several indirect jump instructions as seen in Figure 3.4 for `do_syscall_64`. Additionally, not all registers are cleared on entering the kernel, still containing the kernel address previously filled into all general-purpose registers (see Figure 3.3).

We were able to trace one source of leakage, a Spectre-BTB gadget located in the syscall handler of `sched_yield` syscall. An indirect call `current->sched_class->yield_task` in the `sys_sched_yield` syscall handler mispredicts into the `put_prev_task_fair` function of the Linux scheduler. In this function, an uncleared and not overwritten register `%rsi` is dereferenced, accessing the victim's direct-physical map address [94]. We will call this behavior *speculative dereferencing*. Our updated test system, running an Intel i7-6500U (now on Linux Mint 19) and Linux kernel version `4.16`, showed around 21 fetches per minute. By putting the `lfence` instruction at the beginning of the `put_prev_task_fair` function, as seen in Figure 3.5, the number of fetches were be reduced by 50%, measuring 10 fetches per minute. As no indirect jumps exist in `put_prev_task_fair` and the function access registers that cause leakage on the used system, it is likely one source of leakage. For more information, Schwarzl et al. [94] describe in detail how the gadget was detected.

As the leakage was only reduced by 50% and considering the number of indirect jumps existing in the Linux kernel [58], we assume multiple possible gadgets located in other essential parts of the Linux kernel. These essential parts may include interrupt routines, syscall handler, and the scheduler.

```
1   0xffffffff810014a0:  push   rbp
2   0xffffffff810014a1:  mov    rdx,gs:0x14d00
3   0xffffffff810014aa:  mov    rbp,rsp
4   0xffffffff810014ad:  push   r12
5   0xffffffff810014af:  push   rbx
6   0xffffffff810014b0:  mov    rbx,rdi
7   0xffffffff810014b3:  mov    r8,[rdi+0x78]
8   0xffffffff810014b7:  sti
9   0xffffffff810014b8:  data16 xchg ax,ax
10  0xffffffff810014bb:  data16 xchg ax,ax
11  0xffffffff810014be:  mov    rax,[rdx]
12  0xffffffff810014c1:  test   eax,0x100801c1
13  0xffffffff810014c6:  jne    0xffffffff81001577
14  0xffffffff810014cc:  cmp    r8,0x14c
15  0xffffffff810014d3:  ja     0xffffffff8100150b
16  0xffffffff810014d5:  cmp    r8,0x14d
17  0xffffffff810014dc:  sbb    rax,rax
18  0xffffffff810014df:  and    r8,rax
19  0xffffffff810014e2:  mov    rcx,[rbx+0x38]
20  0xffffffff810014e6:  mov    rdx,[rbx+0x60]
21  0xffffffff810014ea:  mov    rax,[r8*8-0x7e5ffec0]
22  0xffffffff810014f2:  mov    rsi,[rbx+0x68]
23  0xffffffff810014f6:  mov    rdi,[rbx+0x70]
24  0xffffffff810014fa:  mov    r9,[rbx+0x40]
25  0xffffffff810014fe:  mov    r8,[rbx+0x48]
26  0xffffffff81001502:  call   0xffffffff81803000
27  ...
28  0xffffffff81803000:  jmp    rax
29  ...
```

Figure 3.4: This figure shows a part of the instruction trace of the `do_syscall_64` function in the Linux kernel. The trace was recorded on a virtual machine running a Linux `v4.16.18` kernel. First, the stack is prepared. After interrupts are enabled in line 8, line 12 and 13 checks if syscall tracing is activated. If not, line 14 and 15 test if the given syscall number is assigned to a valid syscall. Finally, the registers are prepared according to Figure 3.1, and `__x86_indirect_thunk_rax` is called in line 26. As retpoline is deactivated in this case, `__x86_indirect_thunk_rax` just consists of a jump to `rax`, where `rax` contains the address of the corresponding syscall handler for the called system call.

```
1  static void put_prev_task_fair(struct rq *rq, struct
     task_struct *prev)
2  {
3    asm volatile("lfence\n");
4    struct sched_entity *se = &prev->se;
5    struct cfs_rq *cfs_rq;
6
7    for_each_sched_entity(se) {
8      cfs_rq = cfs_rq_of(se);
9      put_prev_entity(cfs_rq, se);
10   }
11 }
```

Figure 3.5: This figure shows the `put_prev_task_fair` function used by the scheduler of the Linux kernel. In this function, the uncleared `%rsi` is dereferenced, as Schwarzl et al. [94] show. The `lfence` at the beginning of the functions prevents speculative execution.

3.3 Kernel Spectre Gadgets

The Spectre-BTB gadget found in Section 3.2 is located in one of the many syscall handlers of the Linux kernel. Additional to the detected Spectre-BTB gadget, we assume multiple possible gadgets in various syscall handlers and interrupt handlers, as we were able to observe a higher number of cache fetches by inducing a large number of interrupts and context switches during the attack. Any prefetch gadget that can be used for address-translation attacks can be exploited, including PHT, BTB, or RSB gadgets [94]. Figure 3.6 shows how, in general, direct-physical map addresses can be speculatively dereferenced and fetched into the cache. A syscall or interrupt happens after filling the general-purpose registers with our targeted direct-physical address. During the execution of the syscall handler or interrupt handler, an indirect jump speculatively executes a function dereferencing a normally unused and therefore uncleared register. As the register still contains the targeted direct-physical address, the address is fetched into the cache, which can be detected using, e.g., Flush+Reload.

For example, when calling a syscall on an x86_64 Linux system from user-space, the process will enter the kernel at the `entry_SYSCALL_64` function. The `entry_SYSCALL_64` function will prepare the stack and registers for calling the corresponding syscall handler for the called syscall. This preparation includes pushing several general-purpose registers onto the stack, except `rbx`, `rbp`, `r12`, and `r15`. After that, the general Linux syscall handler `do_syscall_64` is called. In this syscall handler, the given syscall number is retrieved from the stack and used to index the corresponding syscall handler function

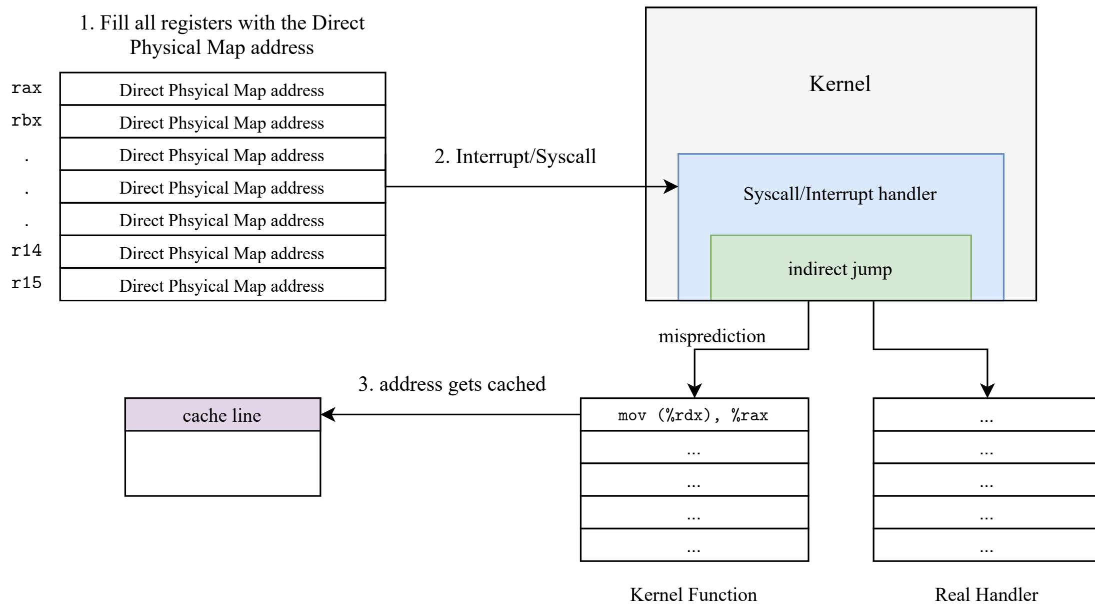


Figure 3.6: This figure shows how direct-physical map (DPM) addresses stored in registers are speculatively dereferenced inside the Linux kernel syscall handlers using, e.g., Spectre-BTB gadgets. The cached DPM can then be detected using, e.g., Flush+Reload.

contained in the Linux syscall page table `sys_call_table`. The assembler code for this behavior can be seen in Figure 3.4. During the execution of the general syscall handler `do_syscall_64` as well as many specific syscall handlers (e.g., `sys_sched_yield`), indirect jumps may be executed. Due to speculative execution, these indirect jumps might speculatively execute the wrong kernel function, dereferencing normally unused registers containing the targeted direct-physical map address.

Which registers are prefetched by the gadget vary across different kernel versions and systems. On a system running Ubuntu 18.10 with a kernel version of `4.18.0-17`, we observed cache fetches when the address was stored in registers `r12`, `r13`, or `r14`. On a Linux Mint 19.1 system running kernel version `4.15.0-99-generic`, registers `rdi` and `rdx` caused the leakage. Debian 8 with kernel version `4.19.28-2` and Kali Linux with `5.3.9-1kali1` needed the address stored in registers `r9` and `r10` [94]. To cover all possible register combinations, we recommend storing the victim address into nearly all available general-purpose registers.

Register	Description
`rax`	system call number
`rcx`	return address to user-space
`r11`	register flags
`rdi`	first argument
`rsi`	second argument
`rdx`	third argument
`r10`	fourth argument
`r8`	fifth argument
`r9`	sixth argument

Table 3.1: List of registers used when calling a system call [2]. In contrast to the usual x86_64 calling convention, `r10` is used for the fourth argument instead of `rcx`.

3.4 Speculative Dereferencing using Spectre

By writing arbitrary virtual addresses into CPU registers prior to calling syscalls or causing interrupts, these addresses were able to be speculative dereferenced and fetched into the cache by Spectre gadgets located in the syscall handler and interrupt handler of the Linux kernel. User-space addresses, as well as normally inaccessible kernel-space virtual addresses, can be used, depending on which transient execution mitigations are activated. This opens several possible attack vectors, including covert channel or the above-mentioned address-translation attack. Figure 3.7 shows a toy example code for speculative dereferencing. The program is basically separated into 5 steps, based on a Flush+Reload cache attack:

1. Flush the target virtual user-space address out of the cache.
2. Optional: Mistrain the Branch Target Buffer (BTB) by calling a syscall that uses certain registers.
3. Fill all available general-purpose CPU registers with either the user-space address or the corresponding kernel direct mapping address of the target.
4. Call a syscall.
5. Reload the virtual user-space address. If the access time was below a certain system depending threshold, the address was cached. Otherwise, a cache miss occurred.

Certain preconditions have to be met in order to successfully execute the attack. First, as we found a Spectre-BTB gadget as the main cause of leakage on our system, the `nospectre_v2` kernel boot parameter for Linux has to be set. The `nospectre_v2` parameter deactivates several kernel countermeasures against Spectre-BTB, also known as Spectre variant 2. These include, among others, Indirect Branch Restricted Speculation

```
1  for (size_t i = 0; i < NUMBER_OF_TRIES; i++) {
2    // Step 1
3    flush(virtual address);
4    // Step 2 (optional)
5    syscall()
6    // Step 3
7    fillRegisters(virtual address or kernel address);
8    // Step 4
9    syscall();
10   // Step 5
11   access_time = reload(virtual address);
12   if(access_time < CACHE_HIT_THRESHOLD) {
13     print("Cache␣hit");
14   else
15     print("Cache␣miss");
16 }
```

Figure 3.7: A toy example of the Register Prefetch PoC. The example uses a Flush+Reload cache attack to verify the address in the register was fetched and therefore cached. `fillRegisters` fills the CPU registers `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, and `r8` up to `r15` with the given address. In step 2, optionally, an additional syscall can be called for mistraining the Branch Target Buffer (BTB).

(IBRS), Indirect Branch Predictor Barrier (IBPB), and `retpoline`. As the Spectre-BTB gadgets are located in kernel-space, in order to prefetch user-space addresses, SMAP has to be deactivated using the `nosmap` kernel parameter. Deactivating KPTI using the `nopti` kernel parameter further improves the number of cache fetches, as we will evaluate in Chapter 4. If using kernel addresses, the attacker has to find the corresponding kernel direct-physical mapping address of the target user-space address.

Schwarzl et al. [94] showed that speculative dereferencing was the underlying root cause for Meltdown being able to leak data from the L3 (LLC) cache. Based on this information, they were able to mount a slightly modified Foreshadow (also Meltdown-P-L1 or L1TF) attack utilizing the L3 cache instead of the L1 cache [94]. This modification enables an attacker to circumvent common Foreshadow mitigations, for example, clearing the L1 data cache on `VMENTER` [94]. Foreshadow-L3 will be discussed in more detail in Chapter 6 [94].

Additionally, Schwarzl et al. [94] showed that improved Spectre hardware mitigations introduced in Ice Lake processors that supposedly replace the costly `retpoline` countermeasure do not affect speculative dereferencing. Therefore, they concluded that, in order to mitigate the attack, `retpoline` should stay enabled. However, on older kernel versions, even activating `retpoline` does not fully eliminate leakage [94]. There, full activating full Spectre-BTB mitigations would be recommended [94].

Chapter 4

Improving the number of fetches

In this chapter, we will evaluate how speculative dereferencing works under various conditions. Therefore, we will first measure the number of cache fetches influenced by a variety of syscalls on various systems. In the end, we will conclude our findings and evaluate how the number of cache fetches can be improved.

4.1 Measuring the leakage

In this section, we will first evaluate if mistraining using various syscalls improves the leakage rate. Second, we will take a look at how various syscalls improve or reduce the number of cache fetches. Furthermore, we will analyze if the leakage varies across systems and kernel versions. In the end, we will measure how using different kernel parameters influence the result.

4.1.1 syscalls

As described in Chapter 3, register prefetching using Spectre-BTB consists of 5 steps. First, the virtual address is flushed out of the cache, for example, using the `cflush` instruction. Optionally, a syscall can be called for mistraining the BTB for the second step. Next, all available general-purpose CPU registers are filled with the kernel direct mapping address of the target. In step 4, a syscall is called while registers are filled with our target address. Finally, we check if our address was cached, e.g., using Flush+Reload. Figure 3.7 shows a PoC code for this attack.

The syscall for step 4 of our speculative dereferencing attack highly influences the resulting leakage. As our attack exploits existing Spectre-BTB gadgets in the kernel, at

least one of these gadgets has to be reached during the execution of the syscall routine in kernel-space. However, as we estimate a huge number of possible gadgets in syscall and interrupt kernel routines, it is not feasible to search for an exhaustive list of possible gadgets that are reached by various syscalls. Furthermore, the position of these gadgets in the execution trace is essential, as the filled registers could be overwritten while executing kernel code.

Due to missing automation tools and the scale of the Linux kernel, an exhaustive search for all kernel gadgets potentially used for speculative dereferencing was not feasible. Therefore, we decided to use our PoC as described in Figure 3.7 to build a framework that measures the number of cache fetches for arbitrary syscalls. Based on this framework, we decided to test the leakage of 18 common syscalls in order to find suitable candidates for speculative dereferencing. The evaluated system was using an Intel i7-8700k using an Ubuntu 18.04 with the Linux kernel version of `4.4.153-generic`. On the system, Spectre v2 countermeasures were deactivated by using the `nospectre_v2` kernel parameter. Additionally, for the experiment, KPTI was deactivated using the `nopti` parameter. For the mistraining step (Step 2 in our PoC), we used the `stat` syscall. Furthermore, we set a static CPU frequency by setting the CPU governor to "performance" [52] and pinned the program to a CPU core in order to produce more reliable results. The experiment used $1,000$ samples per syscall, and each syscall was tested 4 times. The number of cache fetches was then calculated using the average number of hits per $1,000$ tries for each of the 4 rounds.

Table 4.1 shows the result of this experiment. The number of average cache fetches greatly differs between the syscall used. The highest amount of average hits was achieved using the `sched_yield` syscall, with an average of 374.25 cache fetches (STD: 151.495 cache fetches) for $1,000$ tries. On the other hand, the least working tested syscall was `getpriority`, with an average of 15.75 cache fetches (STD: 1.785 cache fetches) in $1,000$ tries. Compared to the second-best syscall, `sched_yield` caused a leak for about 38% of all tries, while the second-best syscall only averaged at about 25%. Furthermore, it is notable that none of the tested syscalls were immune to our attack. However, as only 18 out of more than 290 syscalls were tested, other existing syscalls might show higher amounts of cache fetches.

4.1.2 Mistraining using syscalls

Mistraining the BTB is an important part of running a Spectre-BTB attack [65]. By mistraining the BTB, an attacker is able to redirect indirect branch instructions, allowing sensitive information to be leaked by erroneous speculative execution [65]. However, as the Spectre gadgets are located in the Linux kernel for register prefetching, we are not able to directly mistrain from user-space. Therefore, we decided to mistrain the BTB by using extra syscalls to speculative dereference uncleared registers.

Syscall	Average cache fetches on $1,000$ tries
`sched_yield`	374.25
`getpid`	246.25
`stat`	243.25
`setxattr`	224.50
`mmap`	175.75
`ioperm`	158.50
`geteuid`	158.50
`access`	154.25
`getpgid`	128.75
`getsid`	128.50
`nanosleep`	127.00
`fadvise`	111.50
`ioctl`	99.75
`read`	98.00
`write`	92.00
`close`	19.00
`fsync`	17.25
`getpriority`	15.75

Table 4.1: A list of the best performing syscalls used for speculative dereferencing after filling registers, with PTI and Spectre v2 countermeasures deactivated. The result is presented in the average number of true positive cache fetches for a sample size of $1,000$. The evaluated system was running on an Intel i7-8700K using Ubuntu 18.04 with the Linux kernel version `4.4.143-generic`

By using extra syscalls before setting the registers for mistraining (see step 2 in the PoC code in Figure 3.7), we can influence the number of cache fetches. This is due to the BTB being mistrained into erroneously redirecting speculative execution to code areas, dereferencing registers not cleared of the following syscall. To analyze this effect, we, therefore, tried to mistrain our PoC with 291 different syscalls out of 313 currently available syscalls on a 64bit x86 Linux system [108].

In order to measure the influence of using a syscall for mistraining, we measured the number of true positive cache fetches for a sample size of 200,000. Therefore, we adapted our PoC, as described in Figure 3.7, to use one of the above-mentioned syscalls in Step 2. For comparison, we additionally measured the PoC using no mistraining under the same conditions. The PoC is then run with the `yield` syscall in Step 4 for each of the mistraining syscalls in Step 2. We then measured the number of true positive cache fetches. The adapted PoC is shown in Figure 4.1, where a python script automatically inserts the current syscall at the `#INSERT POINT` line. After filling the registers and calling `sched_yield` three times in order to trigger the speculative dereferencing, Flush+Reload is used in order to detect a cache hit, as explained in Figure 2.5. Nearly all 291 syscalls are tested using their standard library implementation (e.g., `setuid(0)`), as well as using the `syscall` interface for indirect system calls (e.g., `syscall(105,0)`, where 105 is the syscall number). We decided to skip some blocking syscalls like `pause` in order to minimize the overall execution time of the experiment. Additionally, syscalls terminating the program like `exit` were skipped. All in all, the experiment was repeated 10 times.

Table 4.2 shows the results of the experiment in the form of the average number of true positive cache fetches over 10 repetitions for 200,000 tries each. The experiment was conducted on a system utilizing an Intel i7-8700K running Ubuntu 18.04 with the Linux kernel version `4.4.143-generic`. In order to keep the table short, only the 20 best performing syscalls are shown. Running the experiment, we were able to show that the `readv` syscall, having an average of 13,766 cache fetches (STD: 8,940.183 cache fetches), showed the highest number of cache fetches per 200,000 tries. In comparison, omitting the mistraining syscall altogether, on average, 142 cache fetches (STD: 64.131 cache fetches) per 200,000 samples were measured. While more than 50 syscalls improved the number of cache fetches, the majority of syscalls showed a negative influence on the number of cache fetches. For many syscalls, using the indirect system call method (`syscall`) significantly improved the leakage measured, however for some syscalls like `readv`, the opposite was measured.

All in all, we were able to prove that using an extra syscall for mistraining the BTB can positively influence the rate of cache fetches for the register prefetch attack. The highest amount of cache fetches was achieved by utilizing the `readv` syscall for mistraining, whereby `sched_yield` was used after filling the registers. Analyzing the syscalls achieving high numbers of cache fetches, at least one pointer argument as the second or third parameter seems to be required in order to use the syscall for successfully mistraining the BTB.

```
for(int i = 0; i < 200000;i++)
{
  #INSERT POINT

  asm volatile("mov %%rax, %%rbx\n"
       "mov %%rax, %%rcx\n"
       "mov %%rax, %%rdx\n"
       "mov %%rax, %%rsi\n"
       "mov %%rax, %%rdi\n"
       "mov %%rax, %%r8\n"
       "mov %%rax, %%r9\n"
       "mov %%rax, %%r10\n"
       "mov %%rax, %%r11\n"
       "mov %%rax, %%r12\n"
       "mov %%rax, %%r13\n"
       "mov %%rax, %%r14\n"
       "mov %%rax, %%r15\n"
        :: "a"(phys) : "memory","%rbx","%rcx","%rdx","%rdi","%rsi","%r8","%r9","%r10","%r11","%r12","%r13","%r14","%r15");

       sched_yield();
       sched_yield();
       sched_yield();

       if(flushandreload(virt) < CACHE_HIT_MAX)
       {
           counter++;
       }
}
```

Figure 4.1: Implementation of the adapted PoC. At `#INSERT POINT`, one of the 291 tested syscalls will be added automatically using a python script. The experiment is repeated 200,000 times and the number of cache fetches is measured using Flush+Reload, as described in Figure 2.5.

Syscall	Parameters	Avg. cache fetches
readv	readv(0,NULL,0)	$13,766.3$
getcwd	syscall(79,NULL,0)	$7,344.7$
getcwd	getcwd(NULL,0)	$6,646.9$
readv	syscall(19,0,NULL,0)	$5,541.4$
mount	syscall(165,s_cbuf,...,(void*)s_cbuf)	$4,831.6$
getpeername	syscall(52,0,NULL,NULL)	$4,600$
getcwd	syscall(79,s_cbuf,s_ulong)	$4,365.8$
bind	syscall(49,0,NULL,0)	$3,680.6$
getcwd	getcwd(s_cbuf,s_ulong)	$3,619.3$
getpeername	syscall(52,s_fd,&s_ssockaddr,&s_int)	$3,589.3$
connect	syscall(42,s_fd,&s_ssockaddr,s_int)	$2,951.2$
getpeername	getpeername(0,NULL,NULL)	$2,822.4$
connect	syscall(42,0,NULL,0)	$2,776.4$
getsockname	syscall(51,0,NULL,NULL)	$2,623.4$
connect	connect(0,NULL,0)	$2,541.5$
bind	bind(s_fd,&s_ssockaddr,s_int)	$2,489.3$
getpeername	getpeername(s_fd,&s_ssockaddr,&s_int)	$2,348.9$
bind	syscall(49,s_fd,&s_ssockaddr,s_int)	$2,268.8$
connect	connect(s_fd,&s_ssockaddr,s_int)	$1,995.1$

Table 4.2: The 20 best performing syscalls for mistraining the BTB using sched_yield after filling registers. The result is presented in the average number of true positive cache fetches for a sample size of $200,000$. The evaluated system was running on an Intel i7-8700K using Ubuntu 18.04 with the Linux kernel version 4.4.143-generic

4.1.3 Difference between systems

As CPU hardware vulnerabilities like Spectre highly depend on the CPU architecture used [13, 65], the system itself can highly influence the leakage rate. Additionally, as the exploited Spectre-BTB gadgets used during speculative dereferencing are located in the kernel, the kernel version used by the system might be crucial to achieving a high rate of cache fetches. Therefore, in this section, we compare 4 systems using different CPU architectures and kernel versions. Furthermore, each system is tested with various syscalls for mistraining as well as syscalls used after filling registers. Table 4.3 shows a full list of systems evaluated in this experiment, including their CPU, CPU architecture, operating system, and kernel version.

Similar to previous experiments, we measured the performance on all systems using our PoC as described in Figure 3.7. We used either no mistraining or mistraining using the `sendto`, `geteuid`, or `stat` syscall in Step 2. These syscalls were chosen as each of the syscall requires a different number of parameters, additionally testing the influence of parameters on the attack. For the syscall called after filling the registers, we used about 20 different syscalls for Step 4, including `yield`, `fadvise`, and `stat`. However, due to space constraints, only 17 syscalls are shown in Table 4.4. The experiment is repeated several times, each time using a sample size of $1,000$ for each mistraining syscall and syscall combination. While on the evaluated Intel and AMD systems, the `nospectre_v2` kernel parameter was used, the ARM CPU did not support mitigations for Spectre-BTB, and therefore the deactivation of any mitigations was not required.

As seen in Figure 4.4, our speculative dereferencing attack was able to be successfully executed on all tested systems. Two systems showed a significant amount of cache fetches compared to the remaining evaluated systems. The first one was using an Intel i7-8700K and showed an average of 445.82 cache fetches (STD: 387.675 cache fetches) per $1,000$ tries using the `stat` syscall for mistraining. The other system, using an AMD Threadripper 1920X and mistraining using `stat`, showed on average 456.76 cache fetches (STD: 277.831 cache fetches). The lowest number of cache fetches was measured on the Intel 6500U ceiling at 28 cache fetches average (STD: 36.979 cache fetches) using `sendto` mistraining.

Compared to no mistraining, the Intel system achieved 31% more cache fetches using `stat` mistraining, whereas AMD cache fetches increased by about 72%. On other systems, other syscalls worked better. For example, `getuid` for the Intel 6500U only increased the number of cache fetches by less than 10%. On the ARM system, `sendto` showed the highest increase. All in all, using mistraining showed a significant increase in cache fetches on all evaluated systems.

On different systems, calling different syscalls after filling the registers seems to cause more cache fetches. While on the Intel 8700K system using `stat` led to cache fetches in nearly 100% of the tries, on the other evaluated system, it only showed an average amount of cache fetches. On the ARM Cortex-A75 system, nanosleep showed the highest number

of cache fetches, whereas, on the AMD system, the best syscall is highly dependent on the mistraining syscall. In general, the `yield` syscall showed an above-average number of cache fetches on all systems, mostly when skipping mistraining before filling the registers.

All in all, the experiment showed that the number of cache fetches rate highly depends on the system used. Although the Skylake, as well as the Coffee Lake system, nearly used the same kernel version, on the Coffee Lake system, 6 times more cache fetches were be recorded. We are unsure what exactly causes this difference. However, many factors might influence the number of cache fetches, including the CPU frequency, kernel version, or Linux distribution. Furthermore, which combination of mistraining syscall and syscall after filling the registers produces the highest number of cache fetches differs on every system evaluated. However, using `sched_yield` without mistraining generally produced an above-average number of cache fetches on all systems tested.

CPU	Architecture	Operating System	Kernel Version
Intel i7-6500U	Skylake	Linux Mint 19	`4.15.0-52-generic`
Intel i7-8700K	Coffee Lake	Ubuntu 18.04	`4.15.0-55-generic`
ARM Cortex-A57	ARMv8-A	Ubuntu 16.04.6	`4.4.38-tegra`
AMD Threadripper 1920X	Zen Core	Ubuntu 17.10	`4.13.0-46-generic`

Table 4.3: Evaluated systems with their CPUs, CPU architecture, operating systems, and kernel versions used by the operating system.

syscall ↓	**6500U**				**8700K**				**Threadripper 1920X**				**Cortex-A57**			
mistraining →	n/A	sendto	geteuid	stat	n/A	sendto	getuid	stat	n/A	sendto	getuid	stat	n/A	sendto	geteuid	stat
yield	242	5	279	8	845	24	10	403	986	360	470	530	621	292	323	466
close	0	10	0	0	129	277	23	372	330	92	566	310	253	400	231	31
getpriority	0	0	17	0	0	0	1	3	54	175	266	416	503	207	357	83
fadvise	105	96	24	0	8	1	229	334	70	6	77	136	292	107	382	207
stat	49	2	119	106	999	999	990	993	342	55	648	279	105	209	35	362
mmap	59	0	85	1	513	78	356	1	121	46	161	177	36	423	108	62
ioctl	3	0	0	0	592	308	340	871	79	490	738	749	453	425	927	120
access	9	74	85	13	589	0	11	126	357	211	648	271	359	309	223	193
nanosleep	57	71	11	0	159	72	925	983	357	616	812	813	847	707	579	635
getpid	44	4	44	0	18	11	416	4	100	2	192	661	1	0	0	1
fsync	1	0	15	63	59	425	890	845	201	87	570	744	297	460	87	62
geteuid	43	1	1	2	5	825	2	4	5	272	531	457	42	235	152	5
getpgid	6	0	91	140	3	588	64	649	115	524	347	870	49	201	10	23
getsid	0	53	51	0	7	452	560	812	594	114	714	892	125	100	29	10
setxattr	28	113	9	194	999	1000	999	989	126	71	5	47	116	409	268	6
write	15	21	12	19	839	52	192	51	619	61	360	359	418	296	329	64
read	139	26	33	122	5	119	396	139	47	10	12	54	408	651	584	132
Average	47.06	28.00	51.53	39.29	339.35	307.71	376.71	445.82	264.88	187.76	418.65	456.76	289.71	319.47	272.00	144.82

Table 4.4: Evaluation of 4 systems using CPUs by various manufactures using various syscalls for mistraining and after filling registers. The cells show the number of cache fetches for $1,000$ tries. In the last row, the average number of cache fetches for each mistraining syscall is calculated.

,

4.1.4 Kernel Parameters

As mentioned in Chapter 3, the Linux Spectre-BTB mitigations have to be deactivated in order to successfully speculative dereference an address. This is due to countermeasures like retpoline disabling Spectre-BTB by trapping speculative execution of indirect calls. In order to deactivate these mitigations, Linux supports multiple kernel parameters [58, 60]. Spectre variant 2 mitigations can be deactivated using the `nospectre_v2` parameter. This kernel parameter includes countermeasures like Indirect Branch Restricted Speculation (IBRS), Indirect Branch Prediction Barrier (IBPB), `retpoline`, and RSB filling.

However, these are only some of the many CPU vulnerability mitigations available for Linux. All in all, more than 10 different kernel parameters are available, enabling the user to activate or deactivate a couple of dozen mitigations [58], as described in chapter 2. These include mitigations against different variants of Spectre as well as Micro-architectural Data Sampling for attacks like Fallout and RIDL. As many of these countermeasures might influence our attack, we decided to deactivate all possible countermeasures one by one and evaluate our PoC, as we did in previous experiments. The `stat` was used for mistraining with `sched_yield` triggering the speculative dereferencing after filling the registers. The sample size for one round was 10,000,000, and the experiment was repeated 4 times. The evaluated system was running an Intel i7-6500U using Linux Mint 19 with the Linux kernel version `4.15.0-52-generic`.

Parameter	Avg. cache fetches on 10,000,000 tries
no parameter	62.35
`nopti`	99.93
`nospectre_v1`	69.26
`spectre_v2_user=off`	57.06
`spec_store_bypass_disable=off`	67.42
`l1tf=off`	61.84
`mds=off`	53.32
`tsx_async_abort=off`	59.13
`kvm.nx_huge_pages=off`	62.92

Table 4.5: Average number of cache fetches for 10,000,000 tries using different Meltdown and Spectre mitigation kernel parameters [58, 60]. For mistraining, `stat` was used with `sched_yield` being used after filling the registers. The `nospectre_v2` flag is always set, as turning of Specter variant 2 mitigations is a requirement for speculative dereferencing to work, as described in section 3.1. The system used was running on an Intel i7-6500U using Linux Mint 19 with the Linux kernel version `4.15.0-52-generic`

As Table 4.5 shows, out of all tested kernel boot parameters benchmarked, only `nopti` showed significant improvement on our tested system. While only using the `nospectre_v2`

flags averaged at about 62.35 cache fetches (STD: 13.412 cache fetches) on 10, 000, 000 tries, additionally deactivating KPTI using the `nopti` parameter improved the number of cache fetches to an average of 99.93 cache fetches (STD: 14.842 cache fetches) out of 10,000,000 tries. On the other hand, combining the rest of the tested kernel parameters with `nospectre_v2` only showed a minimal difference, averaging between 53 and 69 cache fetches (STD: 13-15 cache fetches).

As described in Chapter 2, the `nopti` kernel parameter deactivates Kernel page-table isolation (KPTI), also known as KAISER [30,57]. KPTI prevents Meltdown-type attacks by unmapping the kernel-space during execution in user-space. Only small parts of kernel memory, for example, kernel entry points from user-space, are always mapped. However, as we suspect most of our Spectre-BTB gadgets in kernel-space, there is a high chance that activating KPTI reduces our chance of speculative dereferencing. This is due to limiting speculative execution from inside kernel-space, as otherwise our gadgets would be mapped. With KPTI deactivated, however, this limitation is not present, enabling a wider range for speculative execution.

4.2 Improving the leakage

In this section, we conclude our findings from the last chapter in order to define methods to increase the number of cache fetches caused by speculative dereferencing of addresses in registers. We, therefore, looked at the POC described in chapter 3 (see Figure 3.7) and apply the software-based attack optimization strategies discovered in our experiments. Additionally, we looked at system and hardware influences on our attack. We were testing various CPU manufactures and types as well as multiple Linux distributions with different Linux kernels. Furthermore, we tested the influence of Linux command line parameters on cache fetches caused by our attack.

We showed that the number of cache fetches varies depending on the syscall used for mistraining and syscalls used after filling the registers. We showed that for mistraining, the `readv` showed promising results on the evaluated system. The number of cache fetches was increased from about 142 cache fetches without mistraining to up to 13, 766 cache fetches on average for 20, 000 tries when calling `readv` prior to filling the registers. Alternatively, the `getcwd` syscall showed promising results, averaging up to 7, 344 cache fetches. However, we were also able to exhibit that mistraining might negatively influence the number of cache fetches, depending on the system and syscall called after filling the registers.

For syscalls called after filling the registers, `sched_yield` showed the highest number of cache fetches on the evaluated system. We could measure about 374 cache fetches out of 1, 000 tries for `sched_yield`, followed by the `getpid` syscall averaging at 246 cache fetches. Additionally, calling the syscall multiple times after filling the registers seemed

to overall stabilize the number of cache fetches. However, as the list of syscalls tested was not exhaustive, future experiments might find a more suitable syscall, especially on different systems.

We were able to show that our attack can be conducted on Intel, AMD, and ARM CPUs, as long as they are susceptible to Spectre-BTB type attacks. However, we showed that although two systems use the same Linux kernel version, the number of cache fetches can differ by a large margin. Additionally, we showed that the leakage caused by syscalls used in speculative dereferencing differs depending on the system. On some systems, calling `stat` after filling registers showed the highest amount of cache fetches. On other systems, only a minimal number of cache fetches were recorded. However, some syscalls, for example, calling `sched_yield` after filling the CPU registers, showed an above-average number of cache fetches on all evaluated systems, as long as Spectre-BTB countermeasures are deactivated.

As our attack depends on Spectre-BTB gadgets located in kernel code, the susceptibility of the system to Spectre-BTB type attacks is mandatory for the attack to work. Therefore, the `nospectre_v2` Linux kernel boot parameter has to be set in order to deactivate countermeasures against Spectre v2, also known as Spectre-BTB. Additionally to this mandatory kernel parameter, deactivating KPTI using the `nopti` kernel boot parameter showed an increase in cache fetches of up to 60%.

All in all, which syscalls to use for mistraining as well as which syscalls to call after filling the registers depends on the system on which the attack is conducted. Additionally, the amount of cache fetches itself highly varies between systems. However, calling the `sched_yield` syscall after filling registers generally resulted in a high amount of cache fetches, especially when called multiple times. For mistraining, `readv` showed promising results, even though this may differ from system to system. Moreover, in general, the overall number of cache fetches highly differs between systems, even though they are running identical kernel versions. Finally, deactivating KPTI on Linux using the `nopti` kernel boot parameter showed an increase in cache fetches of up to 60%.

Chapter 5

Attack Case Studies

Based on our findings described in Chapter 4, we decided to implement several experiments based on speculative dereferencing using kernel Spectre gadgets. Therefore, in this chapter, we will describe and analyze two experiments conducted using our new attack. First, we will build a covert channel, as described in Chapter 3. We will explain the architecture of the covert channel, as well as how speculative dereferencing can be utilized for this purpose. Furthermore, the speed of the covert channel will be measured. Second, we will show how we can not only leak data residing in memory but any variable or value used by a program. Therefore, we will first explain our experiment setup and preconditions. Furthermore, we will discuss how the program works and examine some results.

5.1 Covert Channel

In this section, we will discuss the covert channel we constructed using speculative dereferencing. First of all, the concept of a covert channel will be described. Based on that, we will explain how speculative dereferencing can be utilized in order to construct a cache-based covert channel. Finally, we will compare the performance of the speculative dereferencing-based covert channel to covert channel exploiting other hardware vulnerabilities.

5.1.1 Description

Covert channels are communication channels between processes, usually not allowed to communicate with each other according to system specifications [69, 79]. Usually,

they often use unusual mechanisms originally not designed for data transfer. Therefore, covert channels are often hard to detect by a system's security mechanisms. Additionally, covert channels can be used for capacity evaluation for information leakage [84, 114]. We will use the built covert channel for evaluation and compare our attack to other similar side-channel attacks. The covert channel sets an upper bound for the rate of leakage of potential attacks utilizing speculative dereferencing.

As with other channels used for communicating, covert channels usually consist of a sender and a receiver, for example, two malicious programs. Covert channels often secretly communicate using shared physical resources [119], e.g., input devices [97], network channels [11, 25], and CPU caches [32, 37, 73, 76, 77, 83, 88, 114, 117]. Our covert channel using speculative dereferencing will be constructed as a cache-based covert channel.

Cache based covert channels allow communication between two processes running on a shared CPU [32,37,73,76,77,83,88,114,117]. Our covert channel utilizes the Flush+Reload technique as described in Chapter 2 for the receiving process, combined with the speculative dereferencing attack described in Chapter 3 for the sending process. Both processes run on the same logical core. The sender and receiver agree on a memory region used for the communication by sharing the identity address of that physical page mapped by the receiver. The channel uses 1 bit for communication and synchronizes using time frames based on the systems time stamp counter (TSC) register. Compared to other covert channels [31, 32] using Flush+Reload, our covert channel does not require shared memory or shared libraries.

The sender loads the payload and splits it up into bits, leading with a 1 to signal the start of the communication. For each bit equal to 1, the sender uses speculative dereferencing on the shared identity address in order to get the cache line of the memory region cached. In the case of a 0, no speculative dereferencing is used for this time frame. Each bit is sent for a previously agreed time frame based on the system's TSC register to provide synchronization with the receiver process. A pseudo-code version of the sender is shown in Figure 5.1. In order to improve the communication, the sender can wait for the start of a new time frame before transmitting the leading 1 bit at the beginning of the communication.

The receiver, on the other hand, uses the Flush+Reload technique to wait for cache hits. The memory region of the monitored address is mapped by the receiver process, which shares its identity address with the sending process. To differentiate between a 1 bit and 0 bit, the receiver counts the number of cache hits occurring during one time frame. If this number exceeds a certain threshold, it will be counted as 1. Otherwise, the bit will be recorded as 0. The threshold filters out any noise created by false-positive hits. The receiver will start recording the communication after receiving the preceding 1 bit. In the end, all the bits are combined to form the transmitted payload. A pseudo-code example of this routine is shown in the listing of Figure 5.2.

```
1  function send() {
2    // Load Payload and convert to bits, pretending a 1 as a start bit
3    payload = getPayloadInBits();
4
5    // Set identity address to agreed on memory region
6    // Used by speculative dereferencing
7    prefetch_addr = identity_addr;
8
9    // Get start of next time frame using the systems TSC register.
10   // Each time frame has a length of "TIME_FRAME"
11   time = rdtsc();
12   next = time + (TIME_FRAME - (time % TIME_FRAME));
13
14   // Each bit is transmitted for the duration of one time frame
15   i = 0;
16   while(1)
17   {
18     // Speculative dereferencing if payload bit is 1
19     if(payload[i] == 1)
20     {
21       prefetch();
22     }
23
24     // Check if next timeframe is reached
25     current = _rdtsc();
26     if(current >= next)
27     {
28       // Increase counter, stop when finished
29       i++;
30       if(i == PAYLOAD_BIT_SIZE)
31           break;
32
33       // Set start of next time frame
34       next = current + (TIME_AREA - (current % TIME_AREA));
35     }
36   }
37 }
```

Figure 5.1: Pseudo-code of the sender code routine used for the covert channel. The identity address is the kernel address of the physical page, as described in Figure 2.2. The prefetch function is defined as described in Figure 3.3.

```
1  function receiver()
2  {
3    // Map some memory and calculate the identity address
4    // This identity address is then shared with the sending process
5    addr = mapMemory();
6    identity_addr = getIdentityAddr(addr)
7
8    // Get start of next time frame using the systems TSC register.
9    // Each time frame has a length of "TIME_FRAME"
10   time = _rdtsc();
11   next = time + (TIME_FRAME - (time % TIME_FRAME));
12
13   data_in[PAYLOAD_BIT_SIZE] = {0}
14   i = 0;
15   while(1)
16   {
17     // Use Flush+Reload to listen for cache hits on the address.
18     delta = flushandreload(addr);
19     if(delta < CACHE_THRESHOLD)
20     {
21       // If cache hit is found, count up the hit counter for this bit
22       data_in[i] = data_in[i] + 1;
23     }
24
25     // Check if next timeframe is reached
26     size_t current = _rdtsc();
27     if(current >= next)
28     {
29       // As soon as we get the first hit, start reading bits
30       if(data_in[0] >= MIN_HITS)
31       {
32         // Increase i for every passed time frame, stop when finished
33         counter++;
34         if(counter == PAYLOAD_BIT_SIZE)
35           break;
36       }
37       next = current + (TIME_AREA - (current % TIME_AREA));
38     }
39   }
40
41   // Set bits where the number of hits exceeded the threshold.
42   // Combine bits to form the final payload.
43   payload = combineCheckThreshold(data_in, MIN_HITS);
44   writePayload(payload);
45 }
```

Figure 5.2: Pseudo-code of the receiver code routine used for the covert channel. The identity address is the kernel address of the physical page, as described in Figure 2.2. Flush+Reload is defined as described in Figure 2.5.

5.1.2 Result

We evaluated the covert channel on a test system using an Intel i7-6500U, running Linux Mint 19 with the kernel version `4.15.0-52-generic`. A random message of 1280 bytes was transmitted between two processes running on the same system, on unique CPU cores. In order to improve the leakage, in addition to the required deactivation of the Spectre v2 countermeasures using `nospectre_v2`, KPTI was deactivated using the `nopti` Linux boot parameter. Further, no mistraining was used as well as the `pthread_yield()` after filling the registers.

In our test setup, randomly generated messages of 1, 280 bytes were transmitted between a sending process and a receiving process. Both processes are the implementations of communication partners, as shown in Figure 5.1 and Figure 5.2. We repeated the experiment 50 times, generating a new random message every iteration. Additionally, during all transmissions, additional interrupts were caused by using the Linux `ls` tool. These interrupts were used due to our observation of an increase in leakage, proportional to the number of interrupts the evaluated system handles.

All in all, we were able to show a transmission rate of up to 30 bit/s (STD: 0.00618 bit/s) for a 1, 280 byte payload. However, at this transmission rate, the average error rate of the covert channel was up to 1% (STD: 0.8%). By increasing the time frames used for each bit, we were able to lower the error rate in the cost of transmission speed. By doubling the time frame, we were able to reduce the average error rate down to an acceptable 0.01% (STD: 0.0078%) while achieving a transmission rate of up to 15 bit/s (STD: 0.00521 bit/s). To lower the error rate to 0% (STD: 0.0001%) in the majority of transmissions, the transmission rate had to be reduced down to 6 bit/s (STD: 0.00119 bit/s). All in all, the measured transmission rate of the speculative dereferencing-based covert channel is overall lower than for other cache-based covert channels that do not require shared memory. While Maurice et al. [77] presented an optimized and error-free covert channel achieving up to 45 kb/s, other covert channels showed similar transmission rates and similar error rates to our speculative dereferencing-based covert channel [32,37,73,76,77, 83,88,114,117]. Many cache-based covert channels showed transmission rates between 10 bit/s and 100 bit/s, with error rates between 1% and 6% [32,37,73,76,77,83,88,114,117].

In order to achieve a higher transmission rate, error-correction methods can be used instead of increasing the time frame [77] in order to achieve error-free transmission. Furthermore, as we showed in Table 4.4 in Chapter 4, the rate of leakage heavily depends on the system evaluated. Therefore, the transmission rate can be increased by running the covert channel on a different test system.

5.2 Dereference Trap (Value Leak)

In this section, we will discuss our Dereference Trap experiment. Using the Dereference Trap, we are able to leak actual data using speculative dereferencing. We first discuss our experiment setup and preconditions. As a next step, we will explain the attack we want to conduct in detail. Furthermore, we will examine the implementation and why the attack works. Finally, we will examine the output of an execution and the results we gathered.

5.2.1 Description

As mentioned in previous chapters and experiments, speculative dereferencing can be utilized to dereference arbitrary user-space and kernel-space addresses. However, in this experiment, we will show that additional to addresses, an attacker can leak nearly any arbitrary value or variable used in the program using Dereference Trap. We can use the Dereference Trap to leak values from user-space, kernel-space, and even SGX, as Schwarzl et al. [94] showed.

Dereference Trap works by ensuring that as much virtual address space as possible is mapped by the application. Using speculative dereferencing on a secret contained in a register, the corresponding virtual address of the application will be cached. However, as the virtual address space is huge, mapping a unique physical page to each virtual page to check each address with e.g., Flush+Reload is infeasible [91]. Therefore, we decided to only use 2 pages, each page mapping to one half of the targeted address area. For example, for 32 bit secrets, each half has 2^{10} mappings per physical page [94]. Each half is then scanned using Flush+Reload for each of the 64 cache lines of the page. When cache fetches in one half are detected, this half is split up further. Moreover, the two physical maps are unmapped and mapped to the new split up address area. This process is repeated until only one page is left. Finally, we learn that the value of the secret is within the address boundaries of the page. However, certain preconditions have to be fulfilled.

First, as we utilize Spectre gadgets residing inside the kernel, the value we want to leak has to be filled into CPU registers prior to calling a syscall. Additionally, a value can only be leaked if the targeted value is high enough, as it has to fall into the region of mappable virtual address space. As the majority of leaks experienced on our test system were generated by a Spectre-BTB gadget, the `nospectre_v2` parameter has to be set as the Linux Spectre v2 countermeasures prevent Spectre-BTB. Additionally, as in our experiment, we use user-space addresses, SMAP has to be deactivated using the `nosmap` kernel parameter. Moreover, we use the `nopti` kernel command line parameter to deactivate KPTI, as KPTI significantly reduces the number of cache fetches on our

test system. In order to successfully conduct the attack on user-space addresses, at least the `nospectre_v2` and `nosmap` parameters are mandatory.

In order to map the memory regions, the first parameter of the `mmap` syscall [56] and shared memory mapping are utilized. Shared memory is a memory that can be accessed, mapped, and changed by multiple processes. On Linux, this is realized by creating shared memory objects on the file system that can be mapped using `mmap`. The first parameter of `mmap`, on the other hand, enables the caller to hint the kernel as to which address in the virtual address space the memory should be mapped. However, the kernel is not required to map the memory at the given address. This can be due to the given address being lower than a system set threshold (`/proc/sys/vm/mmap_min_addr`) [56] or the kernel not being able to map contiguous memory for the given size from the given address onwards. However, in the case of a successful `mmap` for the desired address region, an attacker can locate the value to be leaked.

Due to our precondition of the value residing in registers while calling a syscall, the address equal to the value will be cached using speculative dereferencing. Therefore, by using Flush+Reload on the allocated memory region that contains the value as an address, a cache hit can be observed at this address. In order to learn the leaked value more efficiently, we utilized divide and conquer. Therefore, we first divide the targeted address region into two parts. Each part is mapped to a separate shared memory object. Flush+Reload is then used on all cache lines (64 bytes) of every page (4096 bytes) for both memory halves. Cache hits on the address equal to the targeted value and sometimes hits on nearby addresses are detected. The half where cache hits were detected is then further divided and scanned for cache hits. This is continued until only one page causing page hits remains. The start and end addresses of this page act as a boundary for the targeted value. Figure 5.3 shows this method for a memory area consisting of 16 pages.

Figure 5.4 shows the pseudo-code for the attack described in Figure 5.3. First, we assume an address range that we suspect to contain the targeted value. Second, memory is mapped for the assumed address range. In the next step, each cache line of 64 bytes of every 4096-byte page is checked using Flush+Reload, tracing cache hits. This step is repeated multiple times. Mapped memory is split up in the middle and checked for hits separately. These hits are accumulated for each half. After checking all cache lines, the memory half showing more cache hits is chosen as the new assumed address range in the final step. These steps are repeated until only one page is left, defining the boundary of the leaked value.

5.2.2 Result

Figure 5.5 shows the result of a value leaking attack using speculative dereferencing. For the conducted experiment, the value area was estimated between `0x50000000` and

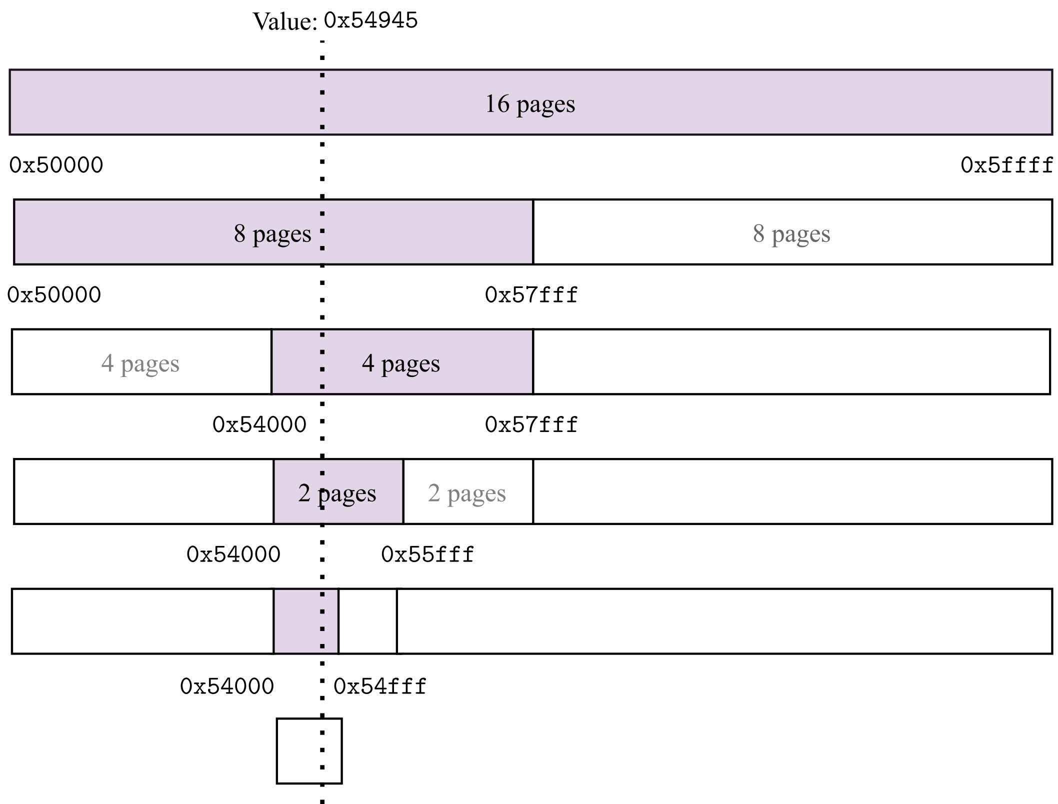


Figure 5.3: Visualization of finding the target value (`0x54945`) using decide and conquer. The estimated boundaries for the target value are `0x050000` and `0x5ffff`. Used as an address region, the resulting memory can be mapped on 16 pages. At every step, using divide and conquer, the half resulting in cache hits are chosen. In the end, one page remains, setting the boundary of the target value between `0x54000` and `0x54fff`.

```
1  // Target value is assumed between addr_start and addr_end
2  range = addr_end - addr_start;
3  // Loop until only one page left
4  while(range > PAGE_SIZE)
5  {
6    // Map the memory area
7    nr_mappings = (area / PAGE_SIZE);
8    mappings[nr_mappings];
9    map(mappings, nr_mappings);
10
11   // Flush+Reload on each half of the mapping, record cache hits
12   half[2] = {0};
13   for(i = 0; i < nr_mappings/2; i++) {
14     for(j = 0; j < NUMBER_OF_CACHELINES; j++) {
15       addr = mappings[i] + j * CACHE_LINE_SIZE;
16       for(k = 0; k < NR_TRIES; k++) {
17         if(flush_reload(addr) < CACHE_HIT_MAX) {
18           half[0]++;
19         }
20         flush_page(mappings[i]);
21       }
22     }
23   }
24   for(i = nr_mappings/2; i < nr_mappings; i++) {
25     for(j = 0; j < NUMBER_OF_CACHELINES; j++) {
26       addr = mappings[i] + j * CACHE_LINE_SIZE;
27       for(k = 0; k < TRIES; k++) {
28         if(flush_reload(addr) < CACHE_HIT_MAX) {
29           half[1]++;
30         }
31         flush_page(mappings[i]);
32       }
33     }
34   }
35
36   // Check which half recorded more hits
37   if(half[0] > half[1])
38     addr_end = addr_start + area / 2;
39   else
40     addr_start = addr_start + area / 2;
41
42   // Unmap and calculate new address range
43   unmap(mappings);
44   range = addr_end - addr_start;
45 }
```

Figure 5.4: Pseudo-code for Figure 5.3. After mapping the memory, each half of the address area is checked for cache hits using Flush+Reload. In the end, the half with the most cache hits is chosen as the new possible address area. After unmapping the memory, the loop is repeated with the new, reduced address range.

```
1  Attack value 0x50004945
2  0x50000000 - 0x5000ffff, 16 mappings
3  Hit in first half (3/0)
4  0x50000000 - 0x50007fff, 8 mappings
5  Hit in second half (0/1)
6  0x50004000 - 0x50007fff, 4 mappings
7  Hit in first half (3/0)
8  0x50004000 - 0x50005fff, 2 mappings
9  Hit in first half (1/0)
10 Variable in area from 0x50004000 to 0x50005000.
```

Figure 5.5: Result of a Test run of an implementation based on the pseudo-code in Figure 5.4. As in the illustration of the value leak in Figure 5.3, the estimated value and, therefore, address area consists of 16 pages. Each iteration was repeated $8 * 1024$ times.

`0x5000ffff`, mapping 16 pages of shared memory. For this experiment, the target value was set to `0x50004945` in order to follow the illustration seen in Figure 5.3. Moreover, both prefetching the value using speculative dereferencing and checking for cache hits in the mapped memory are conducted in the same process. The PoC is based on the pseudo-code seen in Figure 5.4. For this experiment run, every Flush+Reload done on any cache line is repeated $8 * 1024$ times, increasing the chance of detecting cache hits on the cost of execution time.

As the estimated value region can be covered by an address region of only 16 pages, the experiment can be finished in only 4 iterations. Each iteration further decreases the possible value boundaries. As the attacked value is set to `0x50004945`, the first iteration showed cache hits in the first half of the address region between `0x50000000` and `0x5000ffff`, setting the upper boundary to `0x50007fff`. For the second iteration, only 8 pages have to be mapped. Cache hits were detected in the second half of the address area, setting the lower boundary to `0x50004000`. After two additional iterations, only one page showing cache hits remains, setting the final limits for the value (set at `0x50004945`) between `0x50004000` and `0x50005000`. All in all, the experiment showed an execution time of around 15 seconds on an Intel i7-6500U, running Linux Mint 19 with the kernel version `4.15.0-52-generic`.

Chapter 6

Additional Work

In this chapter, we will discuss additional work and experiments that were conducted based on the findings of this thesis. In the following sections, we will look at the work presented in *Speculative Dereferencing of Registers: Reviving Foreshadow* by Schwarzl et al. [94]. First, we will discuss how speculative dereferencing can be used in virtual machines, reenabling Foreshadow type attacks despite recommended Foreshadow mitigations being activated. Next, we will examine how speculative dereferencing can be used to leak from SGX registers. Finally, we will mention a Javascript-based speculative dereferencing attack presented by Schwarzl et al. [94].

6.1 Speculative Dereferencing in Virtual Machines

Schwarzl et al. [94] showed that speculative dereferencing could successfully be used to mount an end-to-end attack from a KVM virtual-machine guest on a Linux host. Possible Spectre gadgets located in interrupt or hypercall routines may be used in order to fetch arbitrary host memory from a virtual-machine guest, in the case of the CPU misspeculating into one of these gadgets [94]. This fetched memory can then be retrieved using Foreshadow [94]. Therefore, Schwarzl et al. [94] conclude that, by using speculative dereferencing, circumvention of recommended Foreshadow countermeasures [54] is possible as long as Specter-BTB mitigations are deactivated and gadgets can be exploited in the interrupt handler or the hypercall routines of the host [94]. Additionally to KVM, kernel prefetching gadgets in combination with Foreshadow can also be exploited on Xen [94, 100, 116].

Schwarzl et al. [94] presented a successful end-to-end Foreshadow attack based on speculative dereferencing, abusing a Spectre-BTB gadget in interrupt routines of the Linux host. The Linux guest was virtualized utilizing `qemu` using KVM as a backend [94]. Foreshadow mitigations were activated; however, Spectre-BTB mitigations were incom-

plete [94]. During the attack, the guest repeatedly fills registers with a host's direct-physical-map address followed by a `sched_yield` syscall [94]. On the host, the cached address was then detected using Flush+Reload [94, 120].

Schwarzl et al. [94] recorded up to 25 fetches per minute for this speculative dereferencing-based Foreshadow attack using interrupts. However, even though speculative dereferencing using hypercalls is theoretically possible, no gadgets in the KVM hypercall routines were able to be detected and exploited [94]. All in all, Schwarzl et al. [94] concluded that recommended Foreshadow mitigations are not sufficient for preventing Foreshadow when no full Spectre-BTB protection is activated.

6.2 Speculative Dereferencing inside SGX enclaves

Intel Software Guard Extensions (SGX) are security-related instructions added on top of the x86 architecture, enabling the creation of so-called enclaves [41, 42]. These enclaves can be utilized to run trusted code and contain sensitive data. Access to these enclaves is restricted on a hardware level, even protecting the enclave content from access by compromised operating systems and hypervisors. Data inside the enclave is only accessible from the enclave code within. However, the enclave has access to the full virtual memory of the host application.

In order to leak data from SGX enclaves using speculative dereferencing, Schwarzl et al. [94] defined the *Dereference Trap* method. Possible Spectre-BTB gadgets inside an enclave's code are utilized to fetch arbitrary memory mapped by the enclave. The secret can then be detected by an attacker checking the virtual address space of the system for possible cache hits. In order to do this efficiently, divide and conquer, similar to the value leak attack of Chapter 5 can be used. Schwarzl et al. [94] reported that, by using *Dereference Trap*, the attack successfully recovered a 32-bit value stored in a 64-bit register in under 16 minutes.

6.3 Speculative Dereferencing in Javascript

While the previous attacks require native running code, Schwarzl et al. [94] presented an attack leaking physical addresses within a JavaScript Context. By using WebAssembly, an attacker can fill 64-bit registers with an attacker-controlled value or address. With this attack it is possible to leak the direct-physical-map address of any arbitrary JavaScript variable [94]. In order to achieve this, first, the direct-physical-map address is guessed and fetched using speculative execution. If Evict+Reload on the target variable yields a cache hit, the guessed direct-physical-map address was correct. As system calls can

not directly be called using Javascript, in order to trigger speculative dereferencing, the code is continuously interrupted, e.g., using disk I/O operations.

While this attack works fine on stand-alone Javascript engines like v8 with up to 20 speculative fetches a second, using a real unmodified Firefox browser reduces this number to a maximum of 1 fetch per minute [94]. Schwarzl et al. [94] explained this by not all registers being used by the browser. Furthermore, some registers can be used by the browser, overwriting the direct-physical-map address of the targeted variable.

Chapter 7

Conclusion

In this thesis, we showed that the original analysis of the address-translation attack was erroneous [31]. We analyzed the attack and showed that instead of prefetch instructions, speculative execution in kernel code causes the prefetching effect utilized in address-translation attacks [13,65,72]. Spectre gadgets in the syscall and interrupt routines of the Linux kernel lead to speculative dereferencing of user-filled general-purpose registers [65]. We were able to locate one Spectre-BTB [13] gadget causing cache fetches in the syscall handler of the `sched_yield` syscall [94].

Based on these findings, we showed that even on current Linux kernel versions and activated address-translation attack countermeasures like KAISER, speculative dereferencing is possible [30, 31]. We run the attack on various systems using different Linux distributions, various kernel versions, and various CPU generations. We measured the number of cache fetches of each run and compared the different test systems. Additionally, we evaluated the influence of various transient execution countermeasures on our attack. Finally, we compared the performance of various syscalls used for speculative dereferencing in order to optimize the number of cache fetches.

We conducted two case studies. We constructed a speculative dereferencing-based covert channel and compared the performance to other cache-based covert channels [32, 37, 73, 76, 77, 83, 88, 114, 117]. Furthermore, we presented the dereference trap technique, allowing an attacker to directly leak data from registers using speculative dereferencing. No encoding steps are needed to leak data from user programs, the kernel, or even SGX.

We showed additional experiments conducted by Schwarzl et al. [94], which are based on the findings of this thesis. We talked about how speculative dereferencing enables Foreshadow despite activated mitigations [94]. Additionally, we covered the usage of the dereference trap to leak data from SGX enclaves [94]. Finally, we talked about the possibility of running speculative dereferencing-based attacks in Javascript [94].

List of Figures

2.1 The virtual address space of a process. [104] 4
2.2 On Linux and OSX, physical memory is mapped twice, once as a kernel or a user page and once as a 1:1 mapping [61, 71]. 4
2.3 2-way set-associative cache with 8 cache lines in 4 sets, 2 lines per set. On access, the set is chosen, and the tags of the lines are compared with the tag of the address. The same memory location is always cached in the same cache set. [35] 7
2.4 Illustration of the level 1, level 2, and level 3 cache on a multi-core processor [46]. 8
2.5 A Flush+Reload attack illustrated. After measuring the access time, the attacker learns if the shared data was accessed [34, 120]. 10
2.6 This diagram is a visualization of the histogram provided by the Cache Template Attack calibration tool [33]. The optimal threshold in this example would be around 250. High access time for cache hits might be caused by scheduling. 12
2.7 The virtual address space of a process before and after applying the KAISER patch. [104] 13
2.8 Code example for instructions being split up into µOPs [23]. Code adapted from *The microarchitecture of Intel, AMD, and VIA CPUs* by Agner Fog [23]. 14
2.9 Classification tree of transient execution attacks [13, 28]. Split up into Meltdown-type and Spectre-type attacks [13, 28]. Based on a graph by Canella et al. [13, 28]. 15
2.10 Toy example of the Meltdown attack [72]. A kernel address is dereferenced, and the result is used to index an array [72]. Due to speculative execution, the indexed part of the array might be prefeched before the memory access permission check can detect invalid access to a kernel address [72]. 16
2.11 Toy example of the Spectre attack [65]. 18

2.12 Retpoline exchanges the jump instruction of (a) to the sequence seen in (b). First, there is a direct call to `load_label` [107]. The RSB entry after that call leads to `capture_ret_spec`. In `load_label`, the target is pushed onto the stack and returned to using `ret`, while speculative execution based on the RSB is trapped inside the `capture_ret_spec` loop [107]. . . 23
2.13 Illustration of a 1-bit covert channel using the Prime+Probe technique [77]. Sender and receiver agree on using a certain cache set `i` for communication. 28

3.1 Example code for the 3 steps of an address-translation attack [31]. The prefetch step is repeated several times in order to increase the chance of the address being cached [31]. 30
3.2 The assembler code of the prefetching component of the prefetch address-translation attack showed in Figure 3.1 [94]. Version (a) shows the original disassembly. In version (b), the prefetch instructions were replaced by `nop` [94]. 31
3.3 First proof-of-concept code showing prefetching of addresses stored in registers. `set_registers` fills all general-purpose registers with the given address. 32
3.4 This figure shows a part of the instruction trace of the `do_syscall_64` function in the Linux kernel. The trace was recorded on a virtual machine running a Linux `v4.16.18` kernel. First, the stack is prepared. After interrupts are enabled in line 8, line 12 and 13 checks if syscall tracing is activated. If not, line 14 and 15 test if the given syscall number is assigned to a valid syscall. Finally, the registers are prepared according to Figure 3.1, and `__x86_indirect_thunk_rax` is called in line 26. As retpoline is deactivated in this case, `__x86_indirect_thunk_rax` just consists of a jump to `rax`, where `rax` contains the address of the corresponding syscall handler for the called system call. 34
3.5 This figure shows the `put_prev_task_fair` function used by the scheduler of the Linux kernel. In this function, the uncleared `%rsi` is dereferenced, as Schwarzl et al. [94] show. The `lfence` at the beginning of the functions prevents speculative execution. 35
3.6 This figure shows how direct-physical map (DPM) addresses stored in registers are speculatively dereferenced inside the Linux kernel syscall handlers using, e.g., Spectre-BTB gadgets. The cached DPM can then be detected using, e.g., Flush+Reload. 36
3.7 A toy example of the Register Prefetch PoC. The example uses a Flush+Reload cache attack to verify the address in the register was fetched and therefore cached. `fillRegisters` fills the CPU registers `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, and `r8` up to `r15` with the given address. In step 2, optionally, an additional syscall can be called for mistraining the Branch Target Buffer (BTB). 38

4.1 Implementation of the adapted PoC. At `#INSERT POINT`, one of the 291 tested syscalls will be added automatically using a python script. The experiment is repeated 200,000 times and the number of cache fetches is measured using Flush+Reload, as described in Figure 2.5. 44

5.1 Pseudo-code of the sender code routine used for the covert channel. The identity address is the kernel address of the physical page, as described in Figure 2.2. The prefetch function is defined as described in Figure 3.3. . . 53
5.2 Pseudo-code of the receiver code routine used for the covert channel. The identity address is the kernel address of the physical page, as described in Figure 2.2. Flush+Reload is defined as described in Figure 2.5. 54
5.3 Visualization of finding the target value (`0x54945`) using decide and conquer. The estimated boundaries for the target value are `0x050000` and `0x5ffff`. Used as an address region, the resulting memory can be mapped on 16 pages. At every step, using divide and conquer, the half resulting in cache hits are chosen. In the end, one page remains, setting the boundary of the target value between `0x54000` and `0x54fff`. 58
5.4 Pseudo-code for Figure 5.3. After mapping the memory, each half of the address area is checked for cache hits using Flush+Reload. In the end, the half with the most cache hits is chosen as the new possible address area. After unmapping the memory, the loop is repeated with the new, reduced address range. 59
5.5 Result of a Test run of an implementation based on the pseudo-code in Figure 5.4. As in the illustration of the value leak in Figure 5.3, the estimated value and, therefore, address area consists of 16 pages. Each iteration was repeated $8 * 1024$ times. 60

List of Tables

3.1 List of registers used when calling a system call [2]. In contrast to the usual x86_64 calling convention, `r10` is used for the fourth argument instead of `rcx`. 37

4.1 A list of the best performing syscalls used for speculative dereferencing after filling registers, with PTI and Spectre v2 countermeasures deactivated. The result is presented in the average number of true positive cache fetches for a sample size of 1,000. The evaluated system was running on an Intel i7-8700K using Ubuntu 18.04 with the Linux kernel version `4.4.143-generic` 42

4.2 The 20 best performing syscalls for mistraining the BTB using `sched_yield` after filling registers. The result is presented in the average number of true positive cache fetches for a sample size of 200,000. The evaluated system was running on an Intel i7-8700K using Ubuntu 18.04 with the Linux kernel version `4.4.143-generic` 45

4.3 Evaluated systems with their CPUs, CPU architecture, operating systems, and kernel versions used by the operating system. 47

4.4 Evaluation of 4 systems using CPUs by various manufactures using various syscalls for mistraining and after filling registers. The cells show the number of cache fetches for 1,000 tries. In the last row, the average number of cache fetches for each mistraining syscall is calculated. 47

4.5 Average number of cache fetches for 10,000,000 tries using different Meltdown and Spectre mitigation kernel parameters [58,60]. For mistraining, `stat` was used with `sched_yield` being used after filling the registers. The `nospectre_v2` flag is always set, as turning of Specter variant 2 mitigations is a requirement for speculative dereferencing to work, as described in section 3.1. The system used was running on an Intel i7-6500U using Linux Mint 19 with the Linux kernel version `4.15.0-52-generic` 48

Bibliography

[1] 0xax. Ingo Molnar. https://git.kernel.org/pub/scm/linux/kernel/git/tip/tip.git/commit/?id=dfe64506c01e57159a4c550fe537c13a317ff01b, 2018. Accessed: Thu, 15 August 2020 15:45:00 +0100.

[2] 0xax. Linux Inside. https://0xax.gitbooks.io/linux-insides/content/, 2020. Accessed: Thu, 14 August 2020 15:00:00 +0100.

[3] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-Flow Integrity. In *CCS* (2005).

[4] AMD. Software Optimization Guide for AMD Family 17h Processors. https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf, 2017. Accessed: Tue, 29 October 2019 18:25:00 +0100.

[5] AMD. Software techniques for managing speculation on AMD processors, Revision 7.10.18. https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018. Accessed: Thu, 08 January 2020 17:45:00 +0100.

[6] Bernstein, D. J. Cache-Timing Attacks on AES, 2005.

[7] Bhattacharya, S., Maurice, C.-m.-t.-n., Bhasin, S., and Mukhopadhyay, D. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. *Cryptology ePrint Archive, Report 2017/968* (2017).

[8] Bhattacharya, S., and Mukhopadhyay, D. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES* (2016).

[9] Bhattacharyya, A., Sandulescu, A., Neugschwandt ner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS* (2019).

[10] Bosman, E., and Bos, H. Framing signals - A return to portable shellcode. In *S&P* (2014).

[11] Cabuk, S., Brodley, C. E., and Shields, C. IP Covert Timing Channels: Design and Detection. In *CCS'04* (2004).

[12] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., and Yarom, Y. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS* (2019).

[13] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium* (2019). Extended classification tree and PoCs at https://transient.fail/.

[14] Carruth, C. Introduce the "retpoline" x86 mitigation technique for variant 2. `https://reviews.llvm.org/D41723`, 2018. Accessed: Thu, 14 January 2020 20:30:00 +0100.

[15] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *CCS* (2010).

[16] Corbet, J. The current state of kernel page-table isolation. `https://lwn.net/Articles/741878/`, 2017. Accessed: Sun, 20 October 2019 17:40:00 +0100.

[17] Delshadtehrani, L., Eldridge, S., Canakci, S., Egele, M., and Joshi, A. Nile: A programmable monitoring coprocessor. *IEEE Computer Architecture Letters* (2017).

[18] Dowd, M., McDonald, J., and Schuh, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Addison-Wesley Professional, 2006.

[19] Edge, J. Kernel address space layout randomization. `https://lwn.net/Articles/569635/`, 2013. Accessed: Thu, 06 August 2020 15:30:00 +0100.

[20] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump over aslr: Attacking branch predictors to bypass aslr. In *MICRO* (2016).

[21] felixcloutier.com. CLFLUSH — Flush Cache Line. `https://www.felixcloutier.com/x86/clflush`, 2019. Accessed: Tue, 29 October 2019 17:45:00 +0100.

[22] felixcloutier.com. PREFETCHh — Prefetch Data Into Caches. `https://www.felixcloutier.com/x86/prefetchh`, 2019. Accessed: Tue, 07 August 2020 18:45:00 +0100.

[23] Fog, A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `https://www.agner.org/optimize/microarchitecture.pdf`, 2019. Accessed: Sun, 12 November 2019 20:00:00 +0100.

[24] Fuchs, A., and Lee, R. B. Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)* (2015).

[25] Gianvecchio, S., Wang, H., Wijesekera, D., and Jajodia, S. Model-based covert timing channels: Automated modeling and evasion. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection* (2008).

[26] Gregg, B. KPTI/KAISER Meltdown Initial Performance Regressions. `http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html`, 2018. Accessed: Thu, 14 January 2020 20:10:00 +0100.

[27] Gruss, D. Cache Template Attacks. `https://github.com/IAIK/cache_template_attacks`. Accessed: Thu, 11 November 2019 18:40:00 +0100.

[28] Gruss, D., and Canella, C. Transient Fail. `https://transient.fail/`. Accessed: Thu, 13 December 2019 18:00:00 +0100.

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

[30] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is Dead: Long Live KASLR. In *ESSoS* (2017).

[31] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[32] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[33] Gruss, D., Spreitzer, R., and Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[34] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P* (2011).

[35] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[36] Horn, J. speculative execution, variant 4: speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`. Accessed: Thu, 16 December 2019 18:30:00 +0100.

[37] Hu, W.-M. Lattice Scheduling and Covert Channels. In *S&P'92* (1992).

[38] Hund, R., Holz, T., and Freiling, F. C. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium* (2009).

[39] Hund, R., Willems, C., and Holz, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P* (2013).

[40] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling. Accessed: Thu, 07 January 2020 17:45:00 +0100.

[41] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2014. Accessed: Tue, 29 October 2019 18:20:00 +0100.

[42] Intel. Intel® Software Guard Extensions Programming Reference,Rev. 2. (2014). https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014. Accessed: Tue, 16 December 2020 17:00:00 +0100.

[43] Intel. Deep Dive: Retpoline: A Branch Target Injection Mitigation. https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation, 2018. Accessed: Thu, 10 January 2020 17:00:00 +0100.

[44] Intel. Intel Analysis of Speculative Execution Side Channels, Revision 1.0. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018. Accessed: Thu, 10 January 2020 17:00:00 +0100.

[45] Intel. Speculative Execution Side Channel Mitigations, Revision 3.0. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018. Accessed: Thu, 08 January 2020 17:45:00 +0100.

[46] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2019. Accessed: Tue, 14 November 2019 15:10:00 +0100.

[47] Ionescu, A. Twitter: Apple Double Map. https://twitter.com/aionescu/status/948609809540046849, 2017. Accessed: Thu, 14 January 2020 17:30:00 +0100.

[48] Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. Spoiler: Speculative load hazards boost rowhammer and cache attacks, 2019.

[49] Johnson, K. KVA Shadow: Mitigating Meltdown on Windows. https://msrc-blog.microsoft.com/2018/03/23/

kva-shadow-mitigating-meltdown-on-windows/, 2018. Accessed: Sun, 20 October 2019 17:40:00 +0100.

[50] Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security 8*, 2/3 (2000), 141–158.

[51] Kemerlis, V. P., Polychronakis, M., and Keromytis, A. D. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium* (2014).

[52] kernel development community, T. CPU frequency and voltage scaling code in the Linux(TM) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt. Accessed: Thu, 01 Februar 2020 19:55:00 +0100.

[53] kernel development community, T. iTLB multihit. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/multihit.html. Accessed: Thu, 22 May 2020 19:55:00 +0100.

[54] kernel development community, T. L1TF - L1 Terminal Fault. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html. Accessed: Thu, 22 May 2020 19:14:00 +0100.

[55] kernel development community, T. MDS - Microarchitectural Data Sampling. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/mds.html. Accessed: Thu, 22 May 2020 19:14:00 +0100.

[56] kernel development community, T. mmap(2) — Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html. Accessed: Thu, 09 December 2020 19:35:00 +0100.

[57] kernel development community, T. Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html. Accessed: Thu, 18 May 2020 19:15:00 +0100.

[58] kernel development community, T. Spectre Side Channels. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html. Accessed: Thu, 21 May 2020 16:40:00 +0100.

[59] kernel development community, T. TAA - TSX Asynchronous Abort. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/tsx_async_abort.html. Accessed: Thu, 22 May 2020 19:51:00 +0100.

[60] kernel.org. Linux Kernel Parameters. https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt. Accessed: Thu, 18 May 2020 19:30:00 +0100.

[61] kernel.org. Complete virtual memory map with 4-level page tables. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2009. Accessed: Sun, 20 October 2019 17:50:00 +0100.

[62] KERNEL.ORG. pagemap, from the userspace perspective. `https://www.kernel.org/doc/Documentation/vm/pagemap.txt`, 2009. Accessed: Sun, 27 October 2019 18:10:00 +0100.

[63] KHASAWNEH, K. N., KORUYEH, E. M., SONG, C., EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation.

[64] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA* (2014).

[65] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *S&P* (2019).

[66] KOCHER, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[67] KORUYEH, E. M., KHASAWNEH, K., SONG, C., AND ABU-GHAZALEH, N. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT* (2018).

[68] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-Pointer Integrity. In *OSDI* (2014).

[69] LAMPSON, B. W. A note on the confinement problem. *Communications of the ACM 16*, 10 (1973), 613–615.

[70] LEE, S., SHIH, M., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium* (2017).

[71] LEVIN, J. Mac OS X and iOS Internals: To the Apple's Core (Wrox Programmer to Programmer), 2012.

[72] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium* (2018).

[73] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *S&P* (2015).

[74] LUTOMIRSKI, A. x86/fpu: Hard-disable lazy FPU mode. `https://lore.kernel.org/patchwork/patch/953648/`, 2018. Accessed: Thu, 14 January 2020 20:10:00 +0100.

[75] MAISURADZE, G., AND ROSSOW, C. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS* (2018).

[76] Maurice, C., Neumann, C., Heen, O., and Francillon, A. C5: Cross-Cores Cache Covert Channel. In *DIMVA* (2015).

[77] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., and Römer, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[78] Mcilroy, R., Sevcik, J., Tebbi, T., Titzer, B. L., and Verwaest, T. Spectre is here to stay: An analysis of side-channels and speculative execution, 2019.

[79] Millen, J. 20 years of covert channel modeling and analysis.

[80] Mulnix, D. L. Intel® Xeon® Processor D Product Family Technical Overview. `https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-d-product-family-technical-overview.html#_Toc419802869`, 2015. Accessed: Wed, 05 August 2020 17:30:00 +0100.

[81] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[82] Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel.

[83] Percival, C. Cache missing for fun and profit. In *BSDCan* (2005).

[84] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[85] Projects, T. C. Site Isolation. `https://www.chromium.org/Home/chromium-security/site-isolation`, 2018. Accessed: Thu, 08 January 2020 17:30:00 +0100.

[86] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., and Bos, H. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium* (2016).

[87] Redhat. Controlling the Performance Impact of Microcode and Security Patches for CVE-2017-5754 CVE-2017-5715 and CVE-2017-5753 using Red Hat Enterprise Linux Tunables. `https://access.redhat.com/articles/3311301`, 2020. Accessed: Thu, 21 May 2020 21:14:00 +0100.

[88] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS* (2009).

[89] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security - TISSEC* (2012).

[90] SCHMIDT, W., HANSPACH, M., AND KELLER, J. A case study on covert channel establishment via software caches in high-assurance computing systems.

[91] SCHWARZ, M., GRUSS, D., LIPP, M., CLÉMENTINE, M., SCHUSTER, T., FOGH, A., AND MANGARD, S. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features.

[92] SCHWARZ, M., LIPP, M., MOGHIMI, D., VAN BULCK, J., STECKLINA, J., PRESCHER, T., AND GRUSS, D. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS* (2019).

[93] SCHWARZ, M., SCHWARZL, M., LIPP, M., AND GRUSS, D. *NetSpectre: Read Arbitrary Memory over Network*. 2019.

[94] SCHWARZL, M., SCHUSTER, T., SCHWARZ, M., AND GRUSS, D. Speculative dereferencing of registers: Reviving foreshadow, 2020.

[95] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, 2015. Accessed: Thu, 05 March 2021 22:30:00 +0100.

[96] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *CCS* (2004).

[97] SHAH, G., MOLINA, A., AND BLAZE, M. Keyboards and covert channels. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (2006).

[98] SHUTEMOV, K. A. pagemap: do not leak physical addresses to non-privileged userspace. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce`, 2015. Accessed: Sun, 27 October 2019 18:15:00 +0100.

[99] SOLIHIN, Y. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 2015.

[100] STECKLINA, J. An demonstrator for the L1TF/Foreshadow vulnerability (2019). `https://github.com/blitz/l1tf-demo`. Accessed: Thu, 09 December 2020 19:35:00 +0100.

[101] STECKLINA, J., AND PRESCHER, T. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

[102] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Memory Deduplication as a Threat to the Guest OS. In *EuroSys* (2011).

[103] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal War in Memory. In *S&P* (2013).

[104] Tanenbaum, A. S., and Bos, H. *Modern Operating Systems*, 4th ed. Prentice Hall Press, USA, 2014.

[105] Tromer, E., Osvik, D. A., and Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology 23*, 1 (July 2010), 37–71.

[106] Tsunoo, Y., Saito, T., and Suzaki, T. Cryptanalysis of DES implemented on computers with cache. In *CHES* (2003).

[107] Turner, P. Retpoline: a software construct for preventing branch-target-injection. `https://support.google.com/faqs/answer/7625886`, 2018. Accessed: Thu, 08 January 2020 17:30:00 +0100.

[108] Valsorda, F. Searchable Linux Syscall Table for x86 and x86_64. `https://filippo.io/linux-syscall-table/`, 2020. Accessed: Thu, 23 Oktober 2020 13:45:00 +0100.

[109] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium* (2018).

[110] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: Rogue In-flight Data Load. In *S&P* (2019).

[111] Wang, Z., and Lee, R. B. Covert and Side Channels due to Processor Architecture. In *ACSAC* (2006).

[112] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution, 2018.

[113] Wiki, D. Microcode. `https://wiki.debian.org/Microcode`. Accessed: Thu, 10 November 2020 18:45:00 +0100.

[114] Wu, Z., Xu, Z., and Wang, H. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium* (2012).

[115] Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium* (2016).

[116] Xiao, Y., Zhang, Y., and Teodorescu, R. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities, 2019.

[117] Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., and Schlichting, R. An exploration of L2 cache covert channels in virtualized environments. In *CCSW'11* (2011).

[118] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C. W., and Torrellas, J. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO* (2018).

[119] Yan, M., Shalabi, Y., and Torrellas, J. Replayconfusion: Detecting cache-based covert channel attacks using record and replay.

[120] Yarom, Y., and Falkner, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[121] Zhang, Y., and Reiter, M. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS'13* (2013).