Information Security

Speculative execution side-channel attacks

Exploring cause, impact and mitigations

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

eingereicht von

Julian Simon Rauchberger

is161524

im Rahmen des

Studienganges Information Security an der Fachhochschule St. Pölten

Betreuung

Betreuer/in: FH-Prof. Dipl.-Ing. Dr. Sebastian Schrittwieser, Bakk.

St. Pölten, August 6, 2018

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

*

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Kurzfassung

Durch die Veröffentlichung der Meltdown und Spectre Schwachstellen Anfang 2018 haben Probleme, die durch unsicheres Hardware Design verursacht wurden, zusätzliche Aufmerksamkeit von Medien und Forschern erhalten. Auf Grund der Komplexität der Sicherheitslücken ist es noch immer schwerig umfangreiches Verständnis über diese Lücken zu erlangen. Diese Arbeit zielt darauf ab, sogenannte Speculative Execution Side-Channel Angriffe im Detail zu erklären. Dies geschieht durch Analyse bekannter Schwachstellen um Ursache, Auswirkungen und verfügbaren Risikominderungen festzustellen. Um auch zukünftige Auswirkungen dieser Angriffe verstehen zu können wird eine umfassende Analyse der eigentlichen Ursachen dieser Sicherheitslücken durchgeführt. Eine ausführliche Beschreibung der Hardwarekomponenten, die an Out-of-order Execution beteiligt sind, wird durchgeführt um Kernprobleme zu identifizieren und aufzulisten. Außerdem wird undokumentiertes CPU Verhalten analysiert und es werden weitere Angriffsvektoren erforscht. Eine neuartige Technik um Hardware Breakpoints, die Lesezugriff auf Speicherbereiche überwachen, zu umgehen wird vorgestellt und angewendet um ein System Management Mode Rootkit zu erkennen. Abschließend werden drei mögliche Hardwareänderungen zur permanenten Verhinderung von Speculative Execution Side-Channel Angriffen erläutert und eine Kombination aus zwei Techniken empfohlen um größtmöglichen Schutz bei geringen Performanceeinbußen zu ermöglichen.

Abstract

With the public release of the Meltdown and Spectre vulnerabilities in early 2018, issues stemming from insecure hardware design have received additional attention by both media and researchers. Due to the complex nature of the vulnerabilities discussed, it is still difficult to gain a comprehensive understanding of these issues. This paper aims to shed light on so-called speculative execution side-channel attacks by analyzing cause, impact and mitigations of known vulnerabilities. To fully understand future implications of these attacks, we conduct a comprehensive analysis on their root causes. We give an in-depth explanation of hardware components involved in out-of-order execution and describe the core issues we identified. Furthermore, we reverse engineer undocumented CPU behavior and explore other attack vectors. We also demonstrate a novel technique of using speculative reads to bypass hardware breakpoints that trigger on memory reads and apply it to detect a System Management Mode rootkit. Finally, we describe three possible ways to alter hardware design to permanently eliminate speculative execution side-channel attacks and recommend a combination of them to provide in-depth protection while keeping performance impact as low as possible.

Contents

1. Introduction

Speculative execution side-channel attacks are a new class of vulnerabilities which has been publicly disclosed in early 2018. Most commonly known as Meltdown and Spectre, these attacks are based on flaws found in many modern CPU architectures including x86 and ARM. As they target properties of the hardware implementation itself, they are often extremely hard if not impossible to mitigate and possibly affect billions of devices.

What makes these vulnerabilities an interesting topic of research is the fact that they expose an entirely new attack vector which has previously often been ignored in threat models as it had been deemed too unlikely to occur. With these attacks, fundamental assumptions about privilege separation at the hardware level are no longer valid. With speculative execution side-channel attacks, it is possible to cross low level boundaries such as address space isolation which have previously been considered a fundamental access control feature that serves as the base for many advanced protection mechanisms.

With knowledge about these attacks readily available to any sufficiently motivated malicious actor, it is necessary to update threat models and include mitigations to stop exploitation attempts. In order to do so, in-depth understanding of the published vulnerabilities is required. Additionally, long-term protections can only be established with a sufficient understanding of the root causes of speculative execution side-channel attacks. They do not merely represent a handful of issues that can be fixed with a set of patches but rather an ongoing field of research where new variants are discovered regularly. Furthermore, mitigations for these vulnerabilities often introduce performance penalties that strongly depend on the workload. As such, it is necessary to understand both the attack surface as well as the impact of workarounds and patches in order to deploy appropriate mitigations without unnecessarily degrading performance.

This paper aims to provide the reader with the in-depth understanding of speculative execution side-channel attacks that is required in order to correctly evaluate threats posed by them. We give an explanation of the parts of the CPU that are responsible for out-of-order execution and eventually gave rise to the attacks described. Furthermore, we give in-depth information about the vulnerabilities, their impact and currently available patches for both Meltdown and the Spectre class of attacks. Additionally, we consider their impact on security-critical features such as Software guard eXtension and System Management Mode.

We also provide an analysis of the root causes that gave rise to these security issues and detail possible future attack surface. We analyze the behavior of speculatively running code with the aim of allowing other researchers to build upon our results. We further apply this knowledge by researching various aspects of speculative execution such as the behavior of the return branch predictor. Additionally, we investigate the possibility of creating write-based speculative execution side-channel attacks and give an in-depth explanation of the hardware components involved. We demonstrate a novel way of employing out-of-order execution to read arbitrary memory without triggering hardware breakpoints and apply the technique to detect a System Management Mode rootkit. Finally, we describe three possible ways to permanently fix speculative execution side-channel attacks by modifying the hardware design of the CPU.

The main contributions of our paper are:

- a detailed root cause analysis of speculative execution side-channel attacks, including software impact and hardware design
- in-depth description of various actual vulnerabilities such as Meltdown, Spectre and the lazy FPU bug
- analysis of possible future attack surface such as write-based attacks and poisoning of the return predictor
- a novel root kit detection technique using speculative reading of memory
- possible design modifications to mitigate the described attacks

2. Background

For a holistic assessment of speculative execution side-channel attacks, it is necessary to understand the complex hardware details that gave rise to these vulnerabilities. Additionally, observing historic hardware issues with security impacts allows for a better understanding of the threat generally posed by such issues.

2.1. Hardware bugs

Although not exactly common, bugs stemming from erroneous hardware design have affected computer systems for a long time. For instance, in 1994, the infamous Pentium FDIV bug caused affected processors to produce incorrect floating-point numbers when executing certain divisions. This ultimately led to Intel recalling the flawed units, costing them millions[1]. While this bug did not cause any immediate security issues, there have been others which could have led to such concerns.

2.1.1. Erratum SKZ6

In 2016, developers of the OCaml programming language encountered an obscure hyper-threading bug affecting Skylake and Kaby Lake processors which they later documented in public blog posts after Intel had released a microcode update to address the issue [1, 2]. The errata released by intel describes the problem as following:
"Under complex micro-architectural conditions, short loops of less than 64 instructions that use AH, BH, CH or DH registers as well as their corresponding wider register (e.g. RAX, EAX or AX for AH) may cause unpredictable system behavior. This can only happen when both logical processors on the same physical processor are active." [3]
While the exact impact of the bug is not immediately clear, it seems very likely that under the conditions described, one logical hyper-threading core can corrupt or otherwise influence the registers of the other sibling core. To the best of the authors knowledge, no public demonstrations of the exploitability of this bug exist. It is very likely that that the bug is practically unexploitable because of how hard it is to

[1]http://www.trnicely.net/pentbug/pentbug.html

trigger in a meaningful way, however it clearly demonstrates how hardware bugs can break fundamental privilege boundaries. The OCaml developer that first analysed the issue also notes that on at least one occasion, page tables of the operating system were corrupted by the bug. This clearly proves that privilege escalation is, at least theoretically, possible.

2.1.2. Erratum HSW136

From a security point of view, another notable hardware bug is Erratum HSW136 [4], which led Intel to disable the newly introduced Transactional Synchronization eXtension (TSX) feature on certain affected CPUs. While details on the issue are sparse, it can be assumed that the impact must have been relatively severe if it led to Intel disabling an entire CPU feature. TSX can be used to implement transactional memory operations at the hardware layer and can be a significant performance improvement for multi-threaded code that relies heavily on locking. With TSX, multiple memory addresses can be modified within a transaction without the changes being visible to other threads. Then, all modifications can be committed with a single instruction or alternatively rolled back. Incorrect implementation of such a feature could lead to unexpected race conditions or maybe even corruption of arbitrary memory if permission checks aren't working correctly.

2.1.3. The memory sinkhole

In 2015, Christopher Domas demonstrated an attack where the memory holding the Local Advanced Programmable Interrupt Controller (LAPIC) registers is remapped into a memory region otherwise inaccessible to even the operating system kernel known as System Management RAM (SMRAM) [5]. By carefully shadowing a key data structure, a Global Descriptor Table (GDT) under the attackers control will be loaded and ultimately result in execution resuming outside of SMRAM. This attack shows that even a relatively weak primitive that allows only to set a small region of memory to zero allows a skillful attacker to escalate privileges.

2.1.4. Meltdown and Spectre

Probably the most widely known example of hardware security issues with far-reaching consequences are the Meltdown and Spectre bugs discovered independently by Google Project Zero, Cyberus Technology and Graz University of Technology. Publicly disclosed in January 2018, these vulnerabilities brought the importance of securely designed and correctly implemented hardware to the eye of the public. Both were practical, exploitable bugs with very real consequences but no easy solution. Careful balancing between effectiveness and performance impact had to be done when implementing patches. It was also

discovered that the same or similar problems could be found in many other architectures such as ARM or x86 CPUs manufactured by AMD. In the following months, other researchers discovered and published papers on variants of the initial vulnerabilities [6] [7].
The chapters 3 and 4 provides an in-depth analysis of the inner workings of these vulnerabilities and the principles they are based on.

2.1.5. Conclusion

While this is by no means a comprehensive list of issues caused by hardware bugs, the examples clearly demonstrate the massive threat that they can pose. All these security issues highlight how hard it is to respond to threats that break the very foundations operating systems base their security on. Kernel as well as userland software completely relies on the correct implementation of certain security features at the hardware layer. If these assumptions do not hold, system confidentiality, integrity and availability can no longer be guaranteed. Unlike software-based security issues, these bugs affect an even larger number of systems and can be very hard or even impossible to patch. As described in the examples above, responses ranged from simple microcode updates over selective disabling of features up to the costly recalling of entire product lines. More research and responsible disclosure is needed to ensure potential security issues are identified and fixed in a timely manner.

2.2. Out-of-order execution

To increase the utilization of all of its components, modern processors make use of out-of-order execution. This means that instructions are not necessarily executed in the same order as they appear in the instruction stream. Since different components of the CPU are used for different computations such as additions, multiplications or memory access, instructions can sometimes be executed in parallel. Many processors also possess extensive circuitry to detect dependencies between operations and can therefore reorder operations to increase utilization and throughput while still producing the correct output. A reorder buffer is used to ensure that even though actual execution is out-of-order, the instructions retire in the same order they were originally issued [8].

2.2.1. Out-of-order engine

The actual hardware responsible for out-of-order execution is called the Out-of-order engine as described in the Intel 64 and IA-32 Architectures Optimization Reference Manual [9]. It splits the instruction stream into dependency chains which are then sent to execution. If one dependency chain has to wait for a resource such as an L2 cache entry, another dependency chain can be executed in the meantime. The Out-of-order engine consists of three major parts working together: the *renamer*, the *scheduler* and the *retirement component*.

The *renamer* connects the incoming in-order view of the instruction stream to the scheduler. It is responsible for moving micro-ops from the micro-op queue to the Out-of-order engine. During this process, it also renames architectural sources and destinations to micro-architectural sources and destinations, hence the name of this unit. At very low levels, the CPU does not work with traditional register names programmers are used to but rather dynamically reassigns them to a larger set of internal micro-architectural registers. Additionally, resources like load and store buffers are allocated to the micro-ops at this point. The renamer is also able to detect and remove false dependencies. An example of this would be the `XOR RAX, RAX` instruction. Even though it operates on the `RAX` register, the result will always be zero and therefore not depend on the initial value of `RAX`. The renamer can detect and correctly remove such Dependency Breaking Idioms to ensure they do not affect out-of-order execution negatively.

The *scheduler* is responsible for dispatching micro-ops to the execution core. It identifies which micro-ops are ready and have all required input available and then selects which of those are dispatched every cycle.

When an instruction has completely been executed and it is certain that no faults or exceptions that would invalidate the result occurred, they are retired. This means that their results take actual effect and will be visible to the outside world. This process is the responsibility of the *retirement component* which also ensures that micro-ops retire in the same order they were issued.

2.2.2. Speculative execution

When out-of-order execution reaches a point where the CPU cannot be sure which way to continue, speculative execution is employed. This happens for instance because of a branch that depends on a memory location that is not in the cache. Since it is not yet known where in memory the next instruction will be and waiting for the memory load would be too time consuming, the branch prediction unit is used to take a guess. Execution is then continued there. In the best case, the prediction is later shown to be correct and the executed instructions can be retired normally. If not, everything after the branch has to be abandoned and the correct path has to be executed. This is usually done by creating a checkpoint at the time of the branch and later reverting the CPU state to this checkpoint. For the sake of simplicity, we will refer to this process as *"snapshots"* and *"rollbacks"* within this paper.

2.2.3. State restoring implementation

At the hardware level, restoring an older CPU state requires two separate mechanisms. The first is to ensure that register modifications can be rolled back, and the old state of all registers can be restored. This is implemented through *register renaming*. Internally, the processor has many more physical registers than those which are logically exposed to the programmer. These so-called micro-architectural registers can be dynamically assigned to the logical registers. To implement the concept of a register snapshot, the CPU stores at a given point in time the association between logical and micro-architectural registers and ensures that these do not get modified. The CPU then proceeds to work only with the other micro-architectural registers which are not part of the snapshot during speculative execution. If the speculated instructions retire, the snapshot metadata can be discarded. In case of a rollback, all modifications are discarded instead, and the registers are restored from the saved state. The second mechanism required to implement snapshots deals with memory writes as these also have to be able to be undone. This is implemented through the use of buffers which store the modifications before they are committed to main memory. In case of a rollback, the contents of the buffers can simply be discarded.
Branch mispredictions generally have a rather large performance penalty as not only the results of already executed calculations have to be thrown away but also the correct ones have to be executed from scratch. Because of this, there is a large incentive to ensure branches are correctly predicted, resulting in complex algorithms being used by the branch prediction units.
To avoid repetitions of the same topic, when we refer to speculative execution within this paper, we generally mean instructions that are being executed but have not been retired yet and might still be abandoned, including normal out-of-order execution that does not directly depend on branch prediction.

2.3. Cache timing attacks

At the core of both Meltdown and Spectre are timing-based side-channels that can leak information from the speculative execution context. While these attacks are by far the most prominent and have an arguably very high impact, the idea of using microarchitectural timing differences to breach security boundaries is not new. Historically speaking, these side-channels have mostly been used to defeat *Kernel Address Space Layout Randomization* (KASLR). With KASLR, the operating system kernel gets loaded at a new, randomized address during each boot. By using a variety of different techniques to access random memory addresses in kernel space and measuring the access times, researchers found various ways to differentiate between mapped and unmapped areas. In some cases, it is even possible to get additional information about a given memory area.

2.3.1. Practical attacks

In 2013, Hund et al. demonstrated multiple timing-based side-channel attacks that disclose information about the kernel memory [10]. The usage of memory access time differences based on the cache level the data can be found in is very similar to the techniques used in the Meltdown and Spectre attacks. Similar techniques have also been previously described by researchers from Graz University of Technology [11]. The *DrK* attack, developed by Yeongjin Jang et al., makes use of the Intel TSX feature to suppress exceptions normally raised by access violations when code with userland privileges tries to access kernel memory [12]. They found that measuring timing of the execution of the TSX abort handler is much more accurate than doing the same with an exception handler. The attack can not only differentiate between mapped and unmapped areas but also between executable and non-executable.
The x86 architecture is especially susceptible to these attacks because it provides all the necessary tools to unprivileged code by default. Instructions of the `CLFLUSH` family can be used to reliably remove a given memory address from cache. Additionally, multiple `PREFETCH` instructions exist that can be used to load a memory address into any given cache level at will. With `RDTSC`, attackers have access to a high-resolution timer that greatly simplifies many attacks. Even though unprivileged access to this instruction can be disabled, there is no major operating system that does so to the best of the authors knowledge. Multiple strategies for cache attacks have been identified in the past, including *Evict+Time [13]*, *Prime+Probe [13]* and *Flush+Reload [14]*.

2.3.2. ARMageddon

Even though specialized assembly instructions make certain attacks easier, it is also possible to exploit timing-based side channels without them, as demonstrated by the *ARMageddon* attacks [15]. They are based on selectively flushing certain shared memory code from cache to determine which code has been executed by other applications. This is used to spy on smartphone users by flushing code that handles different user interactions such as gestures and key presses. Depending on which parts of the code get cached again, attackers can get some understanding of how the victim interacts with the smartphone. Contrary to most of the previous attacks which were focused on Intel CPUs, ARMageddon works on the ARM architecture. The authors did a comprehensive study on the cache eviction policy used by widespread ARM CPUs and were able to reverse engineer the algorithm. This allowed them to remove a given address from cache by accessing and thereby caching other addresses in a way that causes collisions in the cache. With knowledge of the caching algorithm it is possible to execute caching-based attacks even though there is no dedicated flush instruction on most ARM CPUs. Unlike x86, the ARM architecture also does not contain a high-resolution timer with unprivileged access that could be used to determine access times. The authors provide multiple alternative ways such as performance events or a dedicated thread timer to solve this problem. ARMageddon clearly demonstrates that cache-based side-channels can also be exploited on Architectures that do not provide a dedicated flush instruction and a high-resolution timer.

3. Meltdown

The *Meltdown [16]* vulnerability affects most modern Intel CPUs and also some ARM processors. According to AMD, Meltdown does not apply to their products due to differences in processor design[1]. It allows unprivileged users to disclose kernel memory from Ring 3 by using a side-channel during speculative execution and thereby breaks all security assumptions given by address space isolation. On some architectures, Meltdown can also be used to read privileged registers, but this paper focuses on the Intel x86 architecture where this is not true.

3.1. Vulnerability

Meltdown is a form of race condition between fetching an address and completing the permission check. When accessing a memory address the code lacks permission to read, out-of-order execution allows multiple instructions to be executed before the violation is detected. At that time, all modifications based on the value read will be invalidated and the state of the CPU will be rolled back. If implemented correctly, this would be enough for Meltdown to be unexploitable as the result of the fetch would not be available to the attacker. The vulnerability lies in the fact that changes in some microarchitectural states will not be correctly rolled back and can therefore be used as a side-channel to leak information. In the original paper, this side-channel has been the cache hierarchy. By selectively accessing memory and therefore fetching it into cache during out-of-order execution, the researchers were able to leak data one bit at a time. To give a simple example, an attacker could access address A if the secret value is 1 or address B if it is 0. By measuring access times to these locations after the rollback, an attacker would be able to establish a side-channel. This attack scenario is detailed in Figure 3.1. It is currently not clear if and how many other microarchitectural side-channels exist.

[1]https://www.amd.com/en/corporate/security-updates

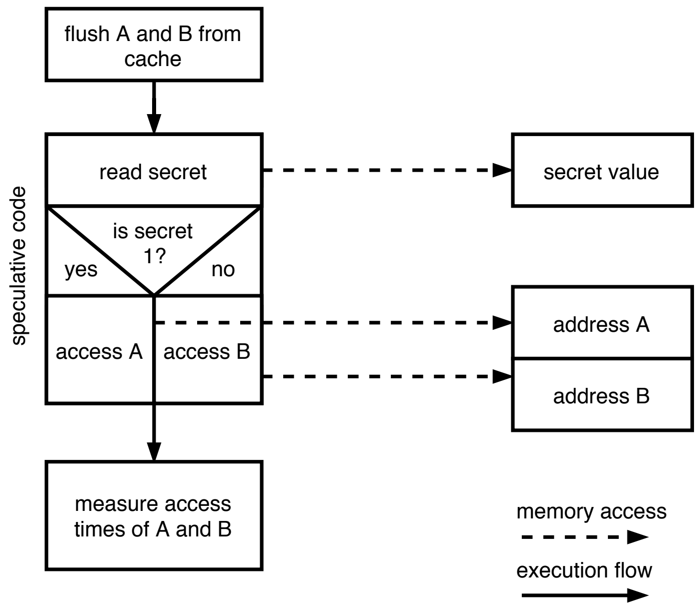

Figure 3.1.: Meltdown attack

3.2. Impact

Meltdown breaks privilege-based separation of kernel and userland memory, more specifically the protection provided by the page tables. The page tables are a hierarchical memory structure set up by the kernel that defines the mapping between physical and virtual addresses. The configuration defined in the page tables is then enforced by the Memory Management Unit (MMU). Each entry contains a *User/Supervisor (U)* bit which is a flag that controls if the page can be accessed from the userland or requires Ring 0 privileges. This normally prevents unprivileged code from reading data from kernel memory but the Meltdown vulnerability breaks this assumption. Research conducted by Google Project Zero [17] indicates that not all kernel memory can be read with Meltdown but rather requires a precondition. They assume this to be presence of the targeted kernel memory in the L1 data cache but do not hold complete confidence in this statement. It seems however reasonable that Meltdown requires the targeted data to be in the cache hierarchy as fetches from main memory typically require around 200 CPU cycles and are therefore too slow to be completed during the short time window between creation of the violation and enforcement that Meltdown is based on. Additional research indicates that Meltdown cannot be used to cross other privilege boundaries such as *Intel Software Guard eXtension* (SGX) [6] and *System Management Mode* (SMM)[2]. This indicates that the Meltdown vulnerability only affects the protections normally provided by page table isolation and is not a more general bypass that can be used to read any protected memory locations.

[2]https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/

3.3. Meltdown patches

The Linux kernel fixes the Meltdown vulnerability with the *kernel page-table isolation* (KPTI) workaround which is based on the older set of *KAISER* (Kernel Address Isolation to have Side-channels Efficiently Removed) patches originally presented in a paper by Gruss, Lipp, Schwarz, Fellner, Maurice and Mangard [18].

The idea behind KPTI is to work around the incomplete protections provided by page tables in light of Meltdown by unmapping as much as possible of the kernel code and data regions while userland code is running. Instead of only using one set of page tables, KPTI introduces a second set. When kernel code is running, the original set that includes both kernel and userland regions is used. When execution passes to unprivileged Ring 3 code, a second set that maps only userland memory and a minimal amount of kernel code to handle syscalls, interrupts and exceptions is being used. Since Meltdown can only be used to read cached and mapped kernel memory, this completely protects against the vulnerability at the cost of performance. The exact impact is hard to measure as it depends heavily on the workload and also on the CPU generation. One of the biggest reasons KPTI impacts performance is the requirement of a *translation lookaside buffer* (TLB) flush during each context switch. As the kernel memory gets temporarily unmapped, the cached mapping entries have to be removed from the TLB as well. This also means they will not be available the next time kernel code runs, strongly limiting the positive effect TLB normally has on address translation performance. This can be avoided by making use of the *processor-context identifiers* (PCID) hardware feature which Linux fully supports since Kernel version 4.14. PCID allows to tag TLB entries with a context identifier that can be used to dynamically control access to the entries. It limits TLB lookups to the currently allowed context. By using different PCIDs during kernel and userland execution, the kernel TLB entries can be made invisible to lower privilege levels without having to flush them. On older processors not supporting PCID, the performance impact of KPTI is therefore much bigger. To the best of the authors knowledge, the page table isolation software workaround is at the time of writing the only possible protection against Meltdown on affected Intel processors. No microcode updates that address Meltdown specifically have been released but Intel has stated that an upcoming hardware redesign will feature protections against Meltdown.

Both the Windows and MacOS operating systems have implemented Meltdown fixes in a similar fashion to Linux. Microsoft states that performance impact is negligible on Windows 10 PCs with Skylake, Kabylake or newer processors. Haswell or older shows a more significant slowdown, especially when paired with an older version of Windows[3].

[3]https://cloudblogs.microsoft.com/microsoftsecure/2018/01/09/understanding-the-performance-impact-of-spectre-and-meltdown-mitigations-on-windows-systems/

4. Spectre

While Meltdown can be very clearly defined and worked around, *Spectre [8]* describes a full class of vulnerabilities with many variants, some of which have been discovered after the original paper had been published. Spectre attacks are based on maliciously inducing branch mispredictions in a way that benefits the attacker by crossing security boundaries. Depending on the variant, this can be done either directly or by mistraining a branch predictor.

4.1. Variant 1: Bounds check bypass

Variant 1 is also known as *bounds check bypass* (BCB) and summarizes attacks where a check is bypassed during speculative execution because the CPU wrongly assumes the check will succeed. This can for instance happen if the user is able to specify an index for an array and the code first checks if the index is within the bounds of the array. By doing multiple runs with an in-bounds index, the branch predictor can be trained to assume that the check will succeed, and the array access will subsequently be executed. Afterwards, an out-of-bounds index is supplied, the branch prediction will assume that the bounds check will succeed again and speculatively run the array access code with an invalid index, ultimately accessing data outside the array. The attacker must then find a way to leak the accessed data through a cache side-channel. This attack is most relevant in the scenario of untrusted code running inside a sandbox, e.g. *JavaScript* in a web browser. Under these circumstances, Spectre variant 1 has been demonstrated to be able to read arbitrary browser memory which might contain secrets such as cookies or passwords.

4.2. Variant 2: Branch target injection

Variant 2, *branch target injection* (BTI), is described as an attack scenario where the internal state of the branch predictor is manipulated to ensure that branch prediction will predict an address favorable to the attacker. If successful, this attack can essentially be used to speculatively execute arbitrary code in the context of the poisoned branch instruction. This vulnerability can most easily be exploited with indirect branches and has been demonstrated to be able to leak data from one process to attack code running in another.

4.3. Spectre-NG

Eight new Spectre variants, dubbed Spectre-NG[1], have been announced in May 2018, however at the time of writing only two of them have been publicly released. Variant 3a, known as *rogue system register read* affects systems that allow speculative reads of system registers and can allow attackers to read their contents by employing the same side-channels used in other Spectre attacks. Variant 4, *speculative store bypass*, is based on processors speculatively reading memory before the addresses of all prior writes are known which may lead to the reading of an earlier value.

4.4. Impact

Spectre highlights a critical issue: the sharing of architectural states between code running in different security contexts, for instance branch prediction tables shared between processes running as two different users. A malicious user could deliberately craft jumps to poison the branch prediction table in a way that incorrectly predicted branches in the process of another user. Similar to the side-channels employed in Meltdown, these mispredictions could then cause the leakage of information. A commonly demonstrated way to exploit this is by bypassing array bounds checks. If no mitigations are present and exploitable gadgets exist, Spectre can theoretically be used to read all memory present on the system. Like Meltdown, this requires an attacker to be able to execute arbitrary code on the system.
Spectre is known to affect most advanced CPUs that have some sort of dynamic branch prediction. This includes Intel and AMD, some high-performance ARM CPUs and possibly others. At the time of writing, the full range of affected devices is still being discovered.

4.4.1. Updating security boundaries

Spectre/Meltdown has led to an update of the threat model used by the Chromium developers. They now consider all active web content such as *JavaScript*, *Flash* or *WebAssembly* as being able to abuse side-channels to read all data hosted in the same address space[2].
In the light of Spectre, it is necessary to reconsider many traditionally uncrossable security boundaries. Theoretically, Spectre can be used to read any memory on the system if vulnerable code runs at any

[1]https://www.heise.de/ct/artikel/Exclusive-Spectre-NG-Multiple-new-Intel-CPU-flaws-revealed-several-serious-4040648.html
[2]https://chromium.googlesource.com/chromium/src/+/master/docs/security/side-channel-threat-model.md

time and has access to the targeted data. As the side-channels exploited by Spectre have historically not been considered when implementing code, it should be assumed that all code written previously to the discovery of Spectre is vulnerable. Without hardware-based fixed provided by CPU vendors, it is only possible to limit the impact of Spectre as much as possible by making exploitation harder to the point where it becomes infeasible. As long as the underlying architecture-state sharing has not been fixed, an attacker with enough resources should be assumed to be able to write a functioning exploit for Spectre.
In contrast to Meltdown, which can only be used to bypass page table-based isolation between userland and kernel, Spectre has a much wider range of impact on privilege boundaries. It can be used to read memory from protected region in the same process (e.g. memory normally not accessible by a JavaScript engine), other processes, the kernel and other protection mechanisms such as SGX. However, many of these attacks are extremely hard to execute and while proof of concept implementations exist, there is no known account of in-the-wild exploitation where actual meaningful data has been compromised. To give an example, Spectre theoretically allows JavaScript hosted on a website to manipulate the branch predictor in a way that attackers can read a text document opened in another program. Practically speaking, this attack is currently infeasible as there are too many unknown variables such as *CPU generation*, *microcode version*, *address space layout randomization* and the *scheduler* that would need to be accounted for. Implementing a reliable attack that works on multiple systems seems almost impossible, but it could theoretically be done with enough resources which is why Spectre should still be considered as a vulnerability that needs to be addressed.

4.5. Patches

Finding workarounds for Spectre vulnerabilities is not as straightforward as with Meltdown. As it is extremely hard if not impossible to determine the full scope of all Spectre vulnerabilities and new variants are still being discovered, it cannot be said that Spectre has been fully mitigated at the time of writing. While certain protections exist, and exploitation has become harder, it is very likely that this will stay a topic of research for the foreseeable future.

4.5.1. Retpoline

Compilers have been updated to include the *Retpoline* feature which protects indirect jumps by replacing them with return statements and adding an endless loop that will never be executed during runtime but which the CPU will predict as the correct branch target, thereby trapping speculative execution. While being a valid mitigation in some cases, Retpoline cannot provide protections under all circumstances[3]. It should be noted that on Skylake and newer processors, the return branch predictor has a fallback mechanism that will use the branch target buffer to predict branches when the return stack buffer underflows. This means that on these systems, retpoline cannot be seen as a reliable protection mechanism and needs to be supplemented by hardware-based protection[4].

4.5.2. Microcode updates

Intel has issued Microcode updates for modern CPUs which further mitigate known issues. Several additional CPU features were introduced that can help kernel developers protect against Spectre-style attacks.

The newly introduced *indirect branch restricted speculation* (IBRS) is meant to provide protection against certain attacks based on variant 2, the branch target injection. When IBRS is active, predicted target addresses cannot be influenced by code that executed in a less privileged prediction mode before that IBRS mode was last set to one[5]. This means IBRS can be used to protect branch prediction in the kernel from code running in userland.

The second new feature, *single thread indirect branch predictors* (STIBP) stops sibling hyperthreading cores from influencing indirect branch prediction on each other. This is especially important since hyperthreads share more hardware resources than physicall cores on a processor do. Providing logical separation between them is something that can only be done at the processor level and not by the operating system [19].

Another new feature, the *indirect branch prediction barrier* (IBPB) can be used to flush the branch prediction state at any time and completely reset it. Operating systems can use this feature to ensure code running previously has no impact on code running afterwards. The kernel can for instance use this to reset the branch prediction table and associated data structures when it switches execution from one process to another. This however comes with a non-negligible performance impact and should therefore only be used where absolutely necessary[6].

All of the new features described here have been delivered to supported CPUs through microcode updates. This of course means that it is crucial for security to deploy these updates in a timely fashion because even if operating systems support the techniques, they cannot make use of them if the underlying microcode does not yet include them.

[3]https://lkml.org/lkml/2018/1/4/724
[4]https://lkml.org/lkml/2018/1/4/724
[5]https://lwn.net/Articles/743019/
[6]https://patchwork.kernel.org/patch/10145335/

5. Determining root causes

In order to determine where further attack vectors in the Intel architecture might be located, it is first necessary to determine the key issues that lead to the exploitability of Meltdown and Spectre. We identified three core findings than can be used as a starting point for further research:

1. Side-channel leakage from speculative execution
2. Microarchitectural states shared between security contexts
3. Delayed or non-existentent security enforcement

The following sections explain these findings in more detail.

5.1. Side-channel leakage from speculative execution

None of the vulnerabilities exploited in Meltdown and Spectre attacks would have any real-world impact if it were not possible to leak information obtained during speculative execution to code running outside of this context. The core issue here is that the complete architectural state at the beginning of speculative execution is not restored during a roll back to a previous checkpoint. This is specifically exploited by abusing the fact that cache loads occurring during speculative execution are not evicted upon a roll-back. This can be used to leak the obtained information. For instance, by either accessing address A if a 1 is encountered or address B if 0 is encountered. It can then later be determined which address had been accessed by measuring load times to determine if it can be found in the CPU cache. It is very likely that cache loads are not the only microarchitectural state that is not reset during a roll back. Further research is required to determine if information can also be leaked in other ways.

5.2. Microarchitectural states shared between security contexts

The second issue is that internal information recorded on certain events such as branches can influence code running in other security contexts. For instance, it is possible that branches taken in process A influence the prediction behavior in process B. This allows a malicious user to manipulate prediction behavior in other contexts. By combining this with a leaking side-channel, it is possible to obtain information otherwise protected from access. The problem here is that internal CPU states are shared between different contexts, can be modified by any of them and in turn influence behavior in other contexts. It remains to be determined how many of these internal states exist and which can be manipulated in ways that have impact on the security of the system. It would be especially interesting to see if there are shared states that are not related to the branch prediction tables exploited in Spectre attacks.

5.3. Delayed or non-existentent security enforcement

The Meltdown attack is based on the fact that it is possible to access otherwise restricted memory regions during speculative execution and also execute additional instructions before the access violation is enforced. In combination with a leaking side-channel, this allows to acquire information otherwise unobtainable and persist some of it in way that is not reset by a roll back. The core problem here seems to be that it is possible to execute additional instructions between the occurrence of the access violation and retirement of the instruction block. In other words, enforcement of access control is delayed instead of immediate during speculative execution. Further vulnerabilities beside reading of unauthorized memory areas might exist. Meltdown demonstrates that code executed during speculative execution does not behave as it would be expected if the system were only doing simple in-order execution. Further analysis is required to determine if there are any additional enforcement differences that could be used by bypass security boundaries.

5.4. Assembly instructions

Additionally, it should be noted that a range of assembly instructions exists on the x86 platform that can simplify exploitation of Spectre and Meltdown type vulnerabilities. Even if they are not strictly required in most cases, we consider them part of the attack surface as they make both discovery and exploitation significantly easier and might make attacks on otherwise unexploitable vulnerabilities more realistic. The following section gives an overview of these instructions and explains which benefits they provide for attack scenarios.

5.4.1. PREFETCH

This family of instructions allows an unprivileged user to fetch memory into the CPU cache. Multiple variants exist that affect different layers of the cache hierarchy. While of course not strictly necessary to execute attacks on speculative execution, this gives an attacker detailed control about the cache and can thus be beneficial.

5.4.2. CLFLUSH

`CLFLUSH` removes memory at the given address from all levels of the cache hierarchy. Researchers have demonstrated that the same effect can also be achieved by selectively accessing certain memory locations in order to get an address of their choosing evicted from cache [15]. This however requires extensive understanding of the eviction strategy used by the CPU which is time consuming to reverse engineer and might differ between models or generations. `CLFLUSH` provides a simple and portable means, greatly simplifying attacks that require certain memory locations to be uncached.

5.4.3. NON-TEMPORAL hints

The Intel architecture contains several variants of MOV instructions which contain a "non-temporal hint", for instance `MOVNTI`. When these instructions are used to read or write memory, the CPU will not fetch the corresponding line into the cache hierarchy. This can be helpful for attackers when they want to ensure that auxiliary code does not pollute the cache.

5.4.4. RDTSC

This instruction can by default be executed from userland code (Ring 3) and allows access to a high-resolution timer. The `EDX` and `EAX` registers are filled with a timestamp counter with a combined resolution of 64 bit. This counter is incremented by the processor at every clock cycle and thereby provides an extremely accurate measurement of timing differences, making it very valuable for determining memory access delays. While `RDTSC` makes attacks easier to implement, it should be noted that there are also other ways of getting high resolution timers for the same purpose, for instance a thread that increments a single integer as fast as possible. While this method does not have the same reliability, it seems to work well enough for actual attacks in practice [20].

6. Analyzing the speculative execution context

The Meltdown vulnerability shows that the *out-of-order context* that speculatively executed code runs in, differs significantly from the regular, well-documented *in-order view* programmers are normally working with. After out-of-order execution has completed, the reorder buffer ensures that the instructions retire in the correct order so that the actual result is not influenced. It is however also possible for faults and violations to occur during out-of-order execution, and in these cases, some behavior has been shown to differ. The reorder buffer ensures that these divergences from expected behavior do not influence the in-order result of a computation by throwing away the results of instructions executed after an exception occurred in regard to the in-order sequence. However, these instructions have still been executed at some point, even if their results are not made visible.

The authors of Meltdown discovered that one of the differences between out-of-order execution and in-order execution is that it is possible to read kernel memory and continue executing instructions for a few cycles before the violation is detected. It seems likely that there are additional differences in the behavior of code during speculative or out-of-order execution. For the purpose of this research, we will refer to this as the *speculative context* and attempt to discern differences in behavior of assembly instructions when running in this context.

6.1. Testing instruction behavior

In order to test the behavior of instructions in the speculative context, we must first write code that creates a branch misprediction in order to execute code speculatively. We can then leak the results of our tests through selective cache loads and discern differences from expected behavior. As branch predictor behavior differs between processor generation, we focus on the Haswell test system used for this research. To intentionally induce a branch misprediction, we create a program that contains a loop which will run n times. During each iteration, an `if` condition in the loop either executes code or jumps over it. The condition is set up to ensure that the code will execute during the first $n - 1$ iterations and

be skipped during the last. Additionally, the condition depends on memory that will be flushed from cache during each iteration. This setup ensures that the code inside the `if` will be speculatively executed during the last iteration and subsequently be rolled back because the CPU will eventually detect the misprediction. We can then leak data from this speculative execution context by selectively accessing one of two addresses during the last iteration. The first signifies a binary one, the second a binary zero. After speculative execution is complete, we can use the `RDTSC` instruction to approximate how many cycles access of each memory location takes. If an address had been accesses during speculative execution, it should already be in the cache hierarchy, making later accesses significantly faster.
The results of the tests run using this setup on a Haswell CPU are described in the following sections.

6.1.1. Memory read

It was possible to directly and accurately read kernel memory from userland as described in the Meltdown paper. We found that only memory which already resides in the cache hierarchy can be read. This is most likely because access to uncached memory locations takes so many cycles that the CPU will detect the branch misprediction and abort speculative execution before the load can complete. Additionally, it seems to also be possible to read from completely unmapped memory locations. During our tests, these reads always resulted in the value zero, but code continued to execute speculatively after the access. We assume that page faults have a delayed effect, similar to segmentation faults. Of course, it is also possible to read every address of the normal process memory.

6.1.2. Memory write

Similar to the read tests, we performed a range of write operations to protected memory to determine possible differences. It is possible to write to kernel memory locations and continue code execution, but when attempting to read back the modified memory, we found that only the original value would be returned. It is possible to write to userland read-only memory. In this case, the changes are also reflected when attempting to read back the data. When writing unmapped memory, the modified location will always return the value `0xFFFFFFFFFFFFFFFF` when attempting to read it.

6.1.3. Memory execution

It is possible to jump to other memory locations and continue execution there as long as the instructions are valid and the memory is mapped with the executable permission. It is not possible to speculatively execute memory that lacks the executable flag, even if it is in userland. Furthermore, we conducted additional tests regarding access of userland data from speculatively executed kernel code. *Supervisor mode* *execution prevention* (SMEP) is a hardware feature found in in modern Intel processors that prevents the execution of userland code from Ring 0 and has been introduced to make privilege escalation exploits harder. We conducted tests to determine if SMEP also stops speculative execution. We implemented the same test software described above as a Linux kernel module and tested it on an install of Ubuntu 16.04. We found that it is not possible to speculatively execute userland code from ring 0 if SMEP is active. It seems that this protection mechanism takes precedence over speculative execution. Unfortunately, our tested processor did not support *supervisor mode access prevention* (SMAP) and speculative userland read/write protection from Ring 0 could not be tested.

6.1.4. Additional findings

We found that many instructions – mostly those that require a higher privilege level to execute – immediately stop speculative execution. Contrary to read or write permission violations which didn't stop execution under any circumstance in our tests, any instruction placed afterwards will not run. This includes for instance any and all accesses to control registers (e.g. `MOV RAX, CR0`), machine specific registers (`RDMSR`, `WRMSR`), the global descriptor table (`LGDT`, `SGDT`) and `SYSCALL`.

7. Exploitability of writes in speculative execution

Considering the flaws Meltdown and Spectre are based on, we considered the possibility to exploit not only memory *reads* but also *writes* during speculative execution. One of the core issues in Meltdown is that the permission check does not have immediate effect when reading kernel memory from cache during speculative execution. Before the access violation aborts and rolls back speculative execution, additional instructions that leak the contents of the read can be executed. This allows an attacker to leak arbitrary data from CPU cache.

Based on this, it seems reasonable that attacks based on cache writes could also exist. It should be possible to write arbitrary data to memory locations and execute additional instructions before the access violation aborts speculative execution. To exploit this in a real-world scenario, one would need to find a way to influence code outside of speculative execution with these modifications. It might be possible to manipulate cache lines inside speculative execution on one core in a way that, also speculatively executed code, running on another core at the same time will be influenced. Most likely this would be detected by the CPU, and the corresponding changes rolled back, but the same techniques used in Meltdown leaks could be applied to read arbitrary memory in other contexts.

7.1. Attack scenario example

To give an unlikely but easy to understand example, it could be possible to modify shared code backed by the same physical memory during speculative execution running on core A. If core B runs speculative execution within a victim process that uses the same shared code, at the same time, those changes might be visible. As soon as the access violation from core A is processed by the CPU, all changes would be rolled back so if this happened accidentally, no corruption would occur. If an attacker were to specifically craft the code, they could read arbitrary memory in the victim process and leak it through cache-loads like in Meltdown. The described scenario is depicted in Figure 7.1This exact scenario is not very likely to work, but it should be a simple example of the type of attack being proposed here: since it is possible to modify arbitrary memory during speculative execution and continue executing additional instructions before the modifications are rolled back, it might be possible to influence code running on other cores at the same time. This is especially true for hyper-threading enabled CPUs as logical cores share many resources that physical cores do not.

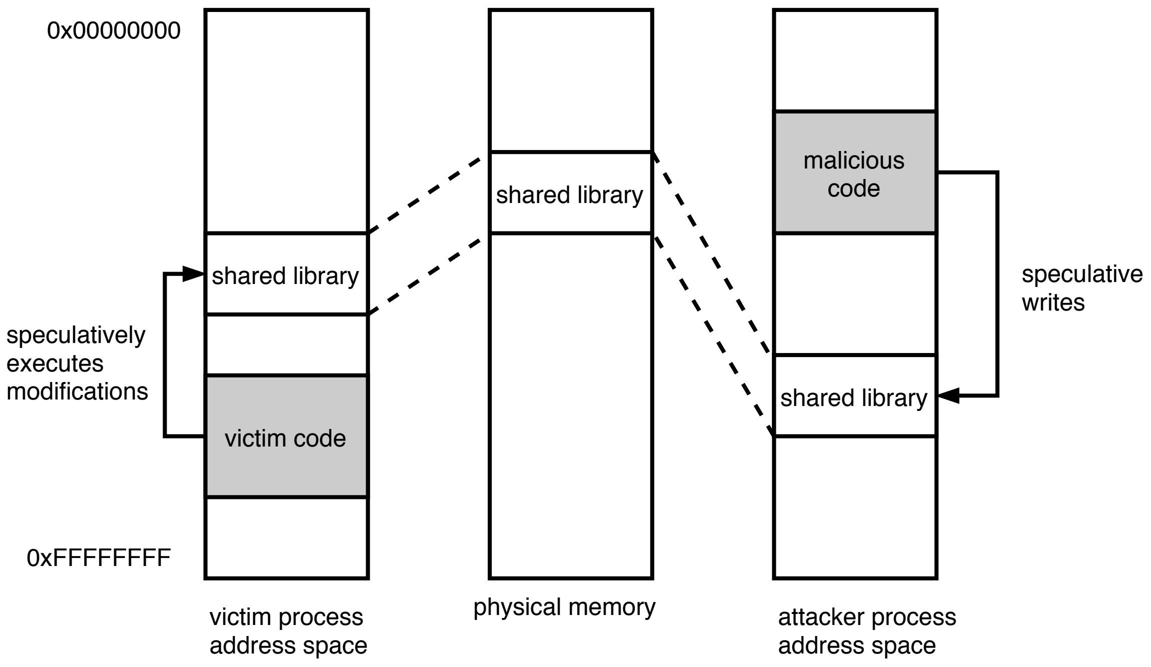

Figure 7.1.: Speculative write attack scenario

In order to evaluate the probability of this scenario, it is necessary to gain a deeper understanding of CPU caches on the Intel platform. The following sections give the necessary background information to evaluate the probability of the proposed attack.

7.2. L1 cache access

On recent Intel CPUs, the *L1 cache* is virtually-indexed and physically-tagged[1], this means that the offset bits of the virtual address are used to calculate L1 array locations. Since the offset bits are the same in virtual and in physical addresses, this can happen before the virtual address is translated to a physical address by the Translation Lookaside Buffer (TLB) or actual walking of the page tables (PT). After that, cache tags are compared to the actual, fully translated physical address to determine the correct entry [21]. As a consequence, this means that L1 cache relies on the actual, physical address of a memory location. If an attacker finds a way to poison this cache, other threads should see the modification when they access memory backed by the same physical address. This scenario happens quite often in modern

[1]https://www.realworldtech.com/sandy-bridge/7/

operating systems. Shared libraries are commonly loaded into memory once and then mapped into the virtual address space of multiple processes. Since L1 cache is virtually-indexed and physically-tagged, poisoning the cache entries of such shared pages should also affect other processes since they are backed by the same physical addresses.

7.3. Hyper-Threading

On modern Intel CPUs, each physical core has its own L1 and L2 cache that is not shared with any of the other cores. Only L3 cache is shared between the cores of a single chip. This makes it more unlikely that the proposed vulnerability could occur in code running on different physical cores. As each of them has a unique L1 and L2 cache, the likelihood that changes made to memory during speculative execution could be visible on any of the other cores [9].
However, if CPUs support *Hyper-Threading* (HT), those caches are shared between two logical cores. Hyper-Threading is a technology developed by Intel to execute multiple logical threads on the same physical CPU. From the point of view of the operating system, the logical threads look like physical CPU cores. In the actual hardware however, two threads run on the same physical core which has some resources duplicated in order to allow for this behavior. For instance, each logical core has its own replicated set of registers. Other resources, such as certain buffers – which will be described in more detail later – are statically allocated between the two logical cores. There are also competitively-shared resources, including the cache hierarchy [9]. This means that during HT operation, both logical cores will share the same physical L1 and L2 caches. Taking this into account, it seems more likely that it might be possible to attack the sibling Hyper-Threading core with modifications made to caches during speculative execution.

7.4. Load and Store Buffers

While often ignored in literature for the sake of simplicity, reads and writes are actually not immediately serviced by the L1 cache even if they are contained therein. To further improve performance, Intel introduced load and store buffers. Writes are first queued into buffers and are not immediately committed to L1 cache. This allows to further improve performance by not having to wait for the cache write to complete but introduces a new problem. If the same memory location gets read again, the change might not be visible in the L1 cache yet. Consider the code in Listing 7.1.

Listing 7.1: reading modified data

```
mov [x], rax
mov rbx, [x]
```

The expected result would be that both `RAX` and `RBX` contain the same value once this code has been executed. However, if the modification made in the first instruction is only added to a buffer and not actually committed to L1 cache, the second instructions would read the old value and produce incorrect results. A simple workaround for this would be to stall the CPU until the buffer has been fully committed to the cache, however this would severely decrease performance in these situations. The solution implemented by Intel is called *store forwarding*. Reads will always first be satisfied from store buffers and only afterwards from the cache hierarchy. This complicates chip design but greatly simplifies software written for the CPU as the buffers become completely transparent to the programmer and do not have to be taken into account.

According to documentation provided by Intel [9], load and store buffers are statically allocated between two logical cores when Hyper-Threading is active and are not shared. In the context of our proposed attack vector, this means that writes only committed to buffers cannot be visible on other logical cores by hardware design.

7.5. Analysis of hardware design

To sum up the last chapters, there are two important pieces of information relevant to the practicality of the proposed attack. First, L1 cache entries are assigned at the level of physical addresses and are shared between logical cores when hyperthreading is active. From that point of view, changes made to L1 cache during speculative execution could very well be visible to other logical cores, if they are also undergoing speculative execution. The second important part is that writes are not directly applied to L1 cache but rather committed to a store buffer. To execute the proposed attack, it would be necessary to find a way to commit store buffer modifications to L1 cache during speculative execution but this seems to be prohibited by design.

A blog post from 2013 suggests that memory snapshotting is performed by using load and store buffers[2]. Modifications made during speculative execution are only ever written to store buffers which are then discarded if a roll back occurs. As described previously, this design would make our proposed attack impossible as these buffers are never shared between logical or physical cores.

[2]https://fgiesen.wordpress.com/2013/03/04/speculatively-speaking/

7.6. Practical tests

We performed various tests to ensure that no detail has been overlooked where we attempted to see if modifications to an address made during speculative execution were visible on another core also undergoing speculative execution at any time. Our test setup included a thread that continuously changed the value of stored at an address during speculative execution without ever retiring the modification. A second thread simultaneously attempted to detect the magic value written by the first thread during speculative execution and leak the result through a cache side-channel. As expected by our analysis of the hardware design, we were unable to detect any modifications on the second core with statistically significant frequency.

7.7. Conclusion

As demonstrated by both our analysis of the hardware specification as well as manually performed testing, we are reasonably certain that the proposed attack does not work with the hardware version we tested. It might be possible with other revisions, microcode or additional knowledge about undocumented parts of the Intel CPU but our setup does not seem vulnerable to the proposed attack. It is likely that Intel hardware is not vulnerable because of the way load and store buffers are split between Hypter-Threading cores.

8. Impact on other security boundaries

While being mostly known for breaking isolation between userland and kernel as well as between operating system processes, attacks based on the Meltdown and Spectre vulnerabilities have also been demonstrated to break other isolation schemes.

8.1. Software guard eXtension

SgxPectre [6] is a variant of the classical Spectre attacks that targets code running inside an Software Guard eXtension (SGX) enclave. This is especially worrisome considering that SGX has been specifically designed to allow for the creation of private memory regions that are protected from access even by code running with higher privileges.
An SGX secure enclave is an isolated execution environment that should provide both confidentially and integrity to the application inside, protecting it from all other software running on the system. Even before Spectre, researchers demonstrated that a cache side-channel could be established with a *Prime+Probe* attack in order to extract secret data from an enclave [22]. The SgxPectre attack takes this research a step further by poisoning branch targets used by the code inside the enclave. By carefully manipulating the branch target buffer with malicious code running on the same system, branches taken during execution of the enclave code can be influenced. As demonstrated by the researchers, secrets can then be leaked through selective cache loads, similar to traditional Spectre attacks.

8.2. System Management Mode

Researchers from Eclypsium implemented a Proof of concept [1] that showed how Spectre can also be used to read data from *system management RAM* (SMRAM). System management mode, sometimes referred to as Ring -2, is a high-privilege mode of operation on Intel CPUs that makes use of a dedicated memory region commonly referred to as SMRAM. After initial configuration during system boot, this

[1]https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/

memory region normally gets locked through dedicated hardware mechanisms and can then no longer be accessed, not even by the operating system kernel.

8.3. Conclusion

That both SGX and SMRAM are impacted by Spectre shows how far-reaching the effects of low-level architectural vulnerabilities can be. Both mechanisms were independently developed and designed to protect memory contents from all other software running on a system. Even though they worked on different principles, Spectre attacks can be used to break them without much customization.

9. The return branch predictor

The initial technical blogpost [17] the Google Project Zero team published on the Meltdown and Spectre attacks noted that there are three different branch prediction mechanisms in modern Intel CPUs. Two of these, the generic branch predictor and the specialized indirect call predictor, have been reverse engineered thoroughly by the Google Project Zero team and are described in detail in the blogpost. The third however, the return branch predictor, has not been analyzed in detail yet. The goal of the research conducted for this chapter was to reverse engineer the behavior and demonstrate the possibility of influencing the outcome of the return branch predictor on a Haswell CPU.

9.1. Function-based control flow on x86

In order to be able to properly analyze the hardware-based features to improve performance of control flow based on function calls on the Intel architecture, it is first necessary to understand the process they are trying to improve.

Software often contains functionality that is required in multiple parts of the program. In order to decrease program size, it is desirable to reuse the same code rather than to store duplicates. The way this is generally handled in source code is the introduction of functions. From an abstract perspective, blocks of code are assigned a name and then referred to wherever needed. When a function is called, execution temporarily jumps to the code of the function and then resumes wherever it had been called from. In order to do this, it is necessary to keep track of the address where the function had been called from in order to be able to jump back there. The x86 architecture provides dedicated instructions to make this process faster and easier to implement. The `CALL` instruction is used to redirect execution flow to a function. It acts like a normal jump but additionally pushes the address after the `CALL` statement on the stack to store where execution will have to resume later. The stack is a specialized memory region dedicated mostly to temporary data such as local variables. When the function has completed, it executes the `RET` instruction which is typically the last instruction in every function. It pops the top-most value from the stack and resumes execution there. While `CALL` and `RET` are normally used in pairs, this behavior is not enforced. `CALL` simply jumps to the given location and pushes the return address on the stack while

RET takes whatever value is currently the top of the stack and jumps there.
This means that the value on the stack could also have been modified in the meantime, resulting in the real execution flow not matching the expected `CALL`/`RET` pair. Function calls could also be implemented with only `MOV` and `JMP` instructions but `CALL` and `RET` provide a convenient and fast way, so they are usually used by modern compilers in most cases. Since the presence of `CALL` and `RET` is a strong indicator of how execution will flow, it makes sense to include a specialized predictor that is able to take advantage of those hints.

9.2. Security relevance

The existence of this specialized return branch predictor is described in the Intel 64 and IA-32 Architectures Optimization Reference Manual where it is explicitly stated in section 3.4.2.1 that the Branch Prediction Unit (BPU) contains a 16-entry Return Stack Buffer (RSB) which enables the BPU to accurately predict `RET` instructions [23].
In the simplest case, this would mean that the CPU pushes the addresses of all encountered `CALL` statements on the RSB, pops an address off upon executing a RET instruction and continues speculative execution at that address. Such a simple design would however incur many performance penalties. As the management of process contexts is implemented in the operating system kernel rather than the hardware itself, the RSB would have to be flushed on each process context switch or it would result in many inaccurate predictions. We therefore assumed from the beginning that the actual implementation would be more complex than a simple "shadow stack" and include some address-based caching mechanism to differentiate between `CALL` and `RET` pairs in different process contexts.
Depending on the way it is implemented, knowledge of the return branch predictor could be very valuable for security research and might allow for a wide range of different attacks. If a way could be found to dump the return stack buffer, it would be very likely that it might contain addresses from other process contexts or even the kernel and thereby allow for generic ASLR/KASLR bypasses. Additionally, attackers seeking to break out of the confinements of a virtual machine could gather valuable information about the host machine kernel. If a way were found to influence the return branch predictor in other contexts, attacks similar to Spectre would be possible. If a way to reliably redirect the speculative target of return instructions were to be found, this would be especially critical for the protections provided by the Retpoline workaround as it relies on attackers being unable to misdirect RET instructions.

9.3. Experiments

We conducted a range of experiments with the return branch predictor on a Haswell CPU to shed more light on its behavior. The goal was to determine if the outcome can be influenced in any way.

9.3.1. Experiment 1: Demonstrating the existence of the RSB

As the first step we implemented a test program that would demonstrate the existence of the return stack buffer. For this, we crafted simple assembly code where the `CALL` instruction's corresponding `RET` would not return to the initial `CALL` but rather somewhere else. This results in the code directly after the `CALL` instruction being unreachable during real execution. We placed code there that would access a memory location not present in cache and later measure access time of that address similar to the cache side-channel in Meltdown/Spectre. If access is fast, it proves that speculative execution had executed the code after the `CALL` statement because the RSB predicted the `RET` would return to the `CALL`. The code used can be seen in Listing 9.1.

Listing 9.1: Forcing mispredictions

```
call A
ret

A:
        call B
        ret
B:
        call C
        ; the code here is unreachable
        mov rsb, [rsb] ; access the side-channel location
        jmp $ ; endless loop to ensure speculative execution ends here
C:
        pop rax ; remove return address of "call C" in B
        ret ; returns directly to A
```

To be able to better describe the results of the experiment, we give names to both possible paths. We call the unreachable code segment in `B` the "call-ret path" because the only way to reach it is by simple matching of `CALL` and `RET` statements without regard for the `POP` instruction in `C`. The actual path that is executed shall be named "real path" as it requires to keep track of the stack contents and is the one that will be really executed.

We placed instructions in the call-ret path that would access a memory location which had been removed from CPU cache with the `CLFLUSH` instruction beforehand. After executing the code, we timed access

to that address with the `RDTSC` instruction. Memory found in the cache would lead to an access time of around 20 cycles while uncached addresses resulted in over 150 cycles, showing a clear difference.
The results of this experiment were as expected. In almost every round of execution, the call-ret path is speculatively executed. This clearly demonstrates that the return branch predictor matches `CALL` and `RET` statements without regard for the real stack contents. If there were no RSB mechanism, the CPU would have no way to predict the call-ret path, demonstrating that a shadow stack or a variant thereof must exist.

9.3.2. Experiment 2: Memory location dependencies

In their research, Google Project Zero demonstrated that the generic branch predictor only uses the lower 31 bits of the address of the last byte of the source instruction to determine the target address. In our next test we determine if the RSB records the full 64 bits of the CALL addresses or only a subset thereof.
We modify the code from experiment 1 to map the `CALL` instruction to a different memory region than the corresponding `RET` instruction and observe if the return branch predictor still predicts the call-ret path. If the RSB only records the lower 31 bits of the addresses like the generic predictor, we expect prediction of the call-ret path to fail if the addresses of `CALL` and `RET` differ in the 32nd bit.
We iterat over all possible combinations of CALL and RET addresses, starting with the address`0x1000` and then continuously shifting it left by one bit until we reach `0x400000000000` (the 47th bit set to 1).
Interestingly, this experiment demonstrates some unexpected anomalies in the return branch prediction.

9.3.3. Result 1: RSB length

We found no instance where the call-ret path prediction failed based on the distance between the CALL and the RET address. Branch prediction succeeded even in the most extreme testable cases with the addresses being `0x0000000000001000` and `0x0000400000000000`. This means that the RSB stores at least 47 bits but most likely the full 64 bits. Since higher addresses cannot be mapped in userspace, further experimentation would require kernel modifications.

9.3.4. Result 2: call-ret path mispredictions

We found a small subset of address combinations where prediction of the call-ret path failed during every run. A simple pattern was not immediately visible, but we found that all failures can be put in one of two categories.

Category 1

CALL address	RET address	bit shift	call-ret pred.
0x0000000000002000	0x0000000000400000	09	0
0x0000000000004000	0x0000000000800000	09	0
0x0000000000008000	0x0000000001000000	09	0
0x0000000000010000	0x0000000002000000	09	0
0x0000000000020000	0x0000000004000000	09	0
0x0000000000040000	0x0000000008000000	09	0
0x0000000000080000	0x0000000010000000	09	0
0x0000000000100000	0x0000000020000000	09	0
0x0000000000200000	0x0000000040000000	09	0
0x0000000000400000	0x0000000000002000	09	0
0x0000000000800000	0x0000000000004000	09	0
0x0000000001000000	0x0000000000008000	09	0
0x0000000002000000	0x0000000000010000	09	0
0x0000000004000000	0x0000000000020000	09	0
0x0000000008000000	0x0000000000040000	09	0
0x0000000010000000	0x0000000000080000	09	0
0x0000000020000000	0x0000000000100000	09	0
0x0000000040000000	0x0000000000200000	09	0

Table 9.1.: 9 bit shift predictions

The first of these is when the addresses are shifted by exactly 9 bits. The behavior was not observed with any other shift widths. Table 9.1 shows all of the address combinations where we observed this anomaly. While the cause for this behavior is not clear immediately, it can be observed that the bits are the same as the collisions that the generic predictor cannot differentiate. The colliding bits observed by Google Project Zero are shown in Table 9.2.

Category 2

The other category where the call-ret path cannot be predicted is when both of the addresses are larger than 2^{31}. While the full list is too long to include in this paper, an excerpt demonstrating the anomaly can be seen in Table 9.3. The third table shows by how many bits the first address is shifted compared to the second while the last column shows during how many of the 1000 test iterations the call-ret path had been predicted. Prediction starts to fail in the line where both addresses are larger than 2^31. The same can also be observed for all other combinations not pictured here.

bit A	bit B
0x40.0000	0x2000
0x80.0000	0x4000
0x100.0000	0x8000
0x200.0000	0x1.0000
0x400.0000	0x2.0000
0x800.0000	0x4.0000
0x2000.0000	0x10.0000
0x4000.0000	0x20.0000

Table 9.2.: Generic predictor collisions

9.3.5. Analysis

The results of this test case clearly demonstrate that the actual return branch prediction uses not only the RSB but also another mechanism. We assume that this is most likely to avoid context-switching based performance penalties.

We can see that the address folding algorithm used by the generic branch predictor also seems to be in use here, however we can currently not be sure in what way. While the RSB itself clearly stores more than 31 bits of the addresses, the folding algorithm seems to be able to only process 31 bits of input. The unexpected behavior seems to occur when the input to the folding algorithm consists only of zeroes.

This happens when both addresses are larger than 2^31 because in that case the upper bits are simply cut, leaving only zeroes for both addresses. As demonstrated by Google Project Zero, certain bit combinations cannot be differentiated by the algorithm. We find it highly likely that these bits are XOR-ed together, resulting again in input to the folding algorithm containing only zeroes.

While the experiment demonstrated that there seems to be an edge case when the input to the folding algorithm consists of only zeroes, further testing is required. It currently seems as if both the address of the `CALL` as well as the `RET` statement are inputs to the branch prediction. This however seems unlikely as the purpose of the branch predictor is to determine the address of the `CALL` statement given only the address of the `RET` statement. In this case, it is not possible for the result of the algorithm to be part of the input.

CALL address	RET address	bit shift	call-ret pred.
0x0000000000001000	0x0000400000000000	34	894
0x0000000000002000	0x0000400000000000	33	895
0x0000000000004000	0x0000400000000000	32	896
0x0000000000008000	0x0000400000000000	31	895
0x0000000000010000	0x0000400000000000	30	896
0x0000000000020000	0x0000400000000000	29	895
0x0000000000040000	0x0000400000000000	28	899
0x0000000000080000	0x0000400000000000	27	896
0x0000000000100000	0x0000400000000000	26	897
0x0000000000200000	0x0000400000000000	25	896
0x0000000000400000	0x0000400000000000	24	895
0x0000000000800000	0x0000400000000000	23	895
0x0000000001000000	0x0000400000000000	22	897
0x0000000002000000	0x0000400000000000	21	897
0x0000000004000000	0x0000400000000000	20	898
0x0000000008000000	0x0000400000000000	19	897
0x0000000010000000	0x0000400000000000	18	897
0x0000000020000000	0x0000400000000000	17	896
0x0000000040000000	0x0000400000000000	16	895
0x0000000080000000	0x0000400000000000	15	0
0x0000000100000000	0x0000400000000000	14	0
0x0000000200000000	0x0000400000000000	13	0
0x0000000400000000	0x0000400000000000	12	0
0x0000000800000000	0x0000400000000000	11	0
0x0000001000000000	0x0000400000000000	10	0
0x0000002000000000	0x0000400000000000	09	0
0x0000004000000000	0x0000400000000000	08	0
0x0000008000000000	0x0000400000000000	07	0
0x0000010000000000	0x0000400000000000	06	0
0x0000020000000000	0x0000400000000000	05	0
0x0000040000000000	0x0000400000000000	04	0
0x0000080000000000	0x0000400000000000	03	0
0x0000100000000000	0x0000400000000000	02	0
0x0000200000000000	0x0000400000000000	01	0

Table 9.3.: Call-ret prediction failures

9.3.6. Experiment 3: Introducing training loops

To further analyze the behavior of the return branch predictor, we tested how the history of previous branches influence further prediction of different addresses.
We started by running 100 training iterations where the `CALL` instruction had been placed on address A and the `RET` instruction on address B. After that, we ran a single round where the `CALL` instruction is on address C and the `RET` instruction on address D. We then determined if the training runs with addresses A and B had any influence on the results of the prediction of the `RET` between C and D.
We then tested this with different values for the training addresses A and B.
We started by picking random training addresses that both had multiple 1s and 0s in different locations. We found that this did not influence the outcome of the call-ret path prediction in any way. We observed the same anomalies where the prediction did not succeed as before.
An interesting result is produced if the only bits set in the training addresses are in bits higher than 31. This results in the input to the folding algorithm being only zeroes during the training phase. When the actual `CALL` and `RET` test is executed, the return predictor fails to predict the call-ret path in every instance except for when the `RET` address is placed on `0x1000`, independently of where the `CALL` address is.
Setting only a few 1s in both training addresses resulted in having some additional address combinations where branch prediction fails during the actual tests. Which that are depends on the bits set during the training phase, we could however not determine any pattern of which addresses start to fail when setting specific bits.
In summary, experiment 3 shows that the input to the branch prediction algorithm is most likely the history of recent branches as well as the address of the RET instruction. Initially it seemed as if the address of the `CALL` instruction is an input as well, however this was most likely implicit due to previous test runs. Since we ran every `CALL` and `RET` combination 1000 times, the previous runs most likely influence the branch history buffer (BHB) in a way that made it seem as if the CALL instruction had been input as well.

9.3.7. Conclusion

We have demonstrated that there is a possibility to influence the results of the return branch predictor on the architecture tested. While further research is required to determine the exact behavior of the algorithm, our results suggest that it is in some ways similar to the generic predictor. Reverse engineering undocumented CPU behavior is a time-consuming and error-prone process as it is hard to take all possible factors that could influence the code being executed into account. We hope that our results can be taken as a basis for further research into the exploitability of the demonstrated behavior.

10. Alternative side-channels

To the best of our knowledge, there are currently no other publicly known, feasible side-channels to leak data from speculative execution other than the widely employed cache loads. It seems likely that additional vectors that have not been found yet exist. They could have substantial impact on the exploitability of known issues such as branch target injection since they could make attacks feasible that were previously thought to be unexploitable because the required side-channels were too hard to establish. This section aims to explore possible ways to leak data from code executed out-of-order and provide points of reference for further research.

The goal is to determine additional side-channels that can be used to leak data obtained during out-of-order execution in way that they persist across a rollback.

10.1. Defining side-channels

To give a more detailed description, we presume that the CPU is in a microarchitectural state A at a given time. Then, a number of instructions I_m that generate a range of modifications M is executed out-of-order and the microarchitectural state becomes $A + M$. However, before this new state can be retired, an instruction I_v that generates an access violation is executed. Before retirement, the reorder buffer detects that the instruction order determines that I_v must be executed before I_m which means that the modifications M made by I_m have to be rolled back. The microarchitectural state is now $A + M - M$ which should result in A but actually produces a state A' since some implicit modifications remain. The differences between A and A' are the side-channels that can be employed to leak data from out-of-order execution.

In the classical Meltdown and Spectre attacks, cache-loads have been employed as the side-channel. Since data loaded into the cache hierarchy during the execution of code that is later rolled back will not be removed, this constitutes one of the aforementioned differences between A and A' that lead to side-channels. It becomes evident that cache loads might not be the only difference that exists, and further research is required to determine other possible attack vectors. It should be noted that different side-channels could also lead to new attack surface as they might allow for the exploitation of known

vulnerabilities in more constrained environments previously not deemed possible.
In order to analyze different approaches, a suitable test-environment is required. From the side-channel definition described above, we deduct that a proof of concept requires the following stages:

1. Preparation of a microarchitectural state A that is suitable for triggering of the side-channel.
2. Execution of a violation instruction I_v that ensures that the modifications of all subsequent code will be rolled back.
3. Execution of code I_m that creates one or more modifications $M_side - channel$ that can later be detected.
4. Code that detects the differences between A and A' and can extract $M_side - channel$ after other modifications have been rolled back.

10.2. Choosing the violation instruction

One of the challenges described in this approach is the choice of the violation instruction I_v. It is strictly required to ensure that none of the instruction in I_m will retire and by definition must create an invalid state, such as an access violation or a page fault, that leads to an exception. However, on the Intel x86 architecture, these errors are normally handled by the kernel which means that not only a transition to ring 0 occurs but also that a potentially large amount of uncontrolled kernel code runs between the generation of the side-channel and the extraction of the data. There is a large chance that this delay might impact the microarchitectural state in a way that modifies or destroys the side-channel and could lead to both false positives and false negatives. It is therefore desirable that the whole program can run in userland without actual transitions of privilege level.
One solution would be to ensure that the aforementioned code runs during a branch misprediction instead of using a violation instruction I_v. This would ensure that none of the instructions retire without requiring any exceptions. This approach has several downsides, most notably the increased complexity of the code and also vastly lowered portability. As the exact details of branch prediction changes between CPU generations, it would be required to write code that can intentionally cause mispredictions in a controlled fashion individually for every targeted model.

10.2.1. Restricted transactional memory

We chose to make use of restricted transactional memory (RTM) which is part of the Intel transactional synchronization extensions (TSX), similar to the approach described in the original Meltdown paper [16].

This instruction set extension is available beginning with the Haswell generation and therefore featured on most modern processors. During our preliminary analysis of Meltdown and Spectre, we found that RTM provides an easy to use, flexible and portable way of running out-of-order code and ensuring that it never retires without requiring privilege level switches or running uncontrolled, non-userland code.
As the name implies, TSX was introduced to allow for hardware-accelerated memory transactions. It allows to define a block of code that runs as a single atomic operation, meaning that all memory modifications made within that block will either be visible at once or be completely rolled back. For rollbacks, TSX requires the programmer to specify an alternative path that will be executed in these cases. Rollbacks can occur at any time, for instance as a result of context switches or memory pressure. It is also possible to manually abort a TSX transaction by executing a specific instruction. To test out-of-order execution, we create a transaction and immediately abort it. This ensures that none of the executed instruction will ever retire and allows us to easily create a portable, controlled environment that minimizes the amount of code running between creation and extraction of the side-channel.

10.3. Creating test code

To facilitate hardware-backed transaction, Intel introduced a range of new instructions. `XBEGIN` marks the beginning of a transactional block and requires the programmer to specify the location where execution should continue in case of a rollback. Similarly, `XEND` marks the point where the modifications should be made visible to other code. `XABORT` takes an error code as argument and can be used to voluntarily abort the transaction at any time.
Listing 10.1 shows a commented version of the general structure our test program has.

Listing 10.1: RTM-based speculative execution

```
run_test_rtm:
    ; code to prepare the microarchitectural state A before out-of-order
    ; execution to be inserted here
    xbegin rtm_done    ; start of the transcation, rtm_done will be
                       ; executed in case of a rollback
    xabort 1           ; the transaction is immediately aborted
    ; this code will only run during out-of-order execution and
    ; will never retire
    ; code that creates the side-channel to be inserted here
    xend    ; ends the transaction and ensures that the XABORT will
            ; be detected here at the latest
rtm_done:
    ; this code will be executed once the transaction has been aborted
    ; code to analyse the new architectural state A' and extract the
    ; side-channel information to be inserted here.
    ret
```

This structure allows great control over which code will retire and which will not without any ring 0 transitions and should work on all processors supporting TSX. This ensures that external pollution of the architectural state is kept to a minimum while also allowing for great portability that does not rely on any undocumented branch prediction behavior.

10.4. Performing tests

To ensure the validity of our approach, we implemented the original Meltdown attack on a vulnerable Linux machine and confirmed that it works as expected. Additionally, we tested a new approach based on timing differences.

10.4.1. Cache hierarchy side-channel

We extract a secret value through a cache-based side-channel. To leak one bit, a designated memory location is flushed from cache using the `CLFLUSH` instruction. Speculatively run code then accesses the memory location if the bit to be leaked is one or skips the access if it is zero.
Afterwards, the amount of cycles to read the memory location is measured and if it is below 150 cycles, it is assumed to be cached already, hence the leaked bit must have been one. This test is repeated n times for each bit of the secret value and a probability for the bit to be one or zero is calculated based on how many times either of those values appears during the test runs. We found that around 100 runs provide reliable results, but more are better as there might be short periods of times where all accesses are slow because of memory pressure.
Our test implementation of Meltdown was successfully able to read data in the syscall table from userland

10.4.2. Timing-based side-channel

To see if data can be extracted without relying on the cache-hierarchy, we tested and implemented a timing-based approach. The basic principle is to control how long out-of-order execution continues depending on the data that should be leaked. By using `RDTSC` to get a high-performance counter, it is possible to approximate how many CPU cycles the transaction takes. Code running inside the transactional block could then be written to take a long time if the bit that should be leaked is a one or complete very fast if it is a zero.

For this approach to work, it is necessary to be able to abort out-of-order execution at any time. Previous analysis conducted as part of this research and described in section 6 showed that there is a wide range of instructions that stop speculative execution and it seemed highly likely that the same behavior could be found in regular out-of-order execution as the two are very similar.
We implemented a proof of concept without speculative execution by writing assembly code that runs a very slow loop if the bit to be leaked is a one and skips the loop if it is zero. We then ran the code multiple times for each bit of a secret value and recorded the execution times. As expected, a clear difference was visible, and it was easy to extract the secret using only the execution duration of the code block. We then ran the same code using an immediately aborted TSX transaction as described above. We experimented with different amounts of runs for each bit and attempted to filter outliers where the CPU was preempted during execution but were unable to detect any meaningful differences in execution time. It seems that the cycle penalty induced by aborting the transaction is about the same as the amount of instructions that can be executed before abortion. In all cases, we monitored a constant minimum execution time of around 170 cycles that did not differ depending on the actual work done during the transaction.

10.5. Lazy FPU bug

The *lazy FPU flaw* is at the time of writing the newest publicly disclosed speculative execution side-channel attack. It was announced by Intel on 13 June 2018 and demonstrates yet another minor flaw that becomes a major problem for system security when combined with a side-channel for extraction of information from speculative execution. To demonstrate the versatility of the TSX-based approach described in Section 10, we implemented a proof of concept using this methodology.
At the time of writing, no publicly available proof of concept exists for this vulnerability and detailed descriptions are few and far between. In order to give a more complete overview of the current state of speculative execution attacks, we analyze the root cause of the vulnerability, implement and test a proof of concept on the Linux kernel and explain countermeasures. This section applies much of the theoretic knowledge presented in this paper to show how a practical attack can be researched and implemented.

10.5.1. Root cause

The *Common Vulnerabilities and Exposures* (CVE) number assigned to the Lazy FPU information leak is CVE-2018-3665 but both the MITRE CVE dictionary[1] as well as the NIST National Vulnerability Database (NVD) [2] list the entry as reserved for a specific security problem but give no further information

[1]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3665
[2]https://nvd.nist.gov/vuln/detail/CVE-2018-3665

at the time of writing. More helpful information is provided by the original Intel advisory[3] as well as Red Hat[4]. These articles make it clear that the cause of the vulnerability lies in deferred restoration of FPU registers on context switches.

Modern Intel CPUs contain a range of floating-point unit (FPU) registers which serve specific purposes such as performing accelerated *floating-point arithmetic* or *SIMD* (single instruction, multiple data) operations. They can greatly accelerate certain classes of calculations but also incur performance overhead. Every time the CPU switches between process contexts, all registers must be saved to main memory and later restored. FPU registers are often especially large and can therefore have a significant impact on the performance of context switches. To lower the penalty, Intel introduced a way of performing FPU register restores only when the process makes actual use of the registers. This ensures that processes which do not use the FPU registers will incur no overhead because the registers will not need to be restored for them.

The Intel 64 and IA-32 Architectures Software Developer's Manual [23] gives further details about the technical details of this feature. Section D.3.6.1 states that lazy restoring of x87 FPU register works through the *Task Switched* bit of the `CR0` register in conjunction with *Device Not Available* (DNA) interrupts. The Task Switched bit of the `CR0` register is set every time the CPU performs a task switch with supported hardware accelerated mechanisms and needs to be manually cleared with the `CLTS` instruction. When lazy FPU restoration is enabled, all access to an FPU register while the Task Switched bit is set to one will result in a DNA interrupt on line 7. The associated interrupt handler can then store the current register state, restore the old state associated with the process, clear the TS bit and resume execution where the interrupt appeared. This mechanism allows to defer FPU register restores to when they are actually needed and can therefore improve performance.

Normally, this would not pose any threats but in the context of speculative execution side-channel attacks, the potential impact on security becomes obvious. Because out-of-order execution allows to execute multiple instructions before exceptions and interrupts are handled, it should become possible to read FPU registers and leak their contents through side-channels before the DNA interrupt handler will be executed. Therefore, it should be possible to read another task's FPU registers before the currently running task's state is restored. To test this hypothesis, we implemented a proof of concept to see if information written to an FPU register in one process can be observed in another process during out-of-order execution.

[3]https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00145.html

[4]https://access.redhat.com/security/cve/cve-2018-3665, https://access.redhat.com/solutions/3485131

10.5.2. Proof of concept implementation

The setup consists of two independent programs, a *getter* that will attempt to continuously read the contents of an FPU register and leak it through a side-channel as well as *setter* that will write a magic value to the same register. For this proof of concept, we make use of the `XMM0` register which is part of the SSE2 (Streaming SIMD Instructions 2) supplementary instruction set but any other FPU register could be used as well.

The setter program is very simple, it uses the `MOVSS` assembly instruction to write a known, hardcoded magic constant into the `XMM0` register and then calls `sched_yield` to signal the kernel that another process should be executed. To increase the likelihood of winning the race conditions, this procedure is repeated in an endless loop.

The getter program is based on the Intel TSX code described in Section 10 and leaks the obtained information through a cache hierarchy side-channel. It uses the `CVTSS2SI` instruction to read the contents of the `XMM0` register into a regular register. It will then compare the value read to the known magic number written in the setter program. If the values are equal, it will read from a predetermined memory location to fetch it into the cache. By measuring access time of this location later on, it can be determined if the contents of `XMM0` were equal to the magic value or not. Figure 10.1 shows the process in detail. The measurement operation will be performed continuously and every 100000 rounds, the percentage of successful detections of the magic value will be printed. Excerpts of the code can be found in Listings A.1 and A.2 in the appendix.

The tested Linux 4.8 kernel supports both *eager* and *lazy* loading of FPU registers on context switches. The currently selected method will be printed during boot and can be changed with the eagerfpu kernel boot parameter. On our tested Ubuntu 16.10 installation, *eager* was the default option and *lazy* had to be manually activated. Commit messages[5] from Andrew Lutomirski shed light on this decision. Because recent processors have dedicated, optimized instructions for storing and loading the FPU state, the overhead is in practice not as large as one would expect. Furthermore, laziness could be overly optimistic because even glibc uses SSE2 nowadays and if many actual restores are required, eager fetching is faster than handling the DNA interrupt. Additionally, manipulation of the TS bit itself is relatively slow, so there is little reason to use the *lazy* strategy instead of *eager* on modern hardware.

This means that default installations of Ubuntu on relatively recent CPUs should not be affected by the lazy FPU attack in practice. We successfully tested and confirmed the vulnerability on an Ubuntu virtual machine with lazy context switches manually activated, running on a host system which used the eager strategy. This confirms that the vulnerability can occur in virtual machines, independently of weather the

[5]https://patchwork.kernel.org/patch/8095191/

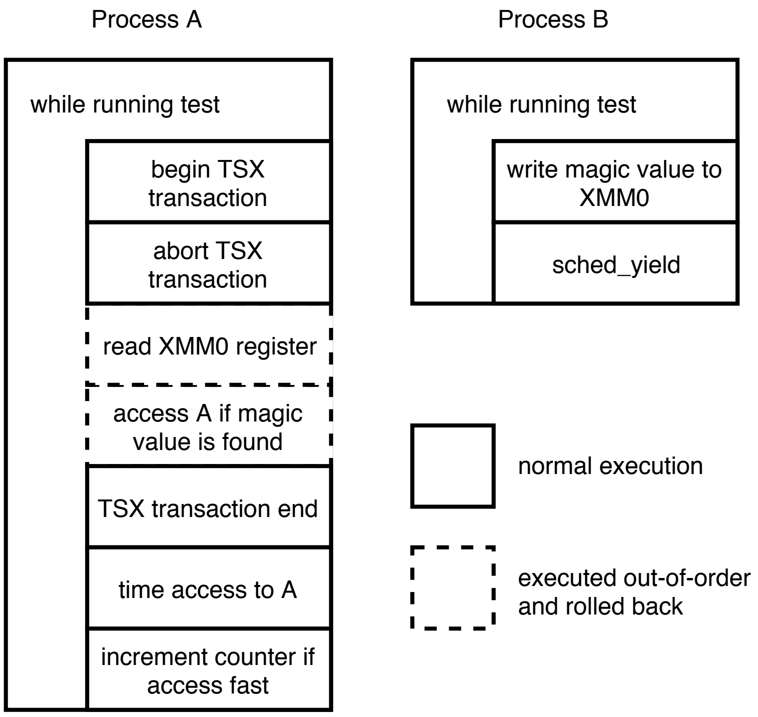

Figure 10.1.: Lazy FPU proof of concept

host is affected or not.

10.5.3. Test results

Our tested VM used two virtual CPU cores with the actual host having four cores and no hyperthreading. When running only the getter program without any setters, the number of detection of the magic value was as expected extremely low, typically ranging between 0.01% and 0.1% with many rounds placing on the lower end of the spectrum. Of course, these were all false positives, the occurrence of which can easily be explained by the nature of the cache hierarchy side-channel. Measuring access times and manipulating the content of the cache in a multi-threaded environment can never be perfectly controlled, a small number of false positives is always to be expected by the attack developer. We also found that the number of false positives increases inside a virtual machine compared to a bare-metal installation, suggesting that additional context switches make measurements fuzzier.
Interestingly, the addition of a single setter process had little to no impact on the results. We assume that this is dependent on the operating system scheduling algorithm. Only when adding a second setter and thereby matching the number of virtual cores could an obvious effect be observed. With one getter and two setters, successful detections of the magic value ranged between 20% and very close to 100%, depending on the individual run with each consisting of 100000 attempts. This demonstrates a clear

correlation and that it is indeed possible to bypass DNA interrupts and read FPU registers with speculative execution side-channel attacks, however the attack sill strongly depends on the environment it is run in, especially the OS scheduling algorithm. As a matter of circumstances, it could still be extremely hard if not impossible to reliably target the registers of a specific process.

10.5.4. Countermeasures

The countermeasure to this attack is to simply not use the lazy loading scheme and opt for a more secure eager fetching. On modern hardware, there should be no performance penalty as eager loading it faster and preferred anyways.

This attack demonstrates again very well the core issues behind speculative execution side-channel attacks. Similar to Meltdown, the problem is that enforcement of the security policy (in this case the interrupt) is delayed and not immediately handled. As long a side-channels that can leak information from speculative execution exist, interrupts cannot be seen as a valid strategy to protect data from being accessed.

11. Bypassing hardware breakpoints with speculative execution

Breakpoints are an important debugging tool and very helpful for determining the root cause of obscure software issues. There are two different types that have different use-cases: *software breakpoints* and *hardware breakpoints*.

11.1. Software breakpoints

Software breakpoints are represented by the `INT 3` instruction and can be placed anywhere in the program flow. Contrary to other interrupts which are encoded using two bytes, the opcode for software breakpoints is `0xCC`, a single byte which makes it easier to replace any instruction with it. If a programmer wants to set a software breakpoint, the debugger will replace the targeted instruction with `INT 3`, causing the program to generate a software interrupt upon execution. The kernel will then pause the debugee and notify the debugger of the breakpoint hit. Subsequently, the debugger of course has to replace the software breakpoint with the original instruction in order to avoid modifying program logic. Software breakpoints are widely used and have the advantage that any number of them can be placed at the same time, but they can also be easily detected by the program being debugged which makes them hard to use in malicious software that actively monitors code for modifications to deter reverse engineering. Furthermore, software breakpoints can only trap execution flow and cannot trigger on memory reads or writes. While being useful for debugging, the use of software breakpoints requires modification of the code which shall be monitored which makes them unsuitable for many reverse engineering tasks and for use in rootkits.

11.2. Hardware breakpoints

Hardware breakpoints work differently and have many advantages regarding their stealth capabilities. They are implemented at the hardware level through special debug registers. These privileged registers can only be read and written two at privilege level 0 and allow for the configuration of invisible hardware breakpoints. The registers `DR0` to `DR3` contain the linear addresses associated with their respective breakpoints. This also demonstrates the biggest limitation, there can only be four different hardware breakpoints at any time. `DR7` is the debug control register and can be used to selectively enable the four breakpoints and to specify the conditions on which they activate. It is possible to trigger a break on execution, data write and both data read/write. The last register, `DR6`, holds information about which debug conditions have occurred, its low order bits are set to one when a breakpoint triggers and before the debug exception handler is executed.

The big advantage of hardware breakpoints over software breakpoints is that they are completely invisible as they do not require memory modifications and that they can be used to monitor memory reads and writes. These features have been abused by rootkits in the past to stealthily monitor memory regions.

Longkit [24] uses this feature to make a small 8 byte region of memory that holds the debug exception handler invisible to the operating system. Most of the code of Longkit resides in the SMRAM which cannot be read by the operating system, even with Ring 0 privileges. To take control of the host OS, the rootkit replaces the debug exception handler with a minimal version that causes a System Management Interrupt (SMI) to execute the rootkit functionality whenever a hardware breakpoint is hit. `DR1` to `DR3` can then be used to intercept arbitrary kernel functions. `DR0` is reserved for the stealth module of Longkit. By using it to monitor reads and writes to the debug exception handler, the only small modification made to the kernel can be rendered invisible to antivirus software. When an attempted read is detected, Longkit will be triggered and replace the result with the old, unmodified value so none of the changes made to OS memory are actually visible. This ensures a high grade of stealth because it is impossible to directly read any of the memory where Longkit resides or where it has made modifications.

11.3. Implementing a proof of concept

Because hardware read/write breakpoints only trigger when the respective instructions retire, it should be possible to read the memory regions protected by them with a speculative execution side-channel attack. This could prove helpful to detect extremely stealthy malware like Longkit.

In order to test this hypothesis, we implemented a proof of concept kernel module that reads 8 bytes of the debug exception handler and compares them to a known good value. First, the location of the debug exception handler is determined by parsing the *interrupt descriptor table* (IDT). The location of the IDT itself is stored in the *interrupt descriptor table register* (IDTR) that can be read with the `SIDT` instruction.

Then, 8 bytes of the determined location are read using two different methods. First, the data is read speculatively with the TSX method described in Section 10. Then, the same memory location is read normally with a call to `memcpy`. The two results can then either be compared to a list of known good and bad values or simply to each other in order to detect inconsistencies which normally should never occur.

A detailed description of the deteciton method is depicted in Figure 11.1.

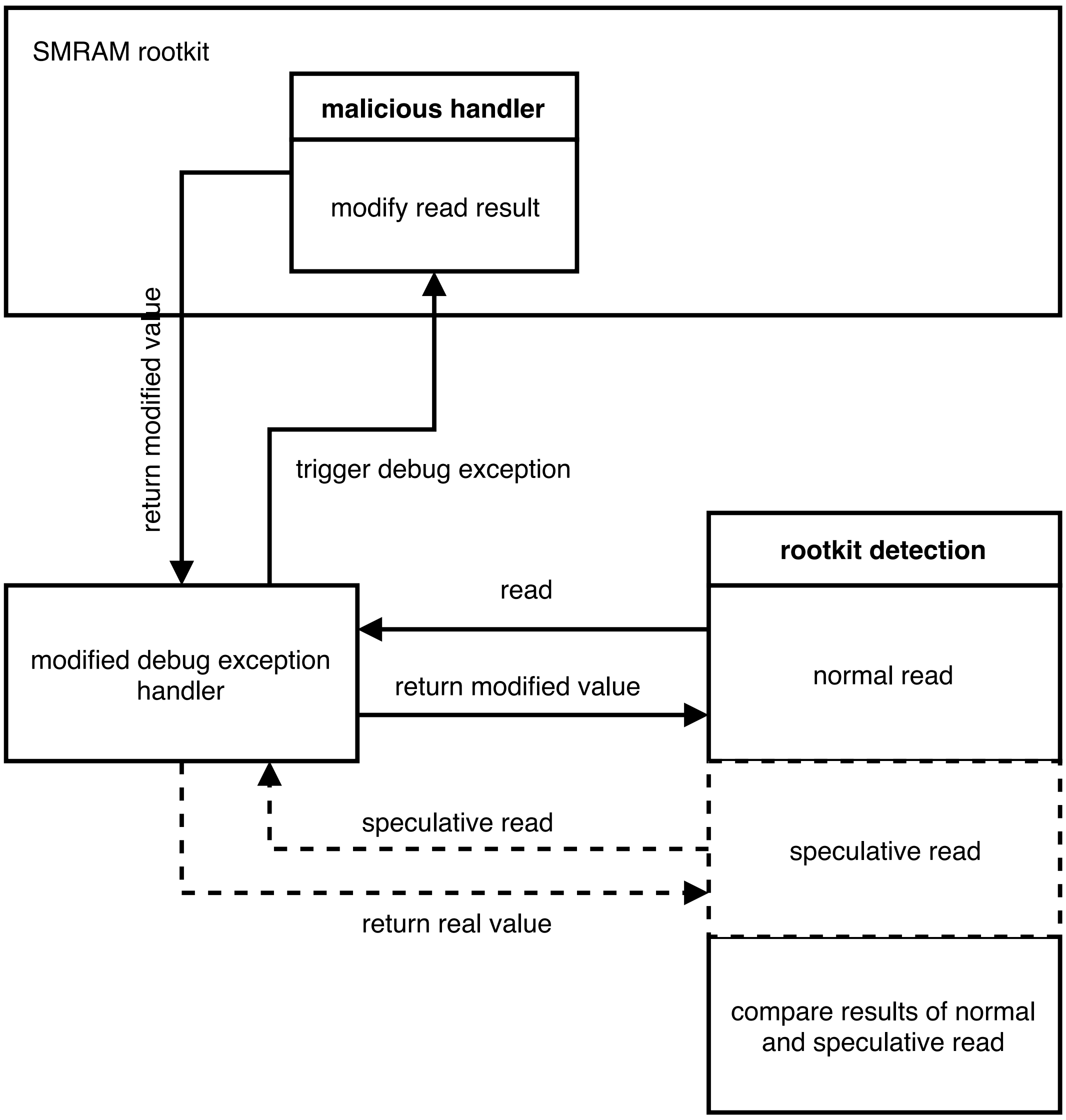

Figure 11.1.: Rootkit detection process

We tested the kernel module inside a *QEMU* virtual machine that was booted using a modified version of *SeaBIOS*[1] with the Longkit payload. We found that the proposed methodology can be reliably used to read memory protected by debug registers without triggering them. Excerpts of relevant parts of the code can be found in Listing A.3 in the appendix.

[1]https://github.com/coreboot/seabios

This demonstration of using speculative side-channels to bypass hardware breakpoints shows that the technique can also be employed for other purposes. Rootkit detection software could detect the presence of otherwise extremely stealthy malware. However, malware authors might also be able to develop new anti-debugging techniques that make it harder for specialists to analyze malicious software as they can no longer rely on hardware breakpoints always triggering when a memory region is being read.

12. Long-term solutions to speculative execution side-channel attacks

In the Sections 3 and 4, we describe currently available protection mechanism that provide short-term mitigations against the risks posed by speculative execution side-channel attacks. However, these software-based protections cannot be seen as a comprehensive solution but rather a workaround born of necessity. They incur significant performance issues for certain workloads and new bypasses are regularly discovered with new variants of Spectre, indicating that they rather fix symptoms than the underlying problems.

12.1. Approaches to a solution

A holistic solution to this class of attacks will require modifications to processor design, very likely significant ones. In this section we describe three different approaches that address the core issues identified and provide an evaluation of required modifications.

12.1.1. Removal of offending features

The most direct approach to protect against data leakage from Spectre and Meltdown attacks would be the removal of all speculation, including any out-of-order execution. If executed instructions were to immediately retire and no assumptions whatsoever about future program flow were to be made, none of the flaws described in this paper could exist. While certainly representing a comprehensive solution, this approach can be seen as theoretical at best. Out-of-order execution and closely related features such as branch prediction make up a large part of recent advancements in processor design. Without it, performance would degrade to a completely unacceptable level. An architectural redesign to rely less on out-of-order execution is also unlikely if not impossible. For these reasons, this approach can be considered too impractical as a long-term solution.

12.1.2. Preventing access to out-of-context information

The second solution is based on the removal of any mechanism that allows an attacker to gain access to any information outside of her context. As a simple example, under this paradigm, Meltdown attacks would be prevented by ensuring that instructions executed out-of-order cannot access memory they lack permissions for. While straightforward for Meltdown, enforcing this paradigm is more complicated for Spectre. From an architectural point of view, this would require all microarchitectural data structures to be exclusive to their respective contexts. For instance, every process would need its own, logical branch prediction table to ensure it cannot influence the tables of the other processes. This can easily be achieved by simply clearing the data structures on each context switch, but this is again likely to induce unacceptable performance degradation. At the very least, a partitioning algorithm would be required that can exclusively assign resources to contexts. Implementation of this could be troublesome both from a hard- as well as a software perspective. To achieve performance similar to the current design, a major increase in total size for all microarchitectural data structures would most likely be required. With operating systems usually running a very large number of processes at the same time, selective flushing of structures would most likely be unavoidable. Further research would be required to determine optimal sizes that can achieve similar performance to the current design. In order to introduce such an isolation scheme in a relatively efficient way, it would most likely be necessary to modify current operating systems to take advantage of them. In some cases, security contexts might not be immediately obvious from a low-level perspective (e.g. a JavaScript sandbox that needs to be isolated from the rest of the data in the same process) which means that this solution would also require new interfaces that can communicate this information to the kernel or the processor.

12.1.3. Removing the side-channel(s)

The third solution to render speculative execution side-channel attacks useless would be the removal of side-channels that can be used to leak data. It does not matter if kernel memory is readable during speculative execution or if the branch predictor can be abused if there is no way for the attacker to leak any of the obtained information to the outside. SafeSpec[25] is a proposed solution to Meltdown and Spectre that leverages shadow data structures that is used during speculative execution. The basic idea is that all operations executed during speculative execution only modify this shadow data structure and have no influence on the real cache. If a rollback is required, flushing the shadow data structure is enough to ensure that no modifications can be observed afterwards. This approach is very practical as it has little performance overhead and only requires the implementation of the described shadow data structures. However, great care must be taken to ensure that no side-channels remain and that no new ones are introduced. If a completely new side-channel that had not been taken into account when designing the countermeasures were to be found, the same issues we are currently facing would arise again.

12.2. Combined approach

From a practical point of view, the most reasonable solution would be a combination of technique two and three. The removal of all currently known side-channels should be a realistic mid- to long-term endeavor. As mentioned, SafeSpec already provides a reasonable basis for further research in this direction.

Simply preventing access to all out-of-context information alone is most likely infeasible as it is extremely complex to do so without major impacts on performance. However, preventing access where possible can be seen as an important defense-in-depth measure if another side-channel were to be found. It should be made as hard as possible to manipulate the outcome of branch prediction and access to memory that bypasses the rules defined in the page tables should be prevented.

It is the opinion of the authors that the best long-term solution to speculative execution side-channel attacks would be to remove known side-channels by redesigning affected processor parts and to ensure that exploitation of vulnerabilities was to be as hard as possible in case another side-channel should be detected.

13. Conclusion

Speculative execution side-channel attacks are a complex topic of research with many different aspects. While Meltdown was an interesting and dangerous vulnerability in itself, it could be patched in software fairly quickly and relatively easily. As we have shown, Spectre is much more dangerous as it represents a full class of vulnerabilities where additional research is still required to find and mitigate all possible attack vectors. It is very likely that new Spectre-style attacks will continue to be found in the foreseeable future. At the time of writing, it is already complex to keep track of all variants and the associated patches. This situation is most likely going to get even worse as more attacks are found. To keep the rising complexity manageable, it might be necessary to create and maintain an up-to-date database of all vulnerabilities, their impact and available mitigations across platforms. This would however be a difficult task in itself and require collaboration by experts from major vendors.

We have also shown how difficult it is to investigate code running in speculative execution. While some of the patterns we identified in this paper can be used to make the process easier, newly arising issues might require different testing methods. Reverse engineering undocumented hardware behavior is very time consuming but necessary as it is required to better understand the precise impact of the vulnerabilities. In order to facilitate better research, it would be desirable if more details on the inner workings of certain parts of CPUs were to be released. Currently, research such as ours is required get the understanding that can be built upon to uncover and protect against further vulnerabilities.

We also found that in some cases, speculative execution reads can be used as protective measures. For instance, we demonstrated how they can be used to detect otherwise extremely stealthy rootkits. However, we think that malicious use far outweighs possible benefits and call for these vulnerabilities to be mitigated by hardware design changes. We proposed three ways that the vulnerabilities could be fixed and recommended a combination of two of them to ensure defense-in-depth with low performance impact.

Speculative execution side-channel attacks are most likely going to be a part of strong threat models in the future. It is going to take years before all CPUs currently in use are replaced with newer models that include protections in hardware design. More research on both attacks and mitigations is required in order to stay ahead of malicious actors and ensure safety and security even when vulnerable CPU models are in use.

A. Source Code Listings

Listing A.1: Lazy FPU assembly

```
global flush_cache
global time_access
global run_test_rtm

section .text

run_test_rtm:
    xbegin rtm_done
    xabort 1
    call run_test
    xend
rtm_done:
    ret

run_test:
    cvtss2si eax, xmm0
    cmp eax, 0xff80a2a1
    jne skip_access
    mov rax, [rdi]
skip_access:
    ret

flush_cache:
    clflush [rdi]

    cpuid
    mfence
    cpuid

    ret
```

```
time_access:

    cpuid
    mfence
    cpuid

    rdtscp
    mov rbx, rax
    mov rax, [rdi]
    rdtscp
    sub rax, rbx

    ret
```

Listing A.2: Lazy FPU timing logic

```
uint64_t attempt_times[ATTEMPTS];
uint64_t result = 0;
uint64_t secret = 0xcafebabe;

while(1) {
    memset(attempt_times, 0x00, sizeof(attempt_times));

    for(uint64_t bit = 0; bit<64; bit++) {

        uint64_t bit_mask = (uint64_t)1 << bit;

        for(uint64_t attempt=0; attempt<ATTEMPTS; attempt++){

            sched_yield();
            flush_cache(p);
            run_test_rtm(p, bit_mask, &secret);

            attempt_times[attempt] = time_access(p);
        }

        uint64_t min_time = attempt_times[0];
        uint64_t probably_one_count = 0;

        for(uint64_t i=1; i<ATTEMPTS; i++) {
            if(attempt_times[i] < min_time) {
                min_time = attempt_times[i];
            }
            if(attempt_times[i] <= ACCESS_TIME_CACHED) {
                probably_one_count++;
            }
        }
        printf("%d/%d attempts\n", probably_one_count, ATTEMPTS);
    }
}
```

Listing A.3: Longkit detection module

```
// performs the "sidt" instruction and returns the result
_idtr = read_idtr();
memcpy(&_idt1, (const void *)_idtr.base + 16*0x01, sizeof(_idt1));
addr = ((unsigned long)_idt1.offset_high << 32) | (_idt1.offset_middle << 16) | (
    _idt1.offset_low);

printk(KERN_INFO "attempting to leak data from %02lx\n", addr);

for(uint64_t bit = 0; bit<64; bit++) {

  uint64_t bit_mask = (uint64_t)1 << bit;

  for(uint64_t attempt=0; attempt<ATTEMPTS; attempt++){
    flush_cache(p);
    run_test_rtm(p, bit_mask, addr);
    attempt_times[attempt] = time_access(p);
  }

  uint64_t min_time = attempt_times[0];
  uint64_t probably_one_count = 0;

  for(uint64_t i=1; i<ATTEMPTS; i++) {
    if(attempt_times[i] < min_time) {
      min_time = attempt_times[i];
    }
    if(attempt_times[i] <= ACCESS_TIME_CACHED) {
      probably_one_count++;
    }
  }
  printk(KERN_INFO "bit %2lu: time=%-4lu (%lu/%d)\n",
    bit, min_time, probably_one_count, ATTEMPTS);


  [...]
}
```

List of Figures

List of Tables

Bibliography

[1] J. Giovannangeli. (2017, June) Skylake bug: a detective story. https://tech.ahrefs.com/skylake-bug-a-detective-story-ab1ad2beddcd. (last access: 3.7.2018).

[2] X. Leroy. (2017, July) How i found a bug in intel skylake processors. http://gallium.inria.fr/blog/intel-skylake-bug/. (last access: 3.7.2018).

[3] I. Corporation. (2018, March) Intel core x-series processor family specification update. https://www.intel.com/content/www/us/en/products/processors/core/6th-gen-x-series-spec-update.html. (last access: 4.7.2018).

[4] ——. (2017, November) 6th generation intel processor family specification update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/desktop-6th-gen-core-family-spec-update.pdf. (last access: 4.7.2018).

[5] C. Domas, "The memory sinkhole - unleashing an x86 design flaw allowing universal privilege escalation." *BlackHat, Las Vegas, USA*, 2015.

[6] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[7] J. Horn. (2018, February) Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. (last access: 3.7.2018).

[8] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[9] I. Corporation. (2012, April) 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. (last access: 4.7.2018).

[10] R. Hund, C. Willems, and T. Holz, “Practical timing side channel attacks against kernel space aslr,” in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 191–205.

[11] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, “Prefetch side-channel attacks: Bypassing smap and kernel aslr,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 368–379.

[12] Y. Jang, S. Lee, and T. Kim, “Breaking kernel address space layout randomization with intel tsx,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 380–392.

[13] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of aes,” in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[14] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack.” in *USENIX Security Symposium*, vol. 1, 2014, pp. 22–25.

[15] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “Armageddon: Cache attacks on mobile devices.” in *USENIX Security Symposium*, 2016, pp. 549–564.

[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[17] J. Horn. (2018, January) Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. (last access: 3.7.2018).

[18] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, “Kaslr is dead: long live kaslr,” in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 161–176.

[19] I. Corporation, “Intel analysis of speculative execution side channels,” *???*, 2018.

[20] L. Wagner. (2018, January) Speculative execution, variant 4: speculative store bypass. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/. (last access: 3.7.2018).

[21] T. Zheng, H. Zhu, and M. Erez, “Sipt: Speculatively indexed, physically tagged caches,” in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 118–130.

[22] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, “Malware guard extension: Using sgx to conceal cache attacks,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[23] I. Corporation. (2018, May) Intel 64 and ia-32 architectures software developer’s manual. https://software.intel.com/en-us/articles/intel-sdm. (last access: 4.7.2018).

[24] J. Rauchberger, R. Luh, and S. Schrittwieser, “Longkit-a universal framework for bios/uefi rootkits in system management mode.” in *ICISSP*, 2017, pp. 346–353.

[25] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Safespec: Banishing the spectre of a meltdown with leakage-free speculation,” *arXiv preprint arXiv:1806.05179*, 2018.