FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Analysis of the Meltdown attack
Student:	Tomáš Zvara
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2019/20

Instructions

Study the Meltdown attack against modern CPU architectures (https://meltdownattack.com/).
Analyze the properties of the attack and the requirements for its successful execution.
Implement the attack in a Linux environment, perform experiments under different conditions, and measure their success rates.
Determine the properties of the attack that make it Linux-specific. Attempt to adapt the attack to the Windows environment.
Compare the results of the attack in both environments. If you observe significant differences, propose the reasons for them.
Evaluate the effectiveness of known defensive measures against your implementation in both environments.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 15, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Analysis of the Meltdown attack

Tomáš Zvara

Department of Computer Systems
Supervisor: Ing. Josef Kokeš

May 14, 2019

Acknowledgements

I wish to express my sincere thanks to my supervisor, Ing. Josef Kokeš, for his valuable guidance, insightful comments and constructive advice. I would also like to thank my family for their continuous support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 14, 2019 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Tomáš Zvara. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Zvara, Tomáš. *Analysis of the Meltdown attack*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Hlavným cieľom tejto bakalárskej práce je popis a prevedenie útoku Meltdown v operačných systémoch Linux a Windows. Teoretická časť práce začína zoznámením sa s architektúrou moderných procesorov a s opisom cache pamäte. Následne pokračuje opisom virtuálneho adresného priestoru a porovnaním jeho implementácie v oboch operačných systémoch. V praktickej časti je opisaná implementácia útoku pre operačné systémy Linux a Windows. Obe implementácie dokazujú, že špekulatívne vykonávanie inštrukcií umožnuje únik dát s využitím cache pamäte ako postraného kanálu. Posledná kapitola sa venuje analýze protiopatrení, ktoré zabránia takémuto špekulatívnemu unikaniu dát a znemožnia vykonanie útoku Meltdown.

Kľúčové slová útok meltdown, špekulatívne vykonávanie inštrukcií, cache, virtuálna pamäť, realizácia útoku

Abstract

The primary goal of this bachelor's thesis is to describe and execute a Meltdown attack on Linux and Windows operating systems. The theoretical part starts with a background information on modern processor architecture and the cache memory. Then it continues with the description of the virtual address space and a comparison of memory management of both operating systems. The practical part contains an implementation of the Meltdown attack in Linux and Windows. Both implementations prove that speculative execution leaks sensitive information through the microarchitectural side channel. Afterwards, an analysis of Meltdown attack countermeasures is provided.

Keywords meltdown attack, speculative execution, cache, virtual memory, attack realisation

Contents

Introduction **1**

1 Background information **3**
1.1 Out-of-order execution . . . 3
1.2 Cache as a side channel . . . 8

2 Exploring the address space **13**
2.1 Address space . . . 13
2.2 From user to kernel space . . . 15

3 The Meltdown attack execution **19**
3.1 The building blocks of attack . . . 19
3.2 Attack algorithm . . . 22

4 Attack evaluation **25**
4.1 Environment set-up . . . 25
4.2 Cache threshold . . . 26
4.3 Proof of speculative leaking . . . 27
4.4 Reading kernel address space . . . 27
4.5 Reading physical memory . . . 29

5 Countermeasures **33**
5.1 Hardware countermeasures . . . 33
5.2 Side-channel mitigation . . . 34
5.3 Software countermeasures . . . 35

Conclusion **37**

Bibliography **39**

A Acronyms **43**

B Contents of enclosed CD **45**

List of Figures

1.1 CPU instruction processing stages 3
1.2 Scalar pipeline 4
1.3 Parallel unified pipeline 5
1.4 Parallel diversified pipeline 5
1.5 Write-after-write hazard – instruction dependency 6
1.6 Dynamic pipeline 7
1.7 A memory hierarchy pyramid 9
1.8 Multi-level cache system 10

2.1 MMU translation from virtual to physical address 14
2.2 Direct-physical mapping in Linux 16
2.3 Windows vs. Linux x86 & x64 address space layouts 17

3.1 The Meltdown attack algorithm 24

4.1 Linux vs. Windows cache threshold 26
4.2 Linux reading cached vs. uncached memory success rate 28
4.3 The Meltdown on physical machine 31
4.4 The Meltdown on virtual machine 32

Introduction

One of the essential features of modern operating systems is the ability to run multiple programs simultaneously on the same hardware without unnecessary mutual interference. This feature becomes even more important with a rise of cloud computing.

A few decades ago, the programmers came up with a concept of memory isolation which ensures that programs cannot access each other's memory as well as the memory of the operating system. And since then it has been working perfectly. In 2018, though, a group of researchers publicly announced a new kind of computer attacks called Meltdown and Spectre. Right after providing evidence in the form of a paper and a working proof-of-concept, the IT world had been overwhelmed for a few months.

As an IT enthusiast, I wanted to understand what lies behind these attacks and why are they so dangerous. Therefore, I have decided to choose the Meltdown attack as a topic for my bachelor thesis. The main goal is an implementation of the Meltdown attack on both Linux and Windows platforms and a comparison of the platform differences as well as their impact.

In this thesis, I gather the theoretical background that is necessary for understanding the building blocks of the Meltdown attack and discuss requirements for its successful execution. In the practical part, I implement the attack in both Linux and Windows environment. I discuss the differences in both implementations and perform tests to measure the success rates. To conclude the work I study the defensive measures against this type of attacks and discuss the overall effectiveness and performance impact of proposed solutions applied in practice.

Chapter 1

Background information

In this chapter, I provide general information about how does a modern processor work, what is a CPU pipeline and how has it evolved in recent years. Primarily, I focus on the features that enable advanced CPU attacks such as Meltdown. After a short introduction of basic concepts, I explain what we understand under the term "out-of-order execution". In the last part of this chapter, I cover the memory hierarchy, CPU cache memories and how can they be misused as a side-channel.

1.1 Out-of-order execution

The never-ending wheel of IT industry pushes the CPU designers to invent faster and better-performing processors. When it comes to size or number of transistors, it seems we cannot go further [1]. The latest innovations in processor architecture aim for better use of the CPU resources, keeping all parts of the processor as busy as possible.

1.1.1 Instruction stages

Every piece of code, whether it's C or Python, is translated into CPU instructions. The CPU then processes those instructions. The processing of an instruction can be broken down to five separate stages, see Figure 1.1. That corresponds to the CPU's distinct hardware components, each one restricted to execute a single processing step.

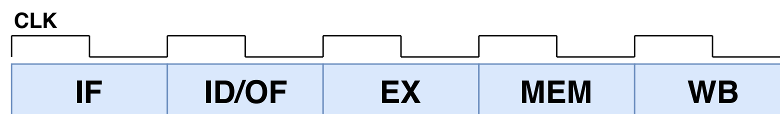

Figure 1.1: Stages of CPU instruction processing: *instruction fetch, instruction decoding, execute, memory access, (register) write back* [2].

1.1.2 Scalar pipeline

The result of each instruction stage provides an input for the next one. There is a well-known name for such data flow, and it is *pipelining*. The CPU pipeline starts at instruction fetch and ends at register write back.

As I mentioned earlier, every instruction stage has its dedicated hardware that is used just when there is an instruction in the appropriate stage. The reason why the CPU processes the instruction in stages is that it enables the CPU to start executing the next instruction before finishing the previous one. Whenever instruction advances to the next stage (e.g. execute), the following instruction, if prepared, can be handled by otherwise idle hardware (e.g. instruction decoding hardware). When each instruction stays in a pipeline stage for precisely one cycle and proceeds to the next one in the next cycle, that is called a *scalar pipelining* [3].

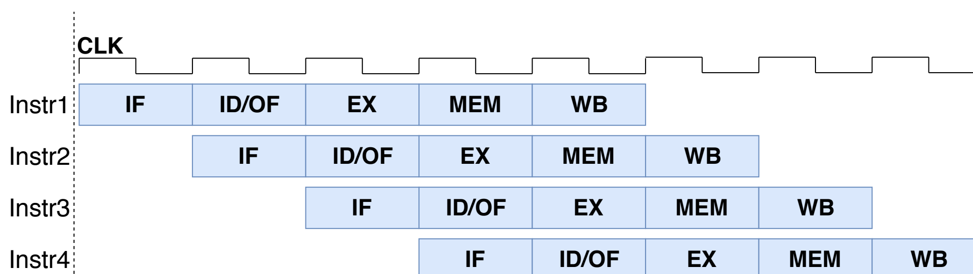

Figure 1.2: Scalar pipeline. CPU fetches *Instr1* in the first clock cycle. The upcoming clock edge moves *Instr1* to the next stage. With the advance of *Instr1* to decode stage, *Instr2* is let into the fetch stage.

Figure 1.2 shows an example of a scalar pipeline and describes what happens in the first two clock cycles. The same principle applies for the next instructions and next clock cycles. The scalar pipeline gives the feeling that the processor executes more than one instruction at a time.

However, there are a few issues that scalar processors face:

1. The maximum throughput for a scalar pipeline is one instruction per clock cycle.

2. Instructions that require long and possibly variable latencies are challenging to unify with simple instructions that need only a single cycle latency.

3. Sometimes instructions might not have all the data necessary for execution. They cannot continue with processing, and that creates a bubble in the pipeline. [3]

1.1.3 Superscalar pipeline

To solve the issues of the scalar CPU, designers came up with a *superscalar processor*. Instead of one single scalar pipeline, superscalar processors have multiple parallel pipelines, see Figure 1.3. Therefore, they are able to initiate the processing of multiple instructions in every clock cycle [3]. Superscalar processors also exceed the maximum throughput boundary for scalar CPU which is one instruction per clock cycle.

Another design technique used in superscalar pipelining is *pipeline diversification*. Instead of implementing more identical pipelines, the execution stage is build up from more heterogeneous functional units. A functional unit is a hardware responsible for instruction execution stage. In Figure 1.4, each pipeline is customised for a distinct instruction type. That eliminates the need for a unified pipeline and leads to more efficient hardware design [3].

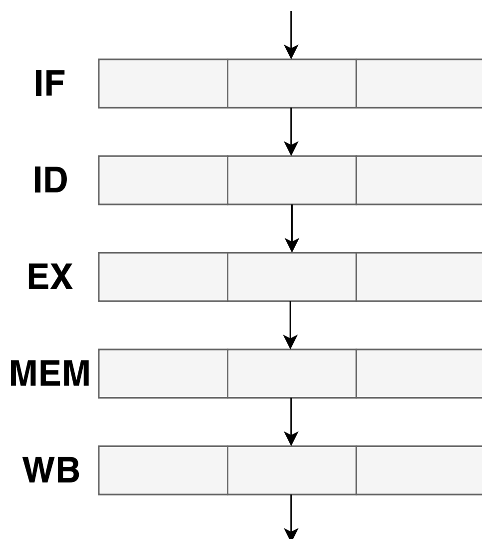

Figure 1.3: The Parallel pipeline is wider than the regular one and can process up to 3 instruction per clock cycle in theory. Of course, this approach requires more sophisticated hardware.

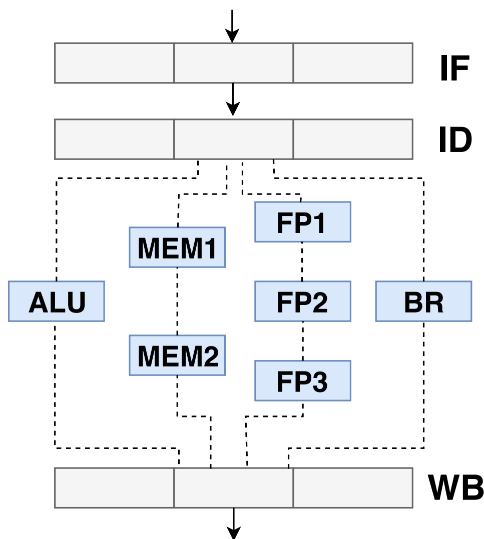

Figure 1.4: Diversified pipeline with four distinct functional units has an advantage of diverse instruction processing, e.g. not every instruction has to pass the memory stage.

1.1.4 Dynamic pipeline

The width of the parallel pipeline determines the number of instructions that can be moved into the next instruction stage in one clock cycle. However, this ideal instruction flow is very often disrupted when used in practice. In the computer programs, there is a high chance (almost certainty) that neighbouring instructions manipulate the same values and their execution depends on the results of the previous instructions.

Hazards, the term used for these program dependencies, arise when an instruction works with a value that previous unfinished instruction might influence. The theory defines three different types of hazards [3]: write-after-read (WAR), write-after-write (WAW), read-after-write (RAW).

Figure 1.5 provides an example of a WAW hazard. The depicted instructions are written in assembly language (ASM). They manipulate with the CPU registers that can be simply explained as program variables. The first instruction LW loads the contents of the main memory from the address specified by register R2 into the register R1. The second instruction ADD adds values stored in register R2 and R3 and writes the result into register R1. More info about registers and main memory can be found in Subsection 1.2.1.

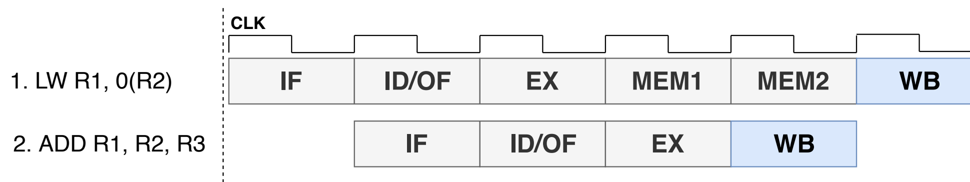

Figure 1.5: Write-after-write hazard. Both instructions write their result into the register R1. However, the writeback stage (WB) of the first instruction executes later, and therefore an unwanted result is produced. Parallel pipelining must preserve the same results as if the instructions were completed in sequence.

When a hazard is detected, the CPU has to stall the pipeline and wait until the previous instruction calculates the proper value. The stalling of instruction creates a bubble in the pipeline and that delays the whole instruction processing.

To reduce the number of stalls, the dynamic planning, i.e. out-of-order execution, is introduced. The out-of-order processor looks ahead across many instructions to issue the independent ones as rapidly as possible. The instructions can be processed in a different order from that written by the programmer, as long as their results are the same as if the instructions were executed in the proper order [2]. Here is a short description of a dynamic pipeline and its instruction stages:

Instruction fetch: Depending on the width of the pipeline, the CPU fetches a certain number of instructions and moves them to the decode buffer.

Instruction decode: The decoder translates the long and complicated instructions into internal low-level instructions called *micro-operations* (μops). Those are decoded and then stored in the dispatch buffer.

Dispatch: The task of instruction dispatching is to route the μops to the appropriate functional unit based on their type. The μops are sorted out to the reservation stations where they wait till their functional unit is ready to process them.

Execute: In the execution stage, functional units take μops from the corresponding reservation station and execute them. The important thing here is that μops do not have to be picked up in the same order as they entered the reservation station. Those μops that are waiting for some data are skipped and processed later. The μops are executed out-of-order. After the execution, the CPU puts them into the reorder buffer. Here, the CPU sorts μops back to the proper order.

Complete & Retire: After finishing the execution, the μops can be completed and update the machine state. Those that update some memory locations are moved to the store buffer where they wait for the CPU to finish the memory writes. Afterwards, the μops are retired.

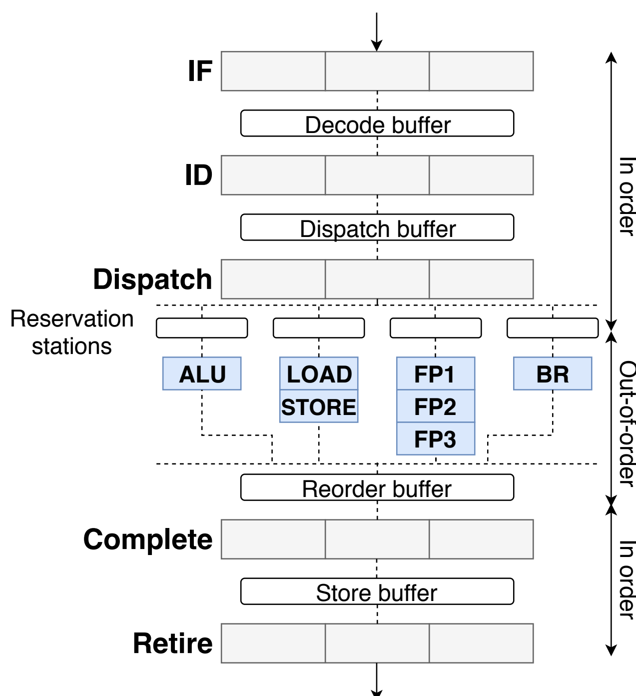

Figure 1.6: Dynamic pipeline [3]

1.1.5 Speculative execution

There is one thing that has to be taken into account when designing superscalar processors and dynamic pipelines, and that is that the control flow of a program does not always have to be linear. The programs contain while and for loops, if statements, functions and exceptions. All of these disrupt the linear instruction flow.

Thus, the modern processors have *branch predictors* which make sure that the CPU always tries to fetch relevant instructions. When the CPU runs into the instruction that is going to change the program flow, the branch prediction units predict how will the instruction order change. If the prediction is correct, the CPU has the proper instructions already in its pipeline and saves some time. If the prediction fails, the CPU pipeline is cleared and re-initialised [4].

In 2018, a group of researchers published Meltdown [4] and Spectre [5] papers where they proved that such speculative behaviour could be misused for exploitation. Afterwards, a few other variants of these attacks appeared [6].

1.2 Cache as a side channel

Since 1980, the speed of instruction processing has been remarkably exceeding the speed of memory access. *"While microprocessor performance has been improving at a rate of 60% per year, the access time to DRAM* [memory] *has been improving at less than 10% per year"* claims Patterson et al. [7]. To cope with this fact, the researchers have designed a memory system that reduces the significant gap between the memory a CPU performance and solves the trade-off between memory size and speed. The system consists of memory components split into several levels where each level differs in its size and speed. The closer to the CPU, the faster and smaller the memory component is and the more recent data it contains. However, the time gaps between individual levels might reveal unwanted information to an adversary.

1.2.1 Memory hierarchy

As I mentioned, the whole memory storage is separated into different levels. Together, all those levels form a *memory hierarchy*. The higher a level in memory hierarchy is, the faster it responds and the less memory capacity it has. A memory hierarchy takes advantage of a *locality of reference*. It is a principle that can be broken down into two dimensions:

Spatial locality is locality in space. If an item in memory is referenced, items whose addresses are close by will tend to be referenced soon too.

Temporal locality is locality in time. If an item in memory is referenced, it will tend to be referenced again soon. [3]

Spatial and temporal locality can be observed in all kinds of computer programs from operating systems up to user-level applications. Those principles apply not only when referencing data but when fetching instructions as well [3].

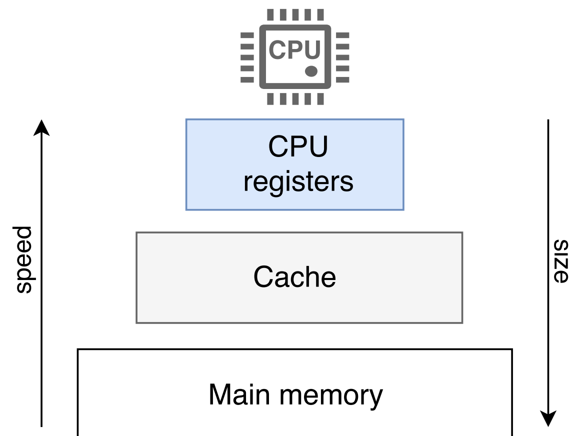

Figure 1.7: A memory hierarchy pyramid. The closer to the CPU the smaller and faster the memory is.

Figure 1.7 depicts a memory hierarchy. Here I provide a short description of each memory component:

CPU registers are the tiniest and fastest components in memory hierarchy. They are located inside the CPU. The number of registers is very limited. Each register typically holds a word of data (32 or 64 bits).

Cache is a small and fast memory that stores frequently used data from the main memory. The cache memory is typically also divided into the levels to improve memory performance even more.

Main memory is the furthest from the CPU in the memory hierarchy. Only the main memory can be directly accessed by the CPU. It contains instructions that are being executed, as well as data that CPU actively operates on. Hardware representation of main memory on modern computers is called RAM.

1.2.2 Cache

The cache is a small and fast memory that is located between the CPU and the main memory. It stores a subset of data from the main memory that were very recently used by the CPU. Due to the locality of reference, there is a high probability that such data will be used again soon. Retrieving the data from the cache saves time and reduces the demands on the main memory [8].

The unit of memory in a cache is a *cache line* with typical size from 64 to 128 bytes. A fixed number of cache lines form a *cache set*. Cache usually consists of more than one cache set. The number of cache lines in a set is called *cache associativity* [8]. Cache maps values from the main memory into the cache set. Because cache set is much smaller than the main memory, some values have to be mapped into the same cache line. If the cache maps a new value to a cache line that is already occupied, cache either puts the value to the different cache set or evicts the older cache line.

Modern processors employ a cache hierarchy consisting of multiple cache levels. Figure 1.8 shows a typical 3-level design. *Last Level Cache*, shortly LLC, is the farthest cache from CPU. In this example, it's the L3 cache.

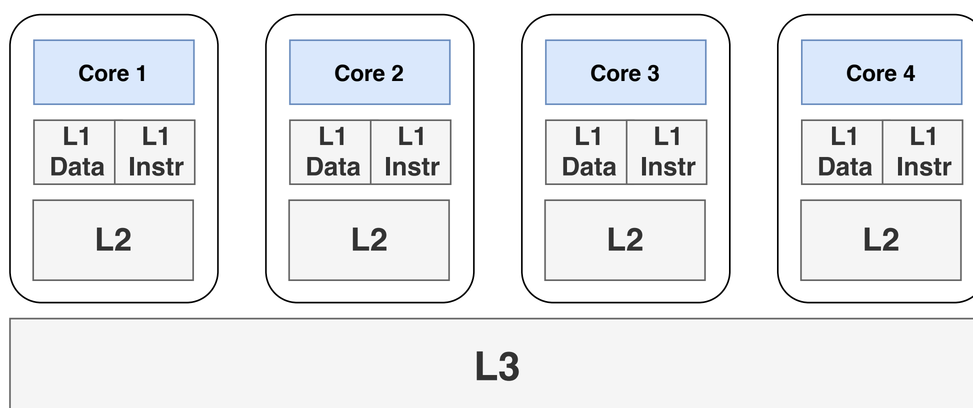

Figure 1.8: Example of a multi-level cache system. The four core CPU has one unified L3 cache shared among all the cores. Each core has one L2 cache and one L1 cache, divided into the data and instruction cache. [8]

When the CPU needs data from the memory, it first checks contents of L1 cache. In the case of a *cache hit*, i.e. the data is found in the cache, the data is retrieved and used. Otherwise, in the case of a *cache miss*, the CPU attempts to get the data from the next cache levels. If all the cache levels report cache miss, the data is read from the main memory. Once a read is completed, the data is typically stored in the cache [5].

1.2.3 Side-channel attack

Side-channel attacks are based on the examination of information that has been unconsciously leaked by a computer system. The information leakage is not caused by a wrong implementation of an algorithm but rather by a system interaction with the environment in a way that has been overlooked by its designers.

For example, consider a user connecting to a server over the internet. The communication is protected with layers of internet protocols that use different

encryption methods. So, even if some adversary is able to eavesdrop the communication, the bytes she is observing are properly encrypted. There is no way she can break the encryption just by examination of all the bytes.

In such a complex system, it is often very easy to miss some possible side-channels, either on the user or server side. The examples of side-channels here might be:

- timing of the messages sent between the user and the server [9],
- server cache usage while encrypting a message with AES [10],
- sound generated by the user is laptop during the RSA decryption [11].

All of this information might be misused by an adversary to break the layers of encryption and read the messages.

The problematic part of side channels attacks is that the channels tend to be quite noisy. Because their primary purpose is not the leakage of information, there is always some interference. When an adversary wants to get at least approximate results, she has to carry out a huge number of measurements. Even in that case, she might recover only a few bits of information. However, a few bits of the encryption key is often enough to break the whole cryptographic system.

A special use case of a side-channel attack is a *covert channel* [4]. While side-channels refer to the accidental leakage of sensitive data by a trusted party, covert channels are exploited by a malware to leak information deliberately. Covert channels exploitation is possible on systems that do not trust their internal components [12].

1.2.4 Cache side-channel

Cache side-channel attacks fall into the category of timing attacks. Cache stores values that have been recently used by the CPU. With a precise counter or timer, it is possible to measure how much time does it take to access the cached memory vs. main memory. This time difference can be used to distinguish between cached and non-cached data. Therefore, an adversary can determine which data the CPU has accessed.

To exploit the timing difference, an attacker sets the cache to a known state before the victim access. Afterwards, she can use two methods to deduce information about the victim's operation.

In the first method, the attacker measures a victim's time of execution. The length of time interval depends on the state of the cache, so the attacker can deduce the cache sets accessed by the victim and, therefore, obtain partial information about the victim's operation. In the second method, the attacker measures how long it takes for her to access data after the victim's operation. This time depends on the cache state prior to the victim's action as well as on the changes the victim's activity caused in the cache state [8].

Based on these two approaches, researches came up with a few variations of cache attacks. Attacks mentioned bellow reveal to the adversary just the address of memory location, not the actual value stored. They assume that the attacker knows or can find out the addresses of relevant data in victim's programs. Ideas how this information is obtained can be found in the provided papers. The most relevant variations are:

PRIME+PROBE [13] starts with the attacker allocating an array with the size equal to cache capacity. Then she reads the values from every memory block of the allocated array (prime). At this moment, the cache is filled with attacker's data. Once the victim has executed her code, the attacker measures the access times to all previously loaded lines to see if any got evicted. If so, the victim has touched an address that maps to the same cache line.

EVICT+TIME [13] uses targeted eviction of cache lines. Firstly, the attacker lets the victim run a code sequence, preloading its working set in the cache and establishing a baseline execution time. Afterwards, the attacker evicts a line of interest from the cache (evict), and lets the victim re-run the same sequence. She measures how long does the victim's code execute (time). A variation from the previous measurement indicates that the line of interest was accessed.

FLUSH+RELOAD [8] relies on the existence of shared memory and the ability to flush the cache by an address. The attacker first flushes a shared line of interest. Once the victim's code has executed, the attacker measures the time of reading the evicted line. A fast reload indicates that the victim touched this line. This attack is used as one of the building blocks of Meltdown and will be mentioned later in Chapter 3. [12]

Chapter **2**

Exploring the address space

In this chapter, I provide a description of memory addressing principles and concepts of virtual memory management. At the end, I discuss how do the Linux and Windows implement these concepts and what are the implementation differences.

2.1 Address space

One of the key responsibilities of an operating system is to efficiently manage and fairly divide hardware resources between simultaneously running programs. Main memory, i.e RAM, is one of those resources as well. To allow multiple processes reside in the main memory concurrently, the operating system has to ensure that each application get its own private part of the memory and that no other process is able to interfere within this area.

A solution for the problem mentioned above is to create an abstraction of the main memory which is called an address space. *Address space* is a set of addresses that a process can use to address memory. Each process has its own address space. The responsibility of OS is to map each independent program address spaces to the main memory.

Even though the main memory size is constantly increasing, software memory requirements keep outrunning this growth [14]. As a consequence, there is a need to have systems that can support multiple programs running simultaneously, where each program will fit in the memory but all of them collectively exceed the memory.

2.1.1 Virtual memory

Virtual memory is a memory management technique devised in 1961 [15]. When implementing virtual memory, each program has its own independent address space that is broken down into small pieces of a contiguous range of addresses. Those pieces are then mapped into the physical memory.

The important feature is that it is not necessary for a running program to have all the pieces in the memory. When the program references an address that is not in physical memory, OS is responsible for finding missing piece, loading it into the memory and executing the failed instruction again. Referencing an address that is already present just triggers a hardware translation mechanism that points the CPU to the proper physical address. This allows the OS to remove unnecessary parts of the program out of physical memory even when the program is still running [14].

To explain the terminology, pieces of address ranges referenced in the context of virtual address space are called *pages*. Otherwise, when someone wants to refer to the pieces of address ranges in the physical memory, they should use the term *frame*. Both pages and frames have the same size, most often 4KiB, but any size that is a power of two can be used [16].

The *Memory Management Unit* (MMU) is a hardware component responsible for converting a virtual address to the physical one. To perform the translation, MMU uses a *Page table* data structure. The structure is an out-tree, i.e all the edges of the tree point away from the root, where parts of the virtual address select the outgoing edge at each level. Therefore, each virtual address uniquely selects a path from the root of the tree to the leaf where the leaf contains the target physical address. Modern OS systems mostly use three or four levels deep page trees for virtual address translation [17].

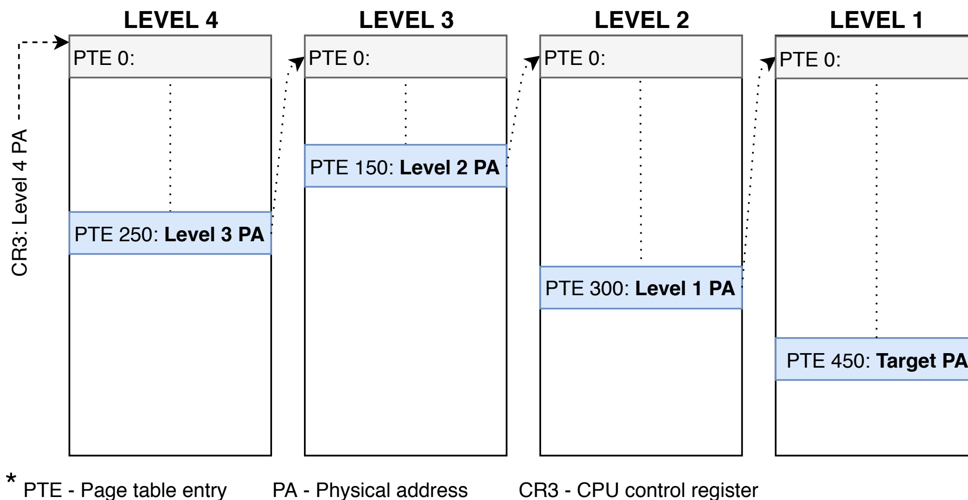

Figure 2.1: Translation of virtual address *0x7D25A59C2000* into the physical address. The MMU uses a four-level page table structure. The page tables are stored in the physical memory as well. [17]

Figure 2.1 shows how the MMU translates the virtual address *0x7D25A59C2000* into the physical one. The CPU has a special control register *CR3* that points to the highest level of the page table (PT) hierarchy. The top 9 bits of the

virtual address index the fourth level PT, selecting page table entry (PTE) 250. This PTE references the third level PT. The next 9 bits of the virtual address are used to find the next PTE, which is 150. Afterwards, succeeding 9 bits select PTE 300 in the second level PT. The remaining bits are used to index the first level PT. Finally, MMU reads the value of PTE 450 and retrieves the targeted physical address [17].

The translation procedure (Fig. 2.1) has to be done on every memory reference and every instruction fetch. In comparison with no memory management, i.e. when the CPU works directly with the physical addresses, it is an unnecessary overhead. There have been many ideas on how to speed up the process of translation. It is worth to mention a *Translation Lookaside Buffer* (TLB) [18] that is a cache for the most recent virtual address look-ups. Also, there have been attempts to avoid multiple page levels with an *inverted page table* that replaces all the page tables with one global page table.

2.2 From user to kernel space

Each process has a private virtual address space with its own page table data structure. The CPU determines the current page table data structure through a value that is stored in the CR3 register. The register value is exchanged upon a context switch, i.e. when the CPU switches the execution to another process. This way the OS is able to manage that any particular process has access only to its address mappings [19].

Process virtual address space is split into a user and a kernel part. The *user address space* can be accessed by the running application. The *kernel address space* can be accessed only if the CPU is running in privileged mode [4]. In the privileged mode, the core of a OS executes and provides its services with complete control over the system. The kernel address space in not addressable from the user mode.

The running application often needs to request a service from the OS, such as file creation or device manipulation. This is done via the system calls. When the application makes a *system call*, it traps into the kernel mode where the OS takes the lead and carries out the necessary commands. If the kernel address space is mapped into the virtual address space of an application, a system call is done without the need to change the page table data structure. That brings a significant performance improvement. Because the user mode pages are still accessible, the OS can also read parameters and access buffers without having to switch back. However, the trade-off is that both address spaces are less private [14].

Lipp et al. [4] mention that the kernel address space *"needs to perform operations on user pages"* that are currently mapped in the physical memory. Therefore, *"the entire physical memory is typically mapped in the kernel"*. How it is done depends on the implementation of the OS.

2.2.1 Direct mapping in Linux

A 32-bit process running in the Linux environment gets 3 GiB of virtual address space for itself, while the remaining 1 GiB is reserved for the kernel. Typically, the kernel resides in low physical memory, but it is mapped into the top of the virtual address space, starting from 0xC0000000 up to 0xFFFFFFFF. Of course, a 64-bit process provides a noticeably larger address space with the possibility to address 16 exabytes. However, processor limitations allow addressing only the lower 48 bits, limiting the possible address space to 256 TiB [20]. Thus, a big part of the 64-bit virtual address space remains unaddressable. The remaining 256 TiB is evenly split between the kernel and user space. Both layouts are displayed in Figure 2.3.

To allow the kernel to manipulate with the physical memory, Linux OS uses a *direct-physical mapping*. It means that the physical memory is directly mapped to a pre-defined virtual address and that the entire physical memory is mapped linearly [4].

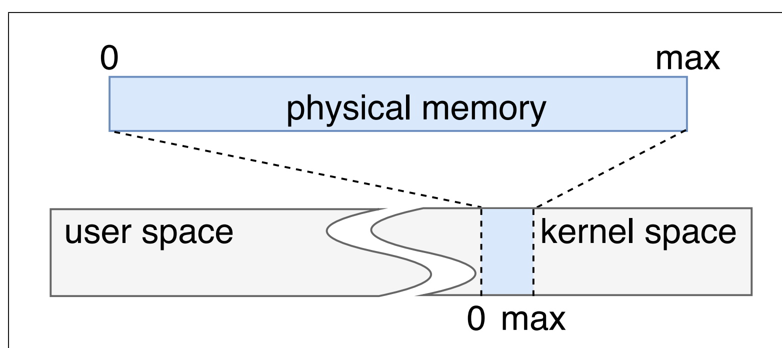

Figure 2.2: Direct-physical mapping in Linux OS. The whole physical memory is mapped into the kernel. Therefore, it is also mapped to every process address space. [4]

2.2.2 Windows pools

A 32-bit Windows process has virtual memory split evenly between kernel and user space. Thus, both user and kernel space get 2 GiB of virtual address space. If the process needs a larger space, Windows is able to reduce the size of kernel address space down to 1 GiB. However, the system must have the option that allows *large space-aware* processes enabled.

The situation in 64-bit Windows address space is practically identical to the Linux 64-bit programs. The 48-bit limitation is caused by a hardware component. Hence, the OS cannot affect that. Figure 2.3 illustrates all the possible mappings in both OS.

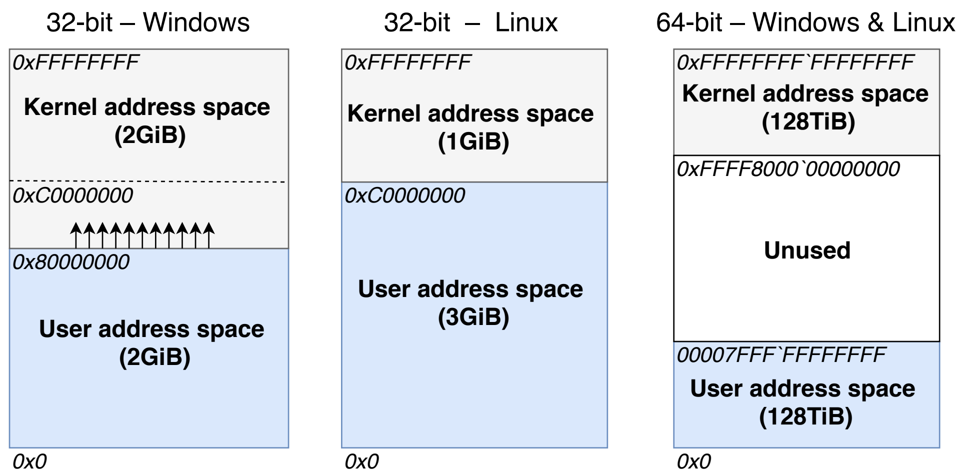

Figure 2.3: Windows and Linux address space layouts. The illustration depicts how much space takes the user and the kernel part. A 32-bit Windows process can expand its user address space up to 3GB. 64-bit address space is similar in both OS.

Windows employ a different technique for mapping the physical memory into the kernel. Instead of a direct-physical map, Windows use working-sets. A *working set* is a set of program virtual pages that are currently in physical memory. Each Windows process has its own working set. The kernel maps working sets of all launched programs into three different parts of its address space:

Non-paged pools are regions of kernel virtual memory that are required to remain in physical memory. The reason is that OS works with these pools on such low levels where a missing page cannot be handled.

Paged pools are parts of kernel virtual memory that can be swapped out of the physical memory in case of high memory usage.

System cache is a part of kernel virtual memory that is used to map currently used files to the cache memory to reduce the number of reads and writes on the disk. [21]

All of these sections map some fraction of the physical memory into the kernel address space of every process [4]. However, the mapping of physical memory is not linear in Windows.

Chapter 3

The Meltdown attack execution

In the prior part of this thesis, I have provided background information that is necessary for understanding of the following sections. To sum it up, I have explained what an out-of-order execution is and how does the cache side-channel attack work in Chapter 1. Later in Chapter 2, I have described the virtual address space of Windows and Linux OS.

In this chapter, I present a Meltdown attack. The Meltdown was publicly announced by Lipp et al. [4] at the beginning of 2018. Together with the Spectre attack [5] that is based on the same principle, Meltdown has been considered to be a very severe attack with a significant impact on end users as well as cloud systems. Based on the information published in the Meltdown paper, I provide an explanation of how does the Meltdown attack work.

3.1 The building blocks of attack

The Meltdown attack allows an adversary to read the contents of kernel address space as well as that of a physical memory without the need of privilege escalation. To read the privileged memory using the Meltdown attack, an adversary has to know how to:

- persuade the CPU to speculate and execute *transient instructions* that read data from forbidden location,
- handle the OS exception for invalid memory access,
- perform FLUSH+RELOAD attack to expose the secret value.

I discuss each of the building blocks in more detail in the next subsections. Then I provide an algorithm of the Meltdown attack.

3.1.1 Transient instructions

The CPU speculation is a feature that keeps the CPU pipeline always busy. The processor tries to guess which instructions should be executed next and runs them ahead before getting the previous results. However, sometimes the instruction order might change, and the CPU has to flush those instructions that are already in its dynamic pipeline.

One of the fundamental requirements for a successful Meltdown attack is the processor's ability to speculate and execute *transient instructions*. This term was introduced in both the Meltdown [4] and Spectre [5] papers. Those instructions that had been advanced to the CPU pipeline, when CPU was speculating, and have to be flushed due to the sudden change of instruction order are called *transient instructions*.

```
1    size_t secret_address = 0xffff820000000000;
2    char* reference_secret = secret_address;
3    char secret_value = *reference_secret; //Segmentation fault
4    //transient instructions, might be executed out-of-order
5    printf("Secret value is : %c\n",secret_value);
6    //...
```

Listing 1: Example of transient instructions in C. Variable *secret_address* contains an address from kernel address space (64-bit). Dereferencing this address without privileged context leads to the Segmentation fault, and therefore a change of the code flow. A block of instructions starting on Line 5 is *transient*. The CPU might speculatively execute this block.

In Listing 1, variable *secret_address* contains an address from the kernel address space range. Line 2 converts the *size_t* to a char pointer, and Line 3 dereferences the value and loads the data of the secret. Since the program does not have proper rights for accessing that virtual address, segmentation fault is invoked. The instructions after Line 3 will never be reached and executed. However, CPU speculation might cause that these instructions are already partially processed somewhere in the CPU pipeline and have to be flushed. Those instruction are called *transient*. They should not change the resulting state of the CPU.

However, as the researchers found out, those instructions do change the microarchitectural state of the system and can leak information through a side-channel. Spectre [5] even proves that transient instructions can be misused in the context of another process. This is very dangerous, especially, when the transient instructions manipulate with a secret value.

3.1.2 Exception handling

To inspect the changes a transient instruction caused in the system it is necessary to ensure that the program will be able to examine the microarchitectural state, even after the expected exception. The second step in the Meltdown attack is an exception handling. Lipp et al. [4] mention three possibilities of successful recovery:

Forking a child process is the most trivial approach. Just before accessing the invalid memory, the process can fork a child process and let the child execute the problematic part. After the exception and probable CPU speculation, the parent can examine the microarchitectural state.

Own signal handler is a bit more sophisticated solution than forking. It reduces the overhead caused by the initialisation of a new process. The program can install a signal handler for the OS exception. When the exception is fired, instead of calling a predefined procedure that immediately stops the executing program, a method provided by the author of the program is triggered. Therefore, the program is able to recover from the exception and examine what had happened.

TSX-NI [22] is an extension to the CPU instruction set architecture (ISA) which provides a hardware transactional memory support. With TSX-NI, a programmer can specify a code region that is executed as an atomic operation. If some error appears during the transactional execution, instructions inside this TSX block are roll-backed, and the state of CPU is reset (not microarchitectural state). After the reset, the program can continue processing without disruption.

Considering these three options, the fastest one is the TSX-NI because the OS is not notified about exception and everything is handled by the actual process. However, only some Intel CPU generations support this extension (Haswell). The computer I used for evaluation did not have this feature. Therefore, I have chosen an exception handling for my Meltdown implementation.

3.1.3 Flush + reload

The third step of the Meltdown attack consists of transforming information from the microarchitectural state into the architectural one. That can be achieved by using one of the methods of cache side-channel attacking. In Chapter 1, I have discussed the three most relevant types of cache side-channel attacks. Here, I am going to dive into the FLUSH+RELOAD variant because it is the one I used in my Meltdown implementation.

Lipp et al. [4] say that the most optimal method for examination of the microarchitectural state is the FLUSH+RELOAD attack. They claim that

it is the most accurate known cache side-channel technique and is relatively simple to implement. It identifies access to specific memory lines, whereas the prior attacks required the manipulation with cache sets. Consequently, FLUSH+RELOAD does not suffer so much from false positives and does not require additional processing [8].

In Meltdown, the situation with FLUSH+RELOAD is a bit different from the standard usage of this attack [8]. Initially, FLUSH+RELOAD was designed for a situation where a victim changes the state of the cache and an adversary tries to recognise that. However, here is FLUSH+RELOAD used as a covert channel with the adversary executing her code on both ends. A sequence of transient instructions is the sending end of the covert channel. The receiving end of the covert channel is the state of the cache memory. An adversary ensures that CPU addresses a secret value. Afterwards, she examines the cache state and reveals the secret.

The essential requirement of the FLUSH+RELOAD is the ability to flush specific memory lines from the cache. Modern CPUs provide a *clflush* instruction that evicts the specific memory line from all cache hierarchy levels [8]. However, some languages do not provide access to this instruction (Javascript) and an adversary has to use a different technique to create a covert channel.

When using this technique, it is up to the adversary to decide the width of FLUSH+RELOAD covert channel. Even though the smallest memory unit that CPU can work with is 1 byte, with a proper use of byte shifting the width of the channel can be just 1 bit. There is also an upper limit given by the size of cache memory.

Of course, transferring more bits (or bytes) at once has some cost. With every single bit, the time and memory demand on cache increase exponentially. Actually, FLUSH+REALOAD is the time bottleneck of Meltdown attack. Thus, it is reasonable to keep the width of the channel to the minimum. However, as Lipp et al. [4] claim, the 1-bit variant also has some downsides. In a 1-bit wide covert channel, the frequency of '0' is around 50 %. The '0' value has a higher false positive rate in Meltdown attack because it might also indicate a failure. Therefore, a 1-bit covert channel has a higher error rate.

Since I am not concerned so much with the speed of the attack in my work and focus more on the comparison of accuracy and feasibility, I have decided to use an 8-bit approach.

3.2 Attack algorithm

Here, I enumerate steps of the algorithm for meltdown attack. To better understand how an adversary reveals a secret value, I have also created an algorithm illustration, see Figure 3.1.

Pre-attack preparation:

- *Find the cache threshold.* Perform a significant number of measurements to get a time interval when accessing cached memory location vs. accessing flushed memory. Based on the time difference set a reasonable value – cache threshold – to distinguish between cache and memory access.
- *Allocate a char array with a size of* $4096 * 256\ B$. The number 256 represents all the possible 8-bit message variations, and 4096 (4KiB) is the size of one memory page. The spreading of one message variant per page eliminates the risk that cache would prefetch the following variation [23]. Also, fill the array with some non-zero values, because OS could optimize the memory usage and the attack would not work.
- *Select the virtual address.* Find an address that is located in the kernel address space of your running program.

Attack:

1. *Flush a char array from the cache memory.* It is enough to flush just the beginning of each page.
2. *Access the desired virtual address.* Read one byte from the address location, shift the data 12 bits left to match the array mapping and use the result to index the array.
3. *Handle the OS signal interrupt.* Use one of the methods mentioned in Subsection 3.1.2.
4. *Measure each array element access time.* For an accurate measurement it is necessary to have a precise timer such as RDTSCP.
5. *Find out which element was in cache memory.* It is the element that has an access time under the cache threshold. In ideal case, there is only one.

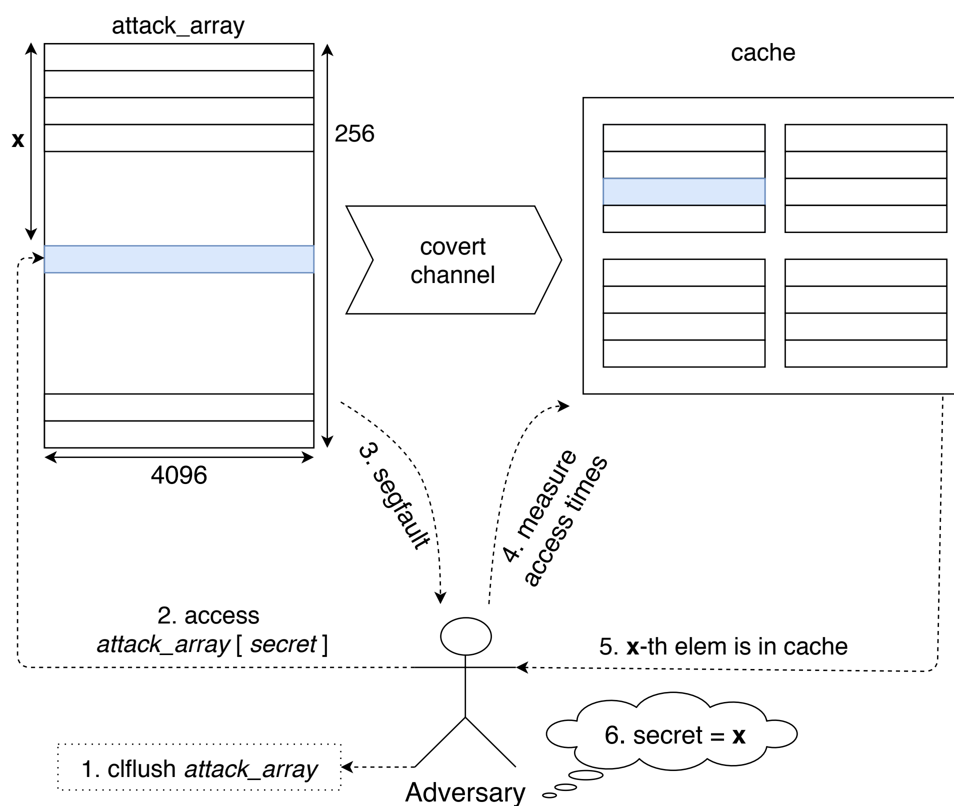

Figure 3.1: The Meltdown attack algorithm. The adversary reveals a secret value using the cache covert channel. The steps of the attack are explained above.

Chapter **4**

Attack evaluation

After the theoretical background of the Meltdown vulnerability, I provide a proof of concept implementation for both Linux and Windows OS. In this chapter, I describe the testing environment and present my implementation together with the test results. The source code and test measurements can be found on the university GitLab page `https://gitlab.fit.cvut.cz/users/zvaratom/projects` and the enclosed CD.

4.1 Environment set-up

To evaluate a Meltdown attack and perform the tests I have set up a dual-boot on HP Probook 6570b with 3rd generation CPU *Intel Core i5-3340M 2.7 GHz*. To test the exploit in the virtual environment, I worked on Acer Aspire E5-571G with 5th generation CPU *Intel Core i7-5500U 2.4 GHz* using VirtualBox [24] as a virtualisation tool.

From Linux distributions, I chose a 64-bit Ubuntu 16.04 LTS, kernel version 4.13.16-041316-generic. I compiled the binaries with a standard Linux C compiler – gcc version 5.4.0. As a Windows environment, I used a 64-bit Windows 7 Professional with Service Pack 1, build version 6.1.7601. I compiled the binaries with a Microsoft Visual C/C++ compiler MSVC – cl version 19.16.27027.1 for x64. In both cases, I intentionally use outdated software in order to avoid kernel Meltdown mitigation patches. Although the versions I used for evaluation have a kernel address space randomisation (KASLR), I turn off this feature for my testing experiments. KASLR is more thoroughly discussed in Chapter 5.

I used a test stressing tools to measure how my implementations behave under a varying load. In Linux, I used the command line *stress* utility. In Windows, I used the *HeavyLoad* [25] stress tool.

Both implementations are written in C, except for the critical transient instruction part which is written in assembler to avoid compiler optimizations.

4.2 Cache threshold

To perform a FLUSH+RELOAD attack, I need to distinguish between the cached and the uncached memory accesses. I do that by setting a cache threshold. A value of the cache threshold depends on the system hardware. Therefore, it is a good idea to perform the measurement every time we attack.

Figure 4.1 shows that both Windows and Linux OS have an almost identical threshold value on my physical machine. The graph displays cached and uncached memory accesses and the calculated threshold value for each measurement (50 in total).

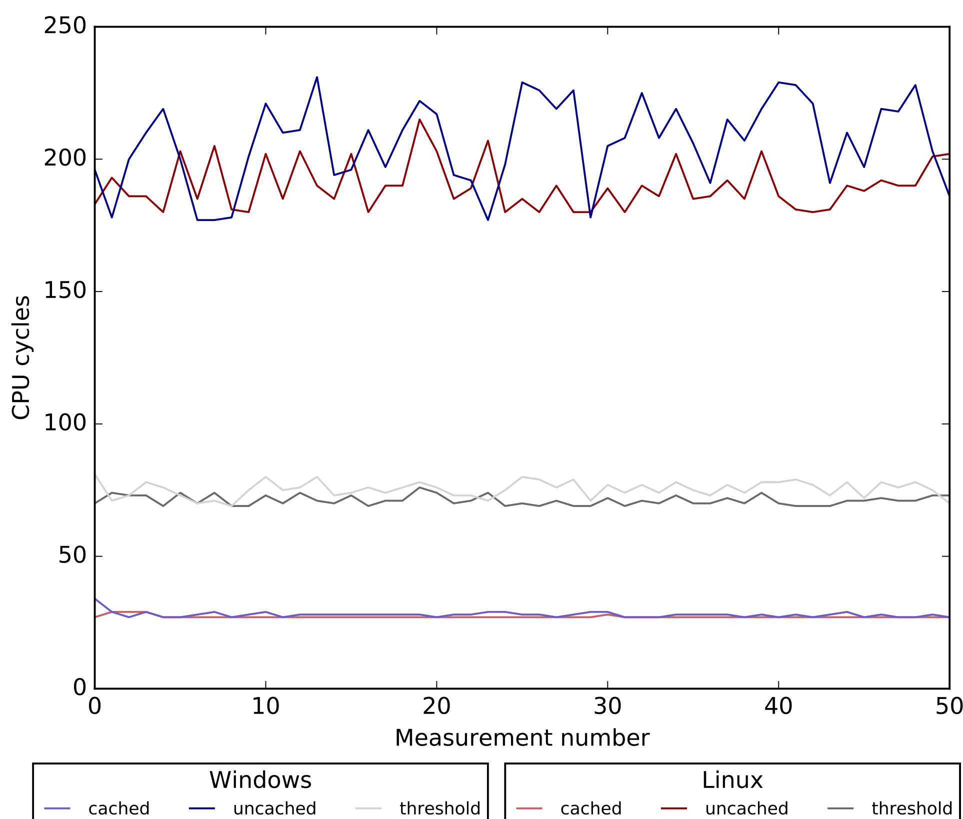

Figure 4.1: Cached vs. uncached memory accesses on Windows and Linux. The cache threshold is around 70-80 CPU cycles for both OSes on my physical machine.

To get a sufficient estimation of the cache threshold in one measurement, the measurement should consist of 100 000 cached and uncached accesses at least. The uncached results have occasional peaks that raise the cache threshold value. If I decrease the memory accesses down to 1000 per measurement the cache threshold jumps to 200 CPU cycles.

4.3 Proof of speculative leaking

The Meltdown vulnerability is based on the fact that the speculative execution leaks information through the side-channel into the microarchitectural state. This behaviour can be abused to bypass the address space protection.

To force the CPU to execute particular instructions out-of-order and in a certain way is challenging. The state of the CPU is influenced by a lot of factors on a live system. Therefore, the first example I worked on just proves that I am able to force the CPU to speculate and read some memory content only by accessing it inside the transient instruction sequence, even if I have proper rights to read it. Here I do not bypass any address space protection.

```
1 ;RCX -> memory address, RDX - attack_array for FLUSH + RELOAD
2 XOR R8,R8
3 MOV R9,[R8]                   ;throws exception
4 MOVZX EAX, byte ptr [RCX]     ;start of transient sequence
5 SHL EAX, 12
6 MOV RBX, [RDX + RAX]
```

Listing 2: The transient instruction sequence from file src/speculative_poc.c. Line 3 throws an exception. Line 4 is processed thanks to the out-of-order execution.

Listing 2 shows the core instructions of the speculation code. On Line 2, the program reads from address 0x0 and that leads to a system exception. Even though the instructions after Line 2 should never execute, the src/speculative_poc.c is able to read the content of the desired memory location using the FLUSH+RELOAD covert channel.

I tested this example on the physical machine running both in Linux and Windows environment. I used the programs to read 1 KiB of memory and performed the tests with low as well as high CPU usage. Both systems succeeded in between 90 and 100 % of cases in all 10 tests. The performed measurements can be found on the enclosed CD in *results/speculative_poc* directory.

4.4 Reading kernel address space

In the previous section, I showed that it is possible to extract the content of some memory location using just the CPU speculative execution and the microarchitectural side-channel. The next step is to misuse this behaviour to bypass hardware protection methods and read some data from kernel virtual address space without the CPU running in the privileged mode.

The kernel address that I am going to use for a tests has to meet two requirements. The first requirement is that I have to know what values does

the particular kernel address hold to validate the Meltdown information leak and to measure its success rate. The second requirement for the kernel address is that the memory location I access needs to be in the L1 cache, because the transient instruction sequence is then executed faster and, thus, gives better results. It is important to mention that the researchers state [4] it is not necessary for the Meltdown attack to have the contents of the memory cached. Although, leaking of the uncached memory requires tweaking of the attack parameters for particular hardware setup.

4.4.1 Linux

My Linux kernel leaking implementation has been inspired by Boldin [26] proof of concept. In his work, he used one of the global kernel symbols that are listed in the file */proc/kallsyms*. He picked up the linux_proc_banner. The reason is that this kernel symbol is cached on every read from */proc/version* file. To contents of the linux_proc_banner can be found in the kernel source code [27]. Therefore, this address meets the requirements stated above.

To prove that leak of the banner is possible, I wrote a script linux_banner.sh. It requires superuser privileges to find out the address of banner in */proc/kallsyms* [28]. After fetching the address, it starts a C binary that reads the content from the fetched memory address.

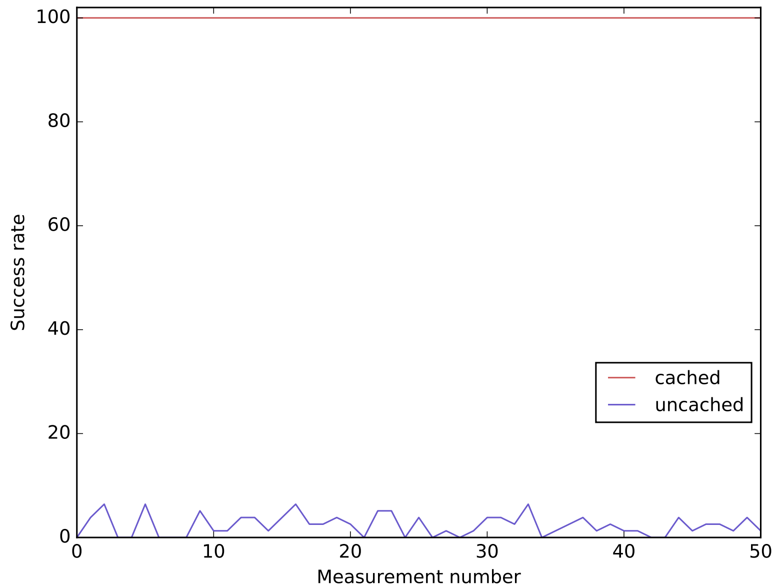

Figure 4.2: The success rate of the cached vs. the uncached memory leak using the Meltdown vulnerability. Having the leaked memory in the L1 cache significantly increases the ability of the Meltdown to read the memory content.

Figure 4.2 shows the difference between leaking of a cached and uncached linux_proc_banner value from the kernel address space. If the leaked information is in the L1 cache memory, the Meltdown success rate increases significantly. However, the memory leak of the uncached memory is still possible.

4.4.2 Windows

In Windows OS, I have started with the idea proposed by the StormCTF group [29]. To exploit some kernel virtual address, they used the *NtQuerySystemInformation* function of the *ntdll.dll* library and got the virtual address of *ntoskrnl.exe* binary located in the kernel virtual address space. To verify the leaked data, they imported the *ntoskrnl.exe* into the user address space of the running program using Windows *LoadLibrary* function.

However, my implementation was not able to leak the *ntoskrnl.exe* binary content from the kernel address space. The FLUSH+RELOAD attack always identified the first array page as the cached one, indicating that the leaked byte was equal to '0'. Lipp et al. mention in their work [4] that this might be caused by the CPU zeroing out the register in case of exception triggering. If the zeroing of the register is faster than the execution of transient instruction sequence the secret value is wiped out of the register. This is also the reason why caching of the sensitive data increases the success rate of the attack.

After the failure, I have tried a different approach. I used a Windows kernel debugger to print the contents of interesting memory locations [30] and tried to execute the Meltdown to read their content. I aimed at the PTE Space location where the kernel stores the 4-level page table mappings for user mode and kernel mode. I expected that such values addresses are accessed by the OS constantly. Thus, I expected that they will be in the cache memory.

I focused on the memory regions that were containing non-zero values around *FFFFF6FB‘7DA00000* and *FFFFF6FB‘7DBED000* addresses. The Meltdown was able to read a few bytes from these locations, but with a low success rate of under 10 %.

The reason for such a low success rate in Windows might be the L1 cache. I am not sure whether the accessed values are cached at that moment. Fetching of an uncached value mi ght cause that the zeroing of the register is faster than memory fetch and no data are leak into the covert channel as a result.

4.5 Reading physical memory

It is true for both Windows and Linux OS that a major part of the physical memory is mapped into the kernel address space. As a consequence, an address space of each program contains memory pages of other running programs as well. Therefore, the Meltdown can be used not only to read internal OS data

structures but also to leak sensitive data of other applications that have their memory pages currently in the physical memory.

4.5.1 Linux

To prove that it is possible to steal the data of another running application, I have created two programs. The first program *src/secret.c* declares a local variable that points to a string on its local program stack. When the program is running, the memory page with the program stack gets into the physical memory. The program prints the string and its physical address to the terminal. To translate the virtual address of the string into the physical one, the program *src/secret.c* requires administrative privileges.

The translation is done by reading the */proc/self/pagemap* file. The file is a symbolic link that points to the */proc/[pid]/pagemap* where *[pid]* is replaced by the process ID of currently running program. As the Linux documentation states [31]: "This file lets a userspace process find out which physical frame each virtual page is mapped to". I index the file with the virtual page number to get a corresponding physical page frame.

The second program *src/local_var_poc.c* takes one parameter which is the physical memory address. When the program starts, it translates this address into the virtual address in its own address space. The translation is done just by adding the physical address kernel direct-mapping offset that is 0xffff 8000 0000 0000. Afterwards, the virtual address is speculatively accessed and the program is able to leak the data from the other running process.

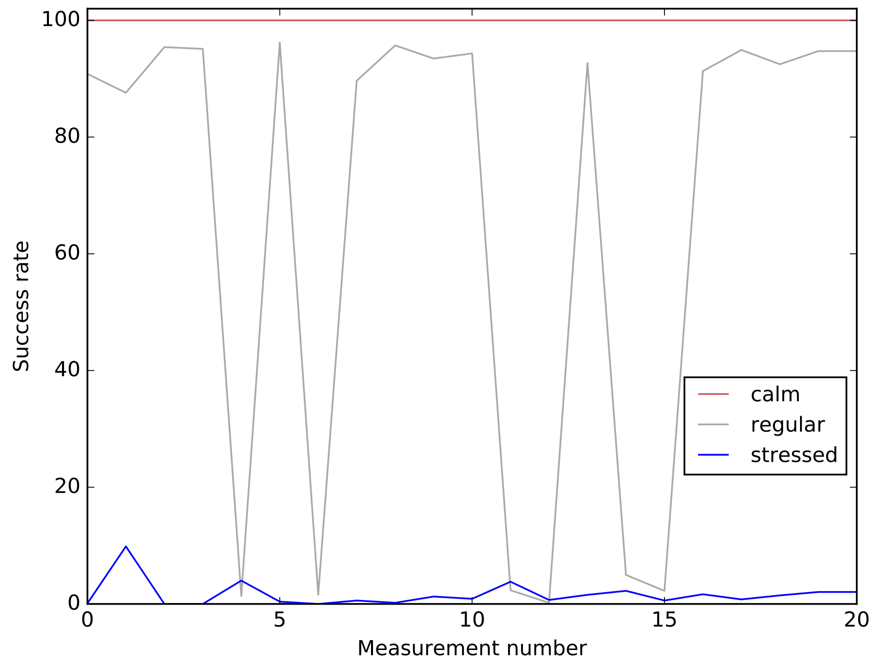

Figure 4.3: The success rate of the Meltdown attack stealing 1 KiB of information from another process. The tests were executed on the physical machine that was exposed to the various system conditions while performing them.

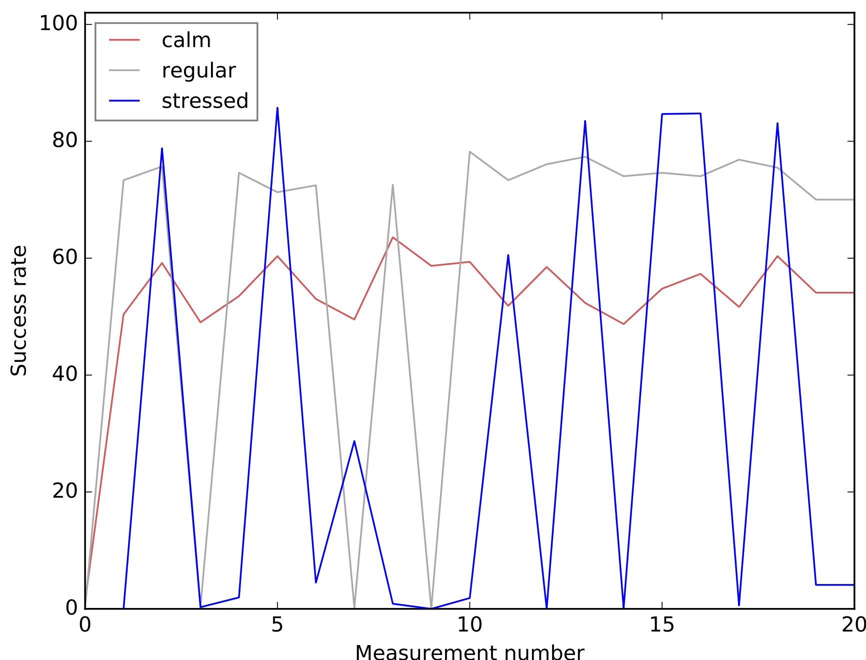

Figure 4.4: The success rate of the Meltdown attack leaking 1 KiB of information of another running process. These tests were performed in the virtual environment.

Figures 4.3 and 4.4 display the results of the tests where I compared the Meltdown success rate in the physical and the virtual environment. I exposed both systems to different conditions, measuring the success rate when the system was in calm (running just the exploit), regular (watching a video on Youtube) and stressed (*stress* command-line utility) state. The result shows that the reliability of the Meltdown exploit depends a lot on the system state and its current resources and that it can be unpredictable.

4.5.2 Windows

Windows mapping of the physical memory into the kernel address space is not so straightforward as in Linux. The Linux example works because I can do the translation between the virtual and the physical address back and forth.

The translation from the virtual to the physical address in Windows can be done by examination of page table data structures and particular page table entries. However, I have not found a way of translating the physical memory into the virtual one. Windows uses working sets for mapping the physical memory into the kernel space. The mapping is not continuous and is spread across three different kernel memory regions. Therefore, I was not able to create a Windows proof of concept.

Chapter 5

Countermeasures

In this chapter, I propose countermeasures against the Meltdown attack presented in Chapter 3. The Meltdown is built upon these three aspects: CPU speculative execution, misuse of a cache as a side-channel, and the mapping of kernel and user space into the single address space. In the following sections, I present known mitigation techniques that can be used to protect against each aspect. I also discuss their effectiveness against my implementation that was demonstrated in Chapter 4.

5.1 Hardware countermeasures

The most straightforward and trivial solution to the Meltdown would be to disable the CPU out-of-order execution completely. However, this is not a feasible option for two reasons. The first is that it is not possible to do that without changing the microarchitecture of the current CPUs. The second reason is that the speculative execution is very critical for the performance of the modern systems. The performance impact of disabling the out-of-order execution would be devastating [4]. Although, Kocher et al. propose a possibility that future CPUs could determine if the data was fetched as a result of CPU speculation [5]. If so, the data might be prevented from being used in subsequent operations that would leak them.

The Meltdown can be seen as a form of race condition between the memory fetch and the corresponding permission check. Lipp et al. suggest [4] the serialisation of both operations to prevent an unauthorised speculative read of the memory content. However, that would cause the stalling of the CPU pipeline on every memory fetch, bringing a significant overhead to every memory operation.

A more realistic solution proposed by Lipp et al. [4] is a hardware support for splitting the user space and kernel space by adding a new hardsplit bit into the CPU control registers. The hardsplit bit set to '1' would signalize

that the kernel addresses have to reside in the upper half whereas the user addresses have to reside in the lower half of the virtual address space. That way, every memory fetch could be immediately identified by comparing the fetch destination with the security boundary in the middle of the address space. The privilege access level of the virtual address could be identified purely from the virtual address, avoiding any further lookups.

As all of these countermeasures fix the problem on the hardware layer, the mitigation would immediately propagate to all the major operating systems. Any of these hardware countermeasures would stop the Meltdown from reading the sensitive information, including both of my implementations. Although, not all of them can be practically used because of their significant impact on CPU performance.

It is worth noting that not all of the modern CPU vendors have been vulnerable to the Meltdown attack. As AMD states, [32] *"No AMD processor has been designed with this behaviour* [Meltdown]*"*. Lipp et al. [4] also claim that ARM CPUs were not affected either. On the other hand, most of the Intel CPUs were affected by the Meltdown attack. After the public disclosure, Intel presented a new hardware protection for 9th generation CPUs as well as micro-code updates for the older generations [33].

5.2 Side-channel mitigation

To protect against the Meltdown information leak into the microarchitectural state through the side-channel, general countermeasures against the cache side-channel attacks can be applied [12].

The essential requirement for a successful cache side-channel attack is a precise timer. The attacker has to distinguish between the fetch from the cache and the fetch from the main memory. If the timer available to the attacker does not have proper granularity or the timer error is big enough, the inaccuracy of the measurement can stop the adversary from leaking the sensitive data. The timer does not have to be an actual time source. A simple sequence of continuous events is enough, e.g. network packets, CPU instruction retiring, etc. Therefore, this approach is very restrictive in practice and very difficult to incorporate in a real-world use [12].

Next, specifically for the FLUSH+RELOAD attack, the adversary needs to execute the *clflush* instruction. One of the possible options would be to make *clflush* accessible only in the privileged context. However, this would not help mitigate the Meltdown as other cache attacks do not require the use of this instruction and make use of a cache eviction instead.

Neither Linux or Windows incorporate any of these solutions. Both OSes provide unprivileged access to the high-resolution timer RDTSCP without inserting any noise. Also, the *clflush* instruction can be accessed without specific privileges as well. Both my implementations, however, are using the

RDTSCP counter to measure the time of memory access and *clflush* instruction for FLUSH+RELOAD attack. Therefore, the proposed countermeasures would be effective against it.

5.3 Software countermeasures

The Meltdown is considered to be a hardware vulnerability because it is based on bypassing the hardware-enforced isolation. That is the reason why all the operating systems were affected by the Meltdown attack after its discovery. The hardware issues should be fixed on the hardware layer. At that time, nobody was sure if the fix would require entirely new hardware or if the microcode updates would be enough [4]. Therefore, the OS developers had to provide some hot-fix to keep their users environment safe until the propagation of hardware changes.

5.3.1 Kernel address space load randomization

Address space load randomization (ASLR) is a computer security technique that on every load of a program randomizes its virtual address space in a non-deterministic way. The randomization can happen on a coarse-grained level or a fine-grained level. The coarse-grained ASLR randomizes only the base addresses of different memory regions, e.g. data, heap, stack. On the other hand, a fine-grained ASLR uses randomization even for changing the order of functions, variables and constants in the memory. Because the latter variant leads to significant performance penalties, the coarse-grained approach is used in most of the systems [19]. The ASLR serves as a countermeasure against control flow hijacking attacks [34]. In these attacks, the attacker needs to find out the memory layout of the targeted process to find the parts of the code that can be misused for the execution.

Nowadays, all the major operating systems have adopted ASLR and use it also for the randomization of the kernel address space. *Kernel address space load randomization* (KASLR) randomizes the kernel address space on every boot of the OS. In Linux, ASLR is present since kernel version 2.6.12, released in 2005. KASLR was merged into the Linux kernel mainline on 2014 in kernel version 3.14. Windows adopted the ASLR for both user and kernel address space in 2007 for systems Windows Vista and higher [35].

For the sake of simplicity, I do not bypass the KASLR in my Meltdown implementations. In Linux, I disable the KASLR option on every boot. That way, the direct-physical mapping of the physical memory in the kernel address space always starts at the same location. In Windows, I use the Windows kernel debugger to examine the contents of the kernel virtual address space. Then I compare it with the results leaked by the Meltdown.

KASLR is not a sufficient mitigation against the Meltdown. It just confuses the adversary and makes the attack more complicated. Lipp et al. [4] proved that the KASLR could be bypassed using a brute force approach.

5.3.2 KAISER

In 2017, Gruss et al. [36] stated that KASLR is no longer a sufficient protection of the kernel address space and came up with a new method of kernel isolation under the name of KAISER. The idea of KAISER is to provide a stronger kernel space isolation by unmapping the kernel from the user space to a separate address space that is accessible only from the privileged mode.

When KAISER is adopted, each process has two address spaces. The first contains only the user space and a few kernel pages that have to be accessible from the user mode. The second contains the whole kernel address space and is accessible only when the CPU is running in the privileged mode. In comparison with the total separation of user and kernel space, KAISER implementation provides a more optimized solution and lowers the impact on CPU performance to the minimum [36].

After the Meltdown discovery, the KAISER became the only viable solution for the vulnerability and had been promptly added into the kernel of all major OSes. Linux developers further optimized the KAISER implementation and then renamed it to kernel page-table isolation (KPTI) [37]. Afterwards, it was merged into the Linux mainline kernel version 4.15. For Windows, Microsoft implemented a similar protection technique called KVA Shadow [38] that was propagated in the beginning of 2018 into the all supported Windows versions.

Because KAISER unmaps most of the kernel address space away from the reach of the user mode, it protects against the Meltdown data leak. After installing the patches, I can state that I was not able to execute the attack against both Linux and Windows kernel. However, as Lipp et al. [4] claims, a few kernel pages that stay mapped in the user address space can still be read by the Meltdown. If these pages contain pointers into the kernel address space, it can help the adversary to break the KASLR and derandomize the kernel address space.

Another issue with KAISER patches was their performance impact. It varies based on the number of system calls done by the running program. Therefore, it was difficult to predict the KAISER impact in the real world scenario. The benchmarks estimated the performance slowdown between 0 % and 800 % [39]. The upper boundary was reached at the frequency of 10 million system calls per second. A real cloud platform database service makes up to 50 000 system calls per second at most [40], though. In this case, the slowdown was around 2.6 % [39] which is a reasonable trade-off of speed and the provided security.

Conclusion

The goal of the practical part of this thesis was to demonstrate a Meltdown attack in both Linux and Windows environments and analyse the differences.

On both operating systems, I was able to prove that speculative execution leaks secret information into the microarchitectural state. This comes as no surprise because this vulnerability is based purely on hardware and the CPU architecture.

On Linux, I demonstrated the reading of kernel address space is possible. As a consequence, the whole physical memory can be read as well. The proof of this statement is my implementation where an adversary program reads the contents of other program's local variables.

On the Windows operating system, I also proved that reading of the kernel address space is possible. However, the proposed implementation has not performed as well in the tests as the Linux variant. I have concluded that this might be caused by the fact that the exploited values were not stored in the L1 cache. Also, I did not manage to read a physical memory on the Windows platform because of its complex non-linear mapping.

In the last chapter, I proposed several countermeasures that could be incorporated on more levels. The software countermeasures that patched the vulnerability caused by the CPU fixed the issue but also brought some performance downgrade that might be noticeable in certain situation. The most reasonable solution is to fix the hardware. The Intel company has already announced a new generation of the CPU that are already protected against the speculative attacks. They will be released in October 2019.

The propagation of new hardware might finally bring the Meltdown and Spectre vulnerabilities to their end. However, until that happens, these attacks will present dangerous threats that force the system administrators to choose between the maximum performance and security of their systems.

Bibliography

1. *After Moore's law* [online]. The Economist Newspaper, 2016 [visited on 2019-04-21]. Available from: `https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law`.
2. HARRIS, David Money; HARRIS, Sarah L. *Digital Design and Computer Architecture*. 2nd ed. Waltham: Elsevier, 2013. ISBN 978-0-12-394424-5.
3. SHEN, John Paul; LIPASTI, Mikko H. *Modern processor design, Fundamentals of Superscalar Processors*. Long Grove: Waveland Press, 2013. ISBN 978-1-4786-0783-0.
4. LIPP, Moritz et al. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium, 2018.
5. KOCHER, Paul et al. Spectre Attacks: Exploiting Speculative Execution. In: S&P, 2019.
6. PAVLÍK, Vojtěch. It's not over with Meltdown; L1TF, POP SS, TL-Bleed and more. In: Linux Days, 2018. Available also from: `https://www.linuxdays.cz/2018/video/Vojtech_Pavlik-Meltdownem_to_neskoncilo.pdf`.
7. PATTERSON, David; ANDERSON, Thomas; CARDWELL, Neal; FROMM, Richard; KEETON, Kimberly; KOZYRAKIS, Christoforos; THOMAS, Randi; YELICK, Katherine. A Case for Intelligent RAM: IRAM. *IEEE Micro*. 1997, vol. 17, pp. 34–44. ISSN 1937-4143. Available from DOI: `10.1109/40.592312`.
8. YAROM, Yuval; FALKNER, Katrina. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium, 2014. Available also from: `https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf`.

9. BRUMLEY, David; BONEH, Dan. Remote Timing Attacks are Practical. In: USENIX Security Symposium, 2003. Available also from: `https://crypto.stanford.edu/~dabo/pubs/papers/ssl-timing.pdf`.
10. IRAZOQUI, Gorka; INCI, Mehmet Sinan; EISENBARTH, Thomas; SUNAR, Berk. Wait a Minute! A fast, Cross-VM Attack on AES. In: *Research in Attacks, Intrusions and Defenses*. Gothenburg: Springer International Publishing, 2014, pp. 299–319. ISBN 978-3-319-11379-1. Available from DOI: `10.1007/978-3-319-11379-1_15`.
11. GENKIN, Daniel; SHAMIR, Adi; TROMER, Eran. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In: *Advances in Cryptology – CRYPTO 2014*. Santa Barbara: Springer Berlin Heidelberg, 2014, pp. 444–461. ISBN 978-3-662-44371-2. Available from DOI: `10.1007/978-3-662-44371-2_25`.
12. GE, Qian; YAROM, Yuval; COCK, David; HEISER, Gernot. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*. 2016, vol. 8. ISSN 2190-8508. Available from DOI: `10.1007/s13389-016-0141-6`.
13. OSVIK, Dag Arne; SHAMIR, Adi; TROMER, Eran. Cache Attacks and Countermeasures: The Case of AES. In: *Topics in Cryptology – CT-RSA 2006*. San Jose: Springer Berlin Heidelberg, 2006, pp. 1–20. ISBN 978-3-540-32648-9. Available from DOI: `10.1007/11605805_1`.
14. TANENBAUM, Andrew S.; BOS, Herbert. *Modern Operating Systems*. 4th ed. Upper Saddle River: Prentice Hall Press, 2014. ISBN 978-0-13-359162-0.
15. FOTHERINGHAM, John. Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. *Commun. ACM*. 1961, vol. 4, pp. 435–436. ISSN 0001-0782. Available from DOI: `10.1145/366786.366800`.
16. WEISBERG, P.; WISEMAN, Y. Using 4KB page size for Virtual Memory is obsolete. In: 2009 IEEE International Conference on Information Reuse & Integration, 2009. Available from DOI: `10.1109/IRI.2009.5211562`.
17. GRAS, Ben; RAZAVI, Kaveh; BOSMAN, Erik; BOS, Herbert; GIUFFRIDA, Cristiano. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS, 2017. Available from DOI: `10.14722/ndss.2017.23271`.
18. MITTAL, Sparsh. A survey of techniques for architecting TLBs. *Concurrency and Computation: Practice and Experience*. 2017, vol. 29. Available from DOI: `10.1002/cpe.4061`.

19. GRUSS, Daniel; MAURICE, Clémentine; FOGH, Anders; LIPP, Moritz; MANGARD, Stefan. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York: ACM, 2016, pp. 368–379. ISBN 978-1-4503-4139-4. Available from DOI: `10.1145/2976749.2978356`.
20. LOMONT, Chris. *Introduction to x64 Assembly* [online]. 2012 [visited on 2019-04-11]. Available from: `https://software.intel.com/en-us/articles/introduction-to-x64-assembly`.
21. YOSIFOVICH, Pavel; RUSSINOVICH, Mark E.; SOLOMON, David A.; IONESCU, Alex. *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More*. 7th ed. Redmond: Microsoft Press, 2017. ISBN 978-0-7356-8418-8.
22. INTEL. *Intel Transactional Synchronization Extensions (Intel TSX) Overview* [online]. 2019 [visited on 2019-04-21]. Available from: `https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview`.
23. INTEL. *Intel 64 and IA-32 Architectures Optimization Reference Manual* [online]. 2016 [visited on 2019-04-21]. Available from: `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.
24. ORACLE. *VirtualBox* [software]. 2018. Version 5.2.18 [visited on 2019-05-09]. Available from: `https://www.virtualbox.org/`.
25. JAM. *HeavyLoad* [software]. 2019. Version 3.5 [visited on 2019-05-09]. Available from: `https://www.jam-software.com/heavyload/`.
26. BOLDIN, Pavel. *Meltdown Exploit PoC* [online]. 2018 [visited on 2019-05-08]. Available from: `https://github.com/paboldin/meltdown-exploit`.
27. TS'O, Theodore. *linux/init/version.c v4.13.16* [online]. 1992 [visited on 2019-05-08]. Available from: `https://elixir.bootlin.com/linux/v4.13.16/source/init/version.c`.
28. ZHANG, Stephen. *Introducing Linux Kernel Symbols* [online]. 2011 [visited on 2019-05-08]. Available from: `https://onebitbug.me/2011/03/04/introducing-linux-kernel-symbols`.
29. STORMCTF. *Meltdown-PoC-Windows* [online]. 2018 [visited on 2019-05-08]. Available from: `https://github.com/stormctf/Meltdown-PoC-Windows`.
30. CODEMACHINE. *X64 Kernel Virtual Address Space* [online]. 2010 [visited on 2019-05-08]. Available from: `https://www.codemachine.com/article_x64kvas.html`.

31. *Examining Process Page Tables - The Linux Kernel 5.1.0 Manual* [online] [visited on 2019-05-08]. Available from: https://www.kernel.org/doc/html/latest/admin-guide/mm/pagemap.html?highlight=pagemap.

32. AMD. *Software techniques for managing speculation on AMD processors* [online]. 2018 [visited on 2019-05-05]. Available from: https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

33. KRZANICH, Brian. *Advancing Security at the Silicon Level* [online]. 2018 [visited on 2019-05-05]. Available from: https://newsroom.intel.com/editorials/advancing-security-silicon-level/#gs.9wmc7l.

34. BUCHANAN, Erik; ROEMER, Ryan; SAVAGE, Stefan. Return-Oriented Programming: Exploits Without Code Injection. In: Black Hat USA 2008 Briefings, 2008. Available also from: https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf.

35. *Address space layout randomization* [online]. 2019 [visited on 2019-05-05]. Available from: https://en.wikipedia.org/wiki/Address_space_layout_randomization.

36. GRUSS, Daniel; LIPP, Moritz; SCHWARZ, Michael; FELLNER, Richard; MAURICE, Clémentine; MANGARD, Stefan. KASLR is Dead: Long Live KASLR. *Lecture Notes in Computer Science*. 2017, vol. 10379, pp. 161–176. ISBN 978-3-319-62104-3. Available from DOI: 10.1007/978-3-319-62105-0_11.

37. CORBET, Jonathan. *The current state of kernel page-table isolation* [online]. 2017 [visited on 2019-05-06]. Available from: https://lwn.net/Articles/741878/.

38. *KVA Shadow: Mitigating Meltdown on Windows* [online]. 2018 [visited on 2019-05-06]. Available from: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

39. GRUSS, Daniel; HANSEN, Dave; GREGG, Brendan. Kernel Isolation From an Academic Idea to an Efficient Patch for Every Computer. *;Login:* 2018, vol. 43. ISSN 0018-8646.

40. GREGG, Brendan. *KPTI/KAISER Meltdown Initial Performance Regressions* [online]. 2018 [visited on 2019-05-06]. Available from: http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

Appendix A

Acronyms

ASM Assembly language

CPU Central Processing Unit

KASLR Kernel Address space Load Isolation

KPTI Kernel Page-Table Isolation

LLC Last Level Cache

MMU Memory Management Unit

μ**op** Micro-operations

OS Operating System

PT Page Table

PTE Page Table Entry

TLB Translation Lookaside Buffer

Appendix B

Contents of enclosed CD

```
meltdown-exploit-linux ............... the directory with Linux exploit
├── meltdown-exploit-windows ......... the directory with Windows exploit
├── thesis.pdf ....................... the thesis text in PDF format
├── readme.txt ....................... the file with CD contents description
└── thesis_src ....................... the directory with the thesis source code
```