Side-channels and countermeasures (cache-timing attacks, Meltdown, Spectre) *

Alicia Dimroth

Technichal University of Munich, Arcisstrae D-80333 Munich, Germany

Abstract. Between the architectural level of a program and its implementation lie security flaws that side-channel attacks attack. This work is going to talk about how these attacks work and what countermeasures can be used against them. In particular powerful side-channel attacks like cache side-channel attacks, as well as Meltdown and Spectre are going to be regardet.

Keywords: Side-channel · Countermeasures · Meltdown · Spectre

1 Introduction

At the execution of a program, the computer unintentionally exposes information about program properties. The information source can rank from power consumption, timing, acoustics to even electromagnetic fields and is additional to the computational output. While these side effects typically do not hold information about the algorithm itself, they can leave indications about the program implementation. Side-channel attacks use this type of information, which is also called side-channel information, to attack cryptographic systems among others when breaking RSA encryption [5].

Recent discoveries of Meltdown [3] and Spectre [4], which are going to be introduced in the following, also lead to a reintroduction of side-channel attacks and their threat to extracting great amounts of private data across systems into current discussions in the field.

From a security perspective, the danger of side-channel attacks often originates from performance enhancing auxiliaries. As a consequence, the trade-off between performance and security has caused various vulnerabilities in computer systems. Their countermeasures can cause performance loss. RSA encryption is an example where optimization techniques like CRT were introduced that led to better calculation performance, however, were later exploited using a timing side-channel attack [5].

To give the reader an introduction to side-channels and their countermeasures, the rest of this paper first is going to give an overview of different side-channel attacks and how they operate. In Section 3, it is specifically going to discuss cache side-channels and various attack techniques like Flush+Reload are

* Supported by the Fraunhofer Institute.

shown. The next section presents Meltdown, a microarchitectural side-channel attack, followed by the Spectre attacks in section 5. In the end, it will discuss countermeasures and give a conclusion.

2 Overview of Side-Channels

Because most side-channel attacks do not have a particularly high bandwidth, they are often used to break cryptographic systems. Cryptographic systems are a fundamental part of information security since they are used to protect information. This can include encrypting data to ensure confidentality and as a result, it means side-channel attacks have a far reach when attacking cryptographic systems. In addition, when attacking encryption mechanisms, they have the advantage that only a small secret, the encryption key, has to be recovered. The following section is going to give an introduction to cryptographic side-channel attacks. Nevertheless, the techniques can be used analog to attack non cryptographic systems.

As described by Zhou [6], while designing a cryptographic system, the cryptographic algorithm is regarded separate from the implementation. Because of that, side-channel information gets disregarded. The approach of a side-channel attack attacking a cryptographic system is to analyze the implementation and exploit correlations between any input data and the computational side-channel information.

Timing attacks focus on information about the amount of time an algorithm needs for different computations. The principle of timing these computations is to gain information about secret internal parameters in the algorithm that are dependent on the calculation time. To be able to analyze timing information, attacked systems are required to have non-constant run times that depend on a secret key. For example, a RSA timing attack depends on the fact that square and multiply operations have different calculation times that can then be distinguished from the outside[5].

Aside from breaking RSA encryption, timing attacks have been used by Cathalo et al. to break a GPS identification system in less than 100 milliseconds [8], which appropriately shows the magnitude of a timing attack.

In contrast, the target of fault attacks is non-standard program behavior that gets triggered by fault injection. Fault injection describes provoking abnormal conditions, e.g. anomalously high temperature, and then exploiting the resulting behaviour. The program behaviour is very software design and implementation dependent.

Other variants: additional to time and fault analysis, power analysis attacks focus on processor power consumption, EM attacks exploit electromagnetic radiation and acoustic attacks analyze the correlation between computations and the sound of the processor. Further readings on these attacks [9] [10] and additional side-channel attacks [6] are recommended. The rest of this work however, is going to focus on cache side-channel attacks and microarchitectural side-channel attacks.

Above-mentioned timing attacks exploit timing differences dependent on secret algorithm parameters. However, a timing difference also occurs at so-called cache-misses. Cache side-channel attacks do not specify exclusively on cryptoanalytic side-channel attacks. Specifically, they were repeatedly used to build so-called covert channels.

A covert channel is a specific use of side-channels where the attacker both prepares the attack and evaluates it as opposed to waiting for victim activity to provide information to analyze. In other words, the attacker actively provokes side-channel behaviour and then analyzes it. This can be used to transmit information through side-channels.

3 Cache Side-Channel attacks

3.1 Introduction to Cache Side-Channel attacks

Arguably one of the most important side-channels is the cache side-channel. This is because of the high bandwidth, size and central position in computer systems it has to offer.

The term "bandwidth" is introduced as a rate at which information is transmitted through a transmission channel. It affects the amount of data that can be transferred across the side-channel, whereas size refers to the size of the cache, which is proportional to the amount of time an attacker has to get data across this channel. That means cache side-channels are a way to get data fast. Additionally, the central position of the side-channel is caused by the root of this vulnerability, which is the CPU. This leads to a far reach of cache side-channel attacks since all software depends on the CPU and therefore, all software is impacted.

Due to these traits it is possible to achieve a side-channel with a comparatively high bandwidth that allows an attacker to stream an HD video though cache at real time, as shown by Anders Fough [1]. Particularly interesting is also the role of cache side-channels in the Meltdown and Spectre attacks that have been discovered by M. Lipp et al. [3] [4].

The way cache side-channel (CSC) attacks work is by exploiting the microarchitectural design of the CPU. Simply put; cache side-channels take a look at how an application uses memory, which every application does, and by which it leaks information about what it is doing. A CSC attack uses and abuses this information.

The part of memory this paper is going to look at while dealing with CSC are called CPU caches. They are a consequence of the memory hierarchies in modern processors. In particular, the focus is going to be on the L3 cache, also known as the Last-Level Cache, or LLC.

In the following subsections, the fundamental traits of memory hierarchies and important cache features are going to be discussed. Second, we can take a closer look at common CSC functionality and look at three big cache side-channel attacks called Evict+Reload, Prime+Probe and Flush+Reload.

3.2 Background

This subsection goes into more detail about memory hierarchies and the Intel LLC in particular.

Memory Hierarchies Because read/write operations to the main memory are performance inefficient, memory hierarchies were invented. The idea is for the processors to use much faster and smaller cache memory as a way of gaining performance efficiency. The way this works is that every cache contains a part of the memory the processor needs to access and by that reducing the number of times the processor needs to access the main memory. As shown in Figure 1, each CPU core has access to a L1, L2 and a, much larger, L3 cache (LLC).

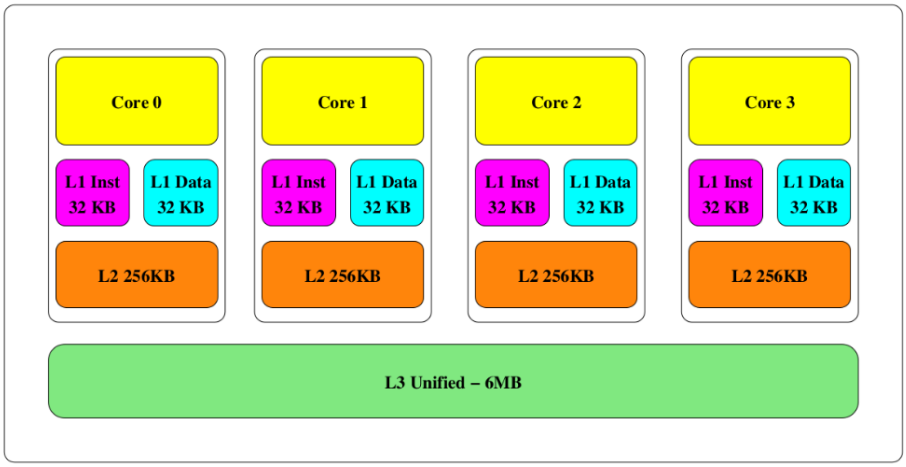

Fig. 1. Intel Ivy Bridge Cache Architecture as exemplified by Yarom et al. [5]

If the processor wants to access a memory address, the information request first goes to the L1 cache. If the required memory address is not stored there, it is called a cache miss. After a cache miss occurs, the processor looks at the L2 cache and finally the L3 cache (or LLC).

However, when a cache miss occurs in the LLC, the processor has to fetch the address from the main memory. This adds latency and the processor has to copy the address into the cache. On average, an instruction where the needed data is cached takes about 80 clock circles, whereas uncached data causes a latency up to about 200 cycles [1].

LLC As shown in Figure 1 L3 cache is across all CPU cores. It can also be divided in so-called slices, assigned to one core each. The slices are connected through a ring bus. Here, it is important to mention that there are no ring

bus privileges, meaning information in one LLC slice can be accessed by all CPU cores. Cache side channel attacks exploit the fact that every CPU core has access to all cache slices. As a consequence, an attacker does not have to share the same execution core as the victim.

Apart from that, another cache property to be mentioned is that the Intel cache hierarchy is inclusive. It means in effect that the information stored in the L1 cache is a subset of the information in L2 and L3 caches. Additionally, everything deleted in the LLC also gets deleted from the higher-level caches.

By using these cache properties, attackers are able to manipulate the cache and by that get information about the data stored in the cache. This can be achieved over three basic procedures; *evict*, *flush* and *prime* [1].

Evicting the cache means that the attacker accesses a memory from the cache until the given address is no longer cached.

In contrast, *flushing* the cache can be achieved by removing a given address from the cache. Here, the *clflush* instruction is often used. *clflush* is an unprivileged Intel instruction to evict a specific memory line from all parts of the cache hierarchy.

Lastly, to *prime* the cache is to place a known address in the cache.

In the following, if caches are mentioned, focus is set on the Intel LLC cache, however, the attack principles can often also be implemented for higher level caches. Moreover, CSC attacks on L1 caches have a higher bandwidth than LLC side-channel attacks since cache access time is directly dependent on the distance from cache to the core [5].

3.3 CSC

The attacks presented in the following section are called Evict+Time, Prime+Probe and Flush+Reload. A special use case of these attacks is building covert channels. However, they can also include, but are not limited to, attacks on cryptographic algorithms [2] or web server function calls [1]. Each of the techniques work with three basic steps. First, they use *evict*, *prime* or *flush* to somehow manipulate the cache state. Second, they wait for some victim activity and in the final step, the attacker examines the change that occurred.

The Evict+Time Technique Evict+Time is a CSC variant that relies on the call of a victim function to reveal information. To set up the cache state, the victim function has to be executed. After that, it is definite that the memory lines that are needed are cached. At the next call of the victim function, the attacker times the execution of the function. Now, he knows a time baseline (also called threshold) for the function execution.

Next, *evict* is used to purposely evict a chosen address from the cache and the victim function gets called a third time. Now, if the third execution takes more time than the previously gathered baseline, the function used an address congruent to the cache set that got evicted.

This technique has been used for example to extract the keys of an Advanced Encryption Standard (AES) encryption [7]. However, Evict+Time tells the attacker no information about what happens inside the function call. Additionally, it requires a victim function call which limits the attack. Because of that, a more powerful variant to Evict+Time is the Prime+Probe technique.

The Prime+Probe Technique The Prime+Probe variant does not rely on a victim function. Instead, it first *primes* the cache to contain an address that the attacker chose. Then, it waits for victim activity, after which the attacker accesses the chosen address again and times how long the access takes. If the access is slow, it means that there was a cache miss and that the victim loaded different memory in the cache, and by that suppressing the previously chosen address. Because the attack does not rely on the execution of a specific function, the technique can be repeated a lot faster.

Since this technique, among others, works for Java Script, the reach of this attack also includes remote exploits and targets browser function calls [1]. A slightly different and even faster CSC variant is called Flush+Reload, however, both can be used to build highly efficient attacks.

The Flush+Reload Technique Flush+Reload is a particularly interesting variant of CSC attacks since it can be used more granular on a single cache line as opposed to a cache set like in Evict+Reload and Prime+Probe. That is why, even though it is not only possible to build high-performance side-channels using Flush+Reload, it is said to be the most accurate known side-channel [3]. Because of that, other attacks, e.g. Meltdown, use Flush+Reload as a covert channel. Additionally, if it is used on the LLC, Flush+Reload can be operated cross-core.

The three basic steps of Flush+Reload are as follows: first, the attacker uses *clflush* to *flush* a shared address from the cache. Second, the attacker waits for victim activity and lastly, he times an access to the address that got flushed. If step three was fast, the attacker knows that the address was placed in the cache again by the victim. Analogous, if step three was slow, the victim did not use the address, instead, it has to be brought from memory which adds latency and the execution of this step becomes significantly longer.

When Flush+Reload is used for the building of a covert channel the victim activity is also executed from the attacker. If not however, the address has to be accessible by both processes. In addition, suppose the attacker and the victim processes are executed on different CPU cores. It seems as if Flush+Reload has the limitation that the flushed cache line has to be from shared memory. If not, the victim activity would never cache the shared address and no information would be gained.

Luckily, shared memory is not very rare. Shared libraries, libc for example, are used by most programs. Another big contribution to shared memory is deduplication, also called content-based page sharing. Deduplication is a measure to minimize memory usage by removing duplicated memory. The idea is that if two

pages have identical content, the system removes one of them to gain memory efficiency.

4 Meltdown

To isolate different processes from each other and thereby protect data from one process against others, operating systems implement memory isolation. Memory isolation is a vital requirement for protecting kernel memory and preventing different user processes to read each other's data.

This section is going to focus on Meltdown; currently one of the most powerful known-of side-channel attacks that was discovered by Lipp et. al [3]. Meltdown is said to "break[...] all security assumptions given by CPU's memory isolation capabilities" [12].

Essentially, Meltdown exploits a side effect of out-of-order execution to read unprivileged data. On an architectural level (or abstract level), by using multiple execution units and thereby compensating instruction workload, out-of-order execution is able to optimize performance. It prefetches future instructions and momentarily executes them. In fact, out-of-order execution performs nearly up to 200 future instructions, also depending on values that have been calculated in advance [13]. Then, if the program is terminated before it executes the prefetched instructions, they get discarded.

It is important to mention that at out-of order execution, no privilege checks are performed. That is caused by the fact that if there is an unprivileged access, an exception will be thrown and all out-of-order executed instructions should be reverted.

So in this perspective, out-of order execution should not pose a security problem. Unfortunately, out-of-order executions leave traces in the cache and Meltdown exploits the indicated side-channel.

To understand Meltdown techniques, first this work gives more detailed background to how memory isolation works and what part out-of-order execution plays in the Meltdown attack. Then, it will give a more detailed description of the steps that Meltdown consists of, by looking at an example pseudo code.

4.1 Background

Memory Isolation The way memory isolation works is by operating over virtual address spaces, where every process gets an address space assigned to. This allows processes to run regardless to other processes. Every virtual address space is divided into pages. Pages are either assigned to a so-called user space or kernel space. The differentiation of user and kernel space is very important because user space only stores information that the user has access to, in contrast to operating in kernel space, which gives access to all user's memory. Therefore, kernel space should not be accessed by a user process. To protect kernel memory, kernel space pages can only be accessed if the CPU runs in privileged mode.

Nowadays, the privileged mode is generally implemented by a supervisor bit that determines whether a page can be accessed by a user process or not. When the system is in kernel mode, the supervisor bit is set and at context switch, when the system switches from kernel mode to user mode, the bit gets cleared. On one hand, this makes context switches much more performance efficient than if the address space would not include the kernel space.

On the other hand, it also poses a major security risk since the mapping of physical memory in kernel space allows Meltdown not only to read data from user space, but also the whole physical memory.

Out-of-Order Execution Out-of-order execution is a mean to optimize performance by executing instructions in advance. Instead of using one execution unit to sequentially execute all program instructions, multiple execution units can be used. They work parallel to look ahead of the current instruction and execute future instructions before the processor completely executed the current one.

Furthermore, out-of-order execution implements speculative execution in the sense that when performing out-of-order execution, it is not certain whether the instruction will be executed completely. However, on an abstract level, a false prediction poses no problem because it is only a temporary state. If the speculation is right, the instructions can be executed immediately, else the instruction gets discarded and any temporary changes that were made get reverted.

In particular, when executing out-of-order, the instructions get fetched, decoded into micro OPs and finally held in a so-called Unified Reservation Station until they are needed. If an operation needs an operand that is not available yet, it listens on the common data bus (CDB) until it arrives. When an instruction needs a register to store values, it uses the Reorder Buffer.

Given it turns out that speculatively executed instructions are not needed, the Unified Reservation Station gets reverted to a state without falsely fetched instructions and the Reorder Buffer gets cleared of all related data. At this point, it is important to mention that, even though the Reorder Buffer gets cleared, the cache is not influenced. On an abstract level this makes sense since data cannot be read directly from the cache. However, this is the security flaw that Meltdown exploits to read unprivileged memory.

4.2 Attack

Meltdown can be divided into two steps. First, the adversary uses out-of-order execution to prepare the cache to a state that can disclose data. Then, the data gets reconstructed by analyzing the cache state.

In the following section, this paper is going to look at the two steps that Meltdown consists of, while referring to the toy example consisting of Algorithms 1-3.

Algorithm 1 Meltdown

```
1: procedure ExtractDataFunction
2:     flushFromCache( probeArray ) ;
3:     prepareCacheFuntion( kernelSpaceAddress ) ;   ▷ As shown in Algorithm 2
4:     data = analyzeCacheFunction() ;               ▷ As shown in Algorithm 3
```

"Step 0" Before focusing on step 1 and step 2, the framework program in Algorithm 1 shows how the steps get connected.

Since the final objective is to read the content of an unprivileged address, a straight forward thought would be to simply place the address in the cache and later read it. However, it is not possible to simply read from the cache.

Because of that, the basic idea of Meltdown is to first prepare the cache (line 2 and line 3) and then extract the content of the address by analyzing the cache (line 4). Analyzing the cache is different to reading from it, because it uses a more indirect approach. An example would by to exploit cache timing behavior to extract information. Since the adversary prepares the cache as well as analyzes it, this step can be described as building a covert channel.

Now, this work takes a closer look at the individual steps.

Step 1 The first step has the purpose of preparing the cache.

Algorithm 2 Meltdown

```
1: procedure PrepareCacheFunction(kernelSpaceAddress)
2:     data = contentOf( kernelSpaceAddress ) ;      ▷ This raises an exception
3:     // the following will not be executed
4:     access( probeArray[ data × 4096 ] ) ;
```

Algorithm 2 shows a toy example for how Meltdown uses out-of-order execution to modify the state of the cache in a way that can be analyzed later. The program first tries to read the content of an address into *data* and after that performs a regular array access depending on the value stored in *data*.

Suppose in line 2 of Algorithm 2, the adversary tries to read the secret value stored at a kernel address. Because the program does not run in kernel mode, Algorithm 2 will raise an exception trying to execute line 2. However, before an exception is raised, the value of *data* will be placed on the CDB. The CDB delivers the value to the Unified Reservation Station, in which micro operation that need *data* as an operand might be waiting.

This leads to a race condition between the raising of the exception and the speculative execution of line 4. If the exception will be raised before the operations from line 4 can perform, the register holding the *data* value gets overwritten with a zero value and no inadvertent states get formed.

Here, it can still be the case that the speculative execution occurs. However, only the first field of *probeArray* will be cached and this holds no side-channel

information for the adversary. The consequences of this case are discussed by Lipp et al. [3].

In contrast, if line 4 gets speculatively executed, depending on the value of *data*, a part of the *probeArray* will be accessed. Consequently, as part of the access, the content of one field of *probeArray* will be cached. The factorial of 4096 in line 4 is dependent on page sizes and the size of the type of *probeArray*. It prevents that multiple array fields get cached consecutively. Nevertheless, as soon as the exception is raised, any changes in the Reorder Buffer and the Unified Reservation Station get discarded. After that, the exception either gets handled or the program terminates.

To sum up, despite the program never executing line 4, the cache state has changed. By that, Meltdown successfully circumvented memory isolation. In step 2, the changed cache state gets analyzed to recover the value of *data*.

Step 2 Before Algorithm 3 can be useful, it is important to understand how line 2 and line 3 of Algorithm 1 changed the cache state.

Since analyzing technique in step 2 works analog to the Flush+Reload technique as shown in section 3.3, the adversary needs to prepare the cache first. This is done by flushing the whole *probeArray* and then preparing the cache with Algorithm 2. After flushing the *probeArray* and executing Algorithm 2, the adversary knows that only one field of the *probeArray* is cached. Namely, the field that was tried to be accessed in line 4 of Algorithm 2.

Now, to analyze the cache state, Meltdown proceeds by using the Flush+Reload technique to detect whether an address is in the cache or not.

Algorithm 3 shows a pseudo code representation of the analysis.

Algorithm 3 Meltdown

```
1:  procedure AnalyzeCacheFunction
2:      i = 0 ;
3:      while i < probeArray.length do
4:          accessTime = time( access( probeArray[i] ) ) ;
5:          if accessTime < treshhold then
6:              data = i ÷ 4096 ;
7:              return i ;
8:          else
9:              i + + ;
10:     // After while retry Algorithm 1
```

In general, this function iterates over *probeArray*. For each field of *probeArray*, the program acts analog to Flush+Reload. It times an access to the field and compares it to a threshold value. This threshold determines whether a cache miss occurred, whereby fast access implies that the field of the array was cached. Also to be mentioned is, that the value of the threshold is system dependent on

additional factors such as cache level or load times and can be calculated as discussed in section 3.3.

As soon as the time an access to a field of the *probeArray* takes is less than the threshold, the adversary knows that the field was cached. The number of the field that is cached is directly dependent on the content of the kernel address. Since step 2 only cached one field of the array, the value of *data* can be reconstructed as shown in line 6.

In contrast, if no array field is cached, it indicates that the exception was raised before line 4 in Algorithm 1 was speculatively executed. In that case, the procedure is repeated in line 10.

4.3 Reach of attack

Meltdown is able to dump the entire physical memory by repeating Algorithm 1. Due to the execution time of Algorithm 1 being very small, Meltdown has a high bandwidth. Lipp et al. achieved a performance of Meltdown with 503 KB/s [3].

Apart from having a high bandwidth, Meltdown also affects a wide range of systems. Since the attack exploits the way Intel processors implement out-of-order execution, it can be successfully on Intel processors. It also proved to be effective on ARM microprocessors [3]. At its discovery, because the problem lies within the hardware level of the implementation rather than software, it is difficult to counteract the root of the attack using software. Due to this, multiple operating systems like Linux, OS X and Windows were unable to prevent Meltdown [3].

Additionally, Lipp et al. also discovered that Meltdown works well in containers sharing the same kernel, where the isolation can be broken [3]. This is particularly critical if hosting providers implement containers as part of the virtualization. If that is the case, Meltdown is able to act across different users.

Due to the far reach of the attack, countermeasures are a vital requirement to information security.

4.4 Countermeasures

Since the vulnerability that Meltdown exploits is not software dependent, but has its root in hardware, there is no software solution to counteract the root of the attack. Hardware solutions on the other hand are expensive and time-consuming since they would require a redesign of CPUs. However, there are software patches trying to mitigate the effects of Meltdown, but they come with a significant performance loss.

Additionally, shortly after the discovery of Meltdown, Intel introduced KAISER, a software patch that was integrated into operating systems. KAISER prevents mapping from kernel memory into user space almost completely and as a consequence, prevents Meltdown to access kernel memory. Since it is necessary for some kernel memory to be mapped into user space, there still is vulnerable kernel

memory. For an alternative, Lipp et al. suggested a hard split of user and kernel space [3].

5 Spectre

Out-of-order execution brings a significant performance enhancement on modern CPUs. When executing out of order, speculative assumptions about the sequential program procedure have to be made.

In the context of Meltdown, this work already focused on speculations about program behavior. While Meltdown speculated about reaching the instruction before the program terminates (Section 4.1), speculative execution also includes speculations about branch predictions.

After taking a look at Meltdown, this work is going to focus on Spectre, a set of powerful side-channel attacks that attack branch predictions. Spectre, as well as Meltdown can be called microarchitectural side-channel attacks since they are all variants of the same technique that attacks a side-channel located on the microarchitectural level and that was caused by speculative execution.

5.1 Difference to Meltdown

Spectre and Meltdown are similar in the way that they both exploit the fact that speculative execution is allowed to execute erroneous code, which leaves traces in the cache. However, Spectre, in contrast to Meltdown, does not attack the race condition between privilege checks and speculative execution. Rather, it focuses on attacking the implementation of branch prediction.

Without depending on a race condition in the execution engine, Spectre works more independently of the implementation of privilege checks in speculative execution. Because of that, Spectre attacks can be applied to a broader range of processors.

Before discussing Spectre variants, the next section introduces the branch prediction implementation and gives background information about conditional and indirect branches.

5.2 Background

Branch Prediction For instance, branch predictions can occur in conditional branches and indirect address references.

Algorithm 4 Spectre

```
1: procedure ConditionalBranch
2:     if x < array1.size then
3:         y = array2[ array1[x] ] ;
```

Algorithm 4 shows a conditional branch example as introduced by Kocher et al. [4]. Since the value of x is dependent on other operations, it is not available at out-of-order execution. Therefore, an instance called branch predictor speculates whether the condition is true or false. If the speculation turns out to be true, this leads to better performance. In contrast, if the prediction was false, the processor rolls back to a state without the false prediction. Section 5.3 is going to focus on exploiting conditional branch misprediction.

Another attack exploiting branch prediction operates in the context of indirect branches.

Algorithm 5 Spectre

```
1: procedure IndirectBranch
2:     jmp [eax] ;
```

An example for indirect branching is shown in Algorithm 5. Since the following program execution depends on the content of register *eax*, this program can easily be redirected by modifying the content of *eax*. How this can get exploited is discussed in section 5.3.

Both, conditional and indirect branch misprediction, require a speculation about the direction a branch in the program sequence will take. Hereby, the processor looks for the most likely outcome. This is implemented by using a Branch Target Buffer (BTB). The BTB consists of a mapping of recently used branch instruction addresses to their jump destination. By modifying a table entry, an adversary could change the instruction sequence of speculative execution.

In addition, because of memory efficiency, the BTB holds only the 30 least significant bits of the key component [4]. This makes attacks on the branch prediction implementation more flexible.

For conditional branches, the processor additionally holds a record of recent branch outcomes. Since the prediction depends on old outcomes, old outcomes influence speculative execution of conditional branches. The next section shows how an adversary can modify this implementation to leak information.

5.3 Attacks

Conditional Branch Misprediction The condition in Algorithm 4 is a security check if an array access is within the bounds of said array. This should prevent attackers from reading secret user information. However, Spectre implemented a way to extract secret information by abusing the condition in line 2 by exploiting Conditional Branch Misprediction.

The basic idea of the technique is to first train the branch predictor to expect a chosen outcome and then trigger a false speculative execution that leaks user information. Suppose, x in Algorithm 4 is chosen by the attacker.

Since the processor holds a record for previous branch outcomes, the attacker would first train the branch predictor to expect the condition to evaluate to *true*.

This can be done by executing Algorithm 4 with valid x. In the second step, the attacker executes Algorithm 4 with a malicious value x.

When the prediction is false and line 3 of Algorithm 4 gets executed with an invalid x, this results in a cached field of *array2*, depending on the value of *array1[x]*. Analog to Meltdown, if the array was flushed from the cache before caching the exploitable array field, the attacker can retrieve the content of *array1[x]*.

However, because Spectre does not attack privilege checks (in contrast to Meltdown), Spectre is limited to reading user address information.

Indirect Branch Misprediction Indirect Branch Mispredicitons attacks code that contains indirect jumps since they are able to jump to more than one possible address. It consists of two building blocks.

The first one is to detect and modify vulnerable code in the victim program so that it accesses secret user information. Modified vulnerable code is also called a gadget. The second building block is to find a way to speculatively execute the gadget.

Kocher et al., one of the groups that developed Spectre, did an example implementation [4]. For vulnerable code they chose two instructions. The first one modified one register the attacker can control and the second one accesses memory at the address of the modified register. This makes it possible to leak memory of an address the attacker chose.

To speculatively execute the gadget, a jump instruction in the program code has to jump to the gadget. For instance, if the branch predicts that *eax* in Algorithm 5 to contain the address of the gadget, the malicious code gets speculatively executed.

In order to mistrain the branch predictor, the BTB has to create an entry from the jump instruction to the gadget, so that speculative execution continues at the gadget.

5.4 Countermeasures

Since Spectre depends on branch prediction, serializing instructions is effective to prevent Spectre. However, it only performs a workaround of Spectre, instead of counteracting the attack.

A countermeasure might be to disable speculative execution. While this would pose a long-term solution, it comes with significant performance loss and therefore is too costly.

6 Countermeasures and trade-offs

The difficulty of counteracting side-channel attacks is that every computer component contributes to leaking side-channel information. Therefore, no unified solution can be applied.

However, multiple software-based as well as hardware-based countermeasures have been invented.

Software-based countermeasures include randomization and generalization of operations. Randomization and generalization have great significance for protecting cryptographic systems since introducing noise to input data de-correlates side-channel information from algorithmic calculations. Square-Multiply-Always for instance is a countermeasure that helps prevent timing attacks on RSA encryption by generalizing side-channel information [5].

However, software-based countermeasures often introduce unnecessary operation overhead. As a result, when using software patches it has to be considered that they often lead to memory and time inefficiency.

In contrast, hardware-based countermeasures operate on hardware level to add random side-channel noise and by that mask which side-channel information is caused by software. Clock randomization for instance helps mitigate the granularity of timing attacks. A randomization of instruction set execution and register usage also shows effect on power analysis side-channels [11].

A disadvantage is that changes in hardware design can come with fundamental changes in system behavior.

7 Discussion

A unified problem for software impacted from side-channel attacks is that the design process views implementation as a black-box that does not leak information. Because of that, possible side-channels are not regarded and only discovered later. For instance the implementation of speculative execution was reported to leave traces in the cache [1] long before it led to the discovery of Meltdown and Spectre attacks.

This shows that to prevent side-channels long-term, they have to be taken into account when designing software.

In addition, Meltdown and Spectre lead to a great urgency to find short-term side-channel countermeasures. The problem with hardware-based countermeasures is that they are long-term solutions since they take time to develop. Additionally, they do not protect existing hardware. In conclusion, short-term countermeasures have to be software-based solutions. This adds to the urgency of providing software design that keeps attention to exploitable side-channels.

8 Conclusion

This seminar work showed that side-channel attacks pose a significant threat to current information security. In order to enhance the resistance of current computer systems, software-based countermeasures need to be deployed. However, when focusing on future systems, redesigning hardware in account to side-channel leakage might be necessary.

In conclusion, even though future countermeasures might come with performance loss, in current systems, prioritizing security is a necessity to prevent large-scale exploits.

References

1. Fogh, A.: Talk. CONFERENCE 2016, Amsterdam
2. Yarom, Y.: FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack, pp.6–9. The University of Adelaine (2016)
3. Lipp, M., Schwarz, M., Gruss, D. et al.: Meltdown, University of Pennsylvania and University of Maryland
4. Kocher, P., Genkin, D., et al.: Spectre Attacks: Exploiting Speculative Execution, University of Pennsylvania and University of Maryland
5. Yarom, Y.: FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. The University of Adelaine (2016)
6. Zhou, Y., Feng, D.: Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. State Key Labatory of Information Security
7. D. A. Osvik, A. Shamir, and E. Tromer: Cache attacks and countermeasures: the case of AES, 2005.
8. J. Cathalo, F. Koeune, J.J. Quisquater. A New Type of Timing Attack: Application to GPS (2003). p. 1
9. Kocher, P., Lee, R., et al.: Security as a New Dimension in Embedded Systems. Proc of the 17th International Confernce on VLSI Design. (2004)
10. Curtsinger, C., et al.: Statistically sound performance evaluation. In Proceedings of the Workshop on Architectural Support for Programming Languages and Operating Systems. (2013)
11. D. May, L.H. Muller, N.P. Smart. Random register renaming to foil DPA. (2001). pp.28-38
12. Lipp, M., Schwarz, M., Gruss, D. et al.: Meltdown, University of Pennsylvania and University of Maryland, p. 2
13. Kocher, P., Genkin, D., et al.: Spectre Attacks: Exploiting Speculative Execution, University of Pennsylvania and University of Maryland, p.11