Politecnico di Torino

Meltdown and Spectre

EXPLOITING CRITICAL VULNERABILITIES IN MODERN PROCESSORS

Embedded Systems Security

Prof. Di Carlo Stefano

Davide Giuffrida - s310265
Matteo Fragassi - s317636
Elena Roncolino - s304719
Umberto Toppino - s277916

Co-funded by the European Union

La Région Auvergne-Rhône-Alpes

AGENCE ERASMUS+ FRANCE · EDUCATION FORMATION

Erasmus+

Acknowledgments

- 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
  - Elena Roncolino
  - Umberto Toppino

License Agreement

Co-funded by
the European Union

Outline

- What are Meltdown and Spectre
- How does Meltdown work?
- What about Spectre?
- Our implementation
- Conclusions

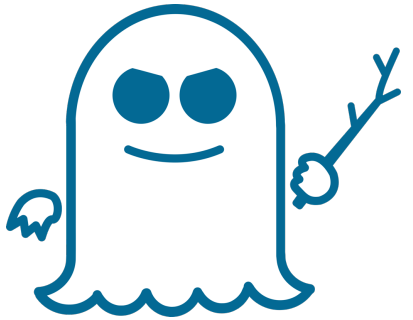
Co-funded by
the European Union

https://meltdownattack.com/

What are Meltdown and Spectre

Hardware vulnerabilities affecting a wide range of modern processors

- Meltdown:
  - Breaks the most fundamental isolation between Operating System and application
  - Allows an attacker to access the memory of other programs and the Operating System
- Spectre:
  - Breaks the isolation between different applications
  - Allows an attacker to trick a error-free application into leaking its secrets



Co-funded by the European Union

Figures taken from: https://meltdownattack.com/

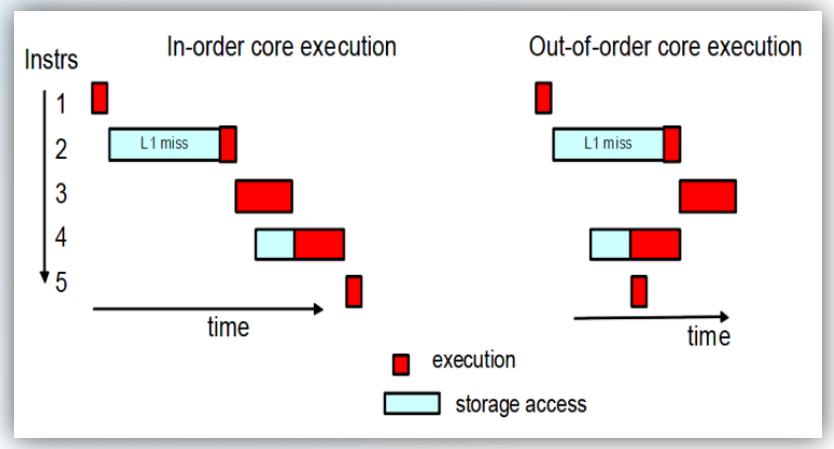
What are Meltdown and Spectre

- They exploit three of the major designs of modern processors:
  - Out-of-order Execution
  - Speculative Execution
  - Caching

Co-funded by
the European Union

What are Meltdown and Spectre

- They exploit three of the major designs of modern processors:
  - Out-of-order Execution
  - Speculative Execution
  - Caching



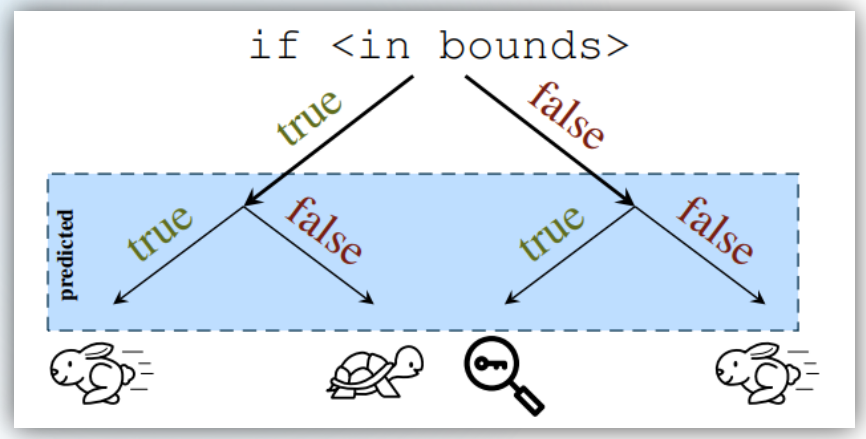
The processor executes the program instructions not in the order in which they appear, but in the order that is more convenient and that best exploit the resources.

Co-funded by the European Union

Figure taken from: http://zseries.marist.edu/pdfs/ztidbitz/28%20zNibbler%20%28z196%200-0%20Execution%29.pdf

What are Meltdown and Spectre

- They exploit three of the major designs of modern processors:
  - Out-of-order Execution
  - Speculative Execution
  - Caching



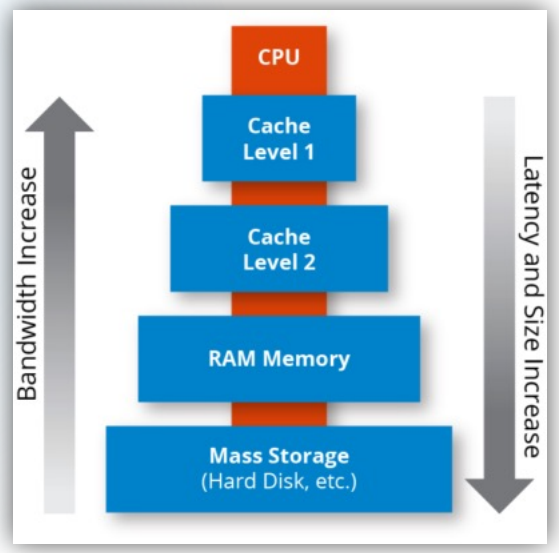
The CPU may execute certain tasks ahead of time, "speculating" that they will be needed. If the tasks are not required, their changes are reverted to the previous state.

Co-funded by the European Union

Figure taken from: https://meltdownattack.com/

What are Meltdown and Spectre

- They exploit three of the major designs of modern processors:
  - Out-of-order Execution
  - Speculative Execution
  - Caching



A small and fast memory (the cache) stores data frequently accessed by the CPU. Exploits temporal and spatial locality to speed up the data access and program execution.

Figure taken from: https://hazelcast.com/glossary/memory-caching/

Co-funded by the European Union

How does Meltdown work?

- Meltdown exploits out-of-order execution to execute instructions acting on data outside of legal memory before an exception is raised

- It can be divided in three steps:
  - Step One: A memory location inaccessible by the attacker is loaded into a register
  - Step Two: A transient instruction accesses a cache line based on the previous data
  - Step Three: The attacker determines the accessed line and retrieves the secret data

Co-funded by
the European Union

How does Meltdown work?

- Following the previous three steps, Meltdown can access all data locations in memory, leaking information it shouldn't have access to.
- The transient instructions act as a covert channel to transfer the data from the kernel memory to the attacker

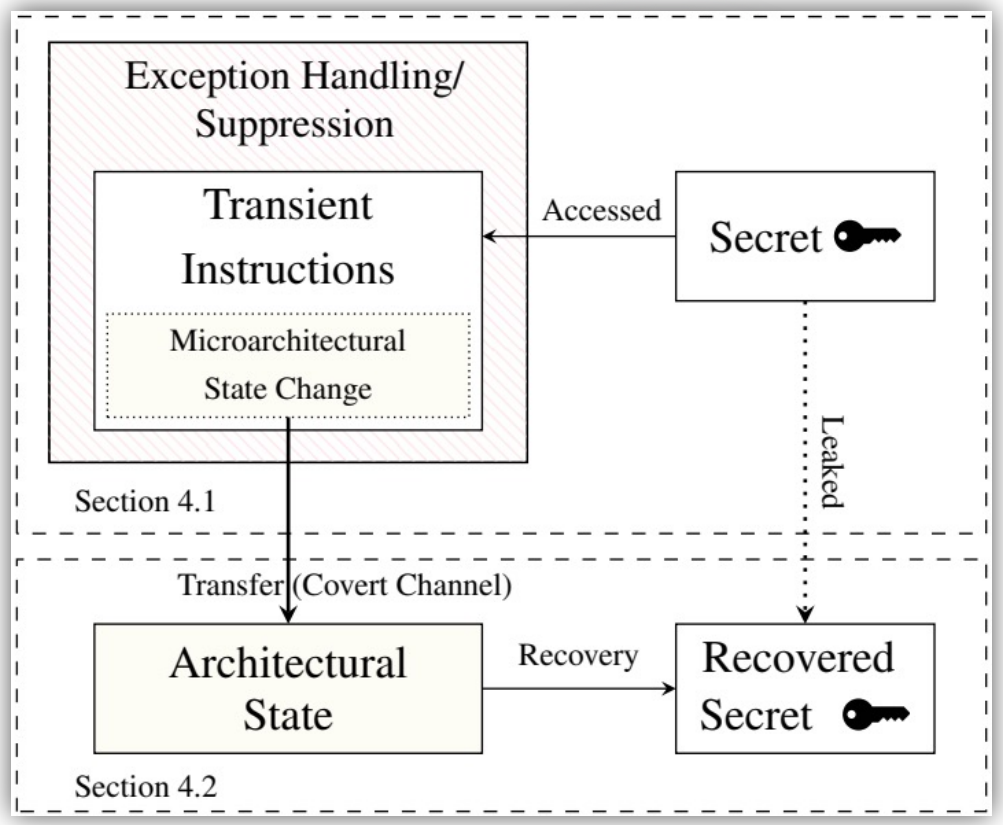


Figure taken from: https://meltdownattack.com/meltdown.pdf

Co-funded by
the European Union

What about Spectre?

- Spectre can be implemented in two different variants:
  - Exploiting conditional branch misprediction
  - Poisoning indirect branches

- Both variants follow the same general steps:
  1) Setup: mistraining of speculative predictions
  2) Transfer: access secret data exploiting speculative (mis-)execution
  3) Recovery: monitor cache access times to retrieve the secret data

Co-funded by
the European Union

Spectre: Variant 1

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

- The first variant exploits conditional branch misprediction: let's consider an example with the code above:
  - `x` is an integer received from an untrusted source
  - `array1[]` is an array of bytes of size `array1_size`
  - `array2[]` is an array of bytes of size `1MB`

Co-funded by the European Union

Figure taken from: https://spectreattack.com/spectre.pdf

Spectre: Variant 1

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

- Phase One: the Setup
  - Previous operations receive valid values of `x`, leading the branch predictor to assume that the `if` statement will likely be `true`
  - `array1_size` and `array2` are uncached
  - The value of `x` is maliciously chosen so that `array1[x]` resolves to a secret byte `k` in the victim's memory
  - The secret byte `k` is cached

Co-funded by the European Union

Figure taken from: https://spectreattack.com/spectre.pdf

Spectre: Variant 1

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

- Phase Two: the Execution
  - The processor tries to compare `x` and `array1_size` causing a cache miss since `array1_size` is not present in cache
  - While waiting for `array1_size`'s value, the branch predictor executes the following line assuming that the `if` statement will be `true`
  - The CPU requests `array1[x]` which returns the secret `k`
  - `array2[k*4096]` is requested affecting the cache in a way that depends on the value of `k`

Co-funded by the European Union

Figure taken from: https://spectreattack.com/spectre.pdf

Spectre: Variant 1

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

- Phase Three: the Recovery
  - Although the code execution resumes normally reverting the changes of a wrong branch prediction, the cache state remains affected!
  - By measuring access times of `array2`'s data it's possible to identify the cached data at address `array2[k*4096]`

Figure taken from: https://spectreattack.com/spectre.pdf

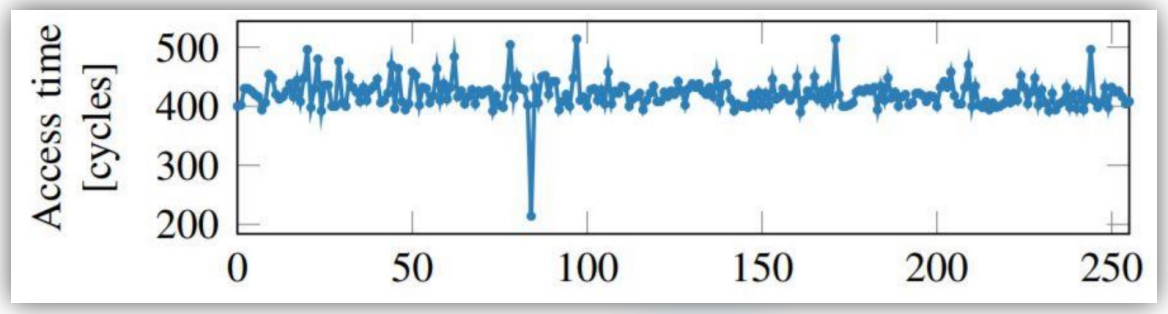
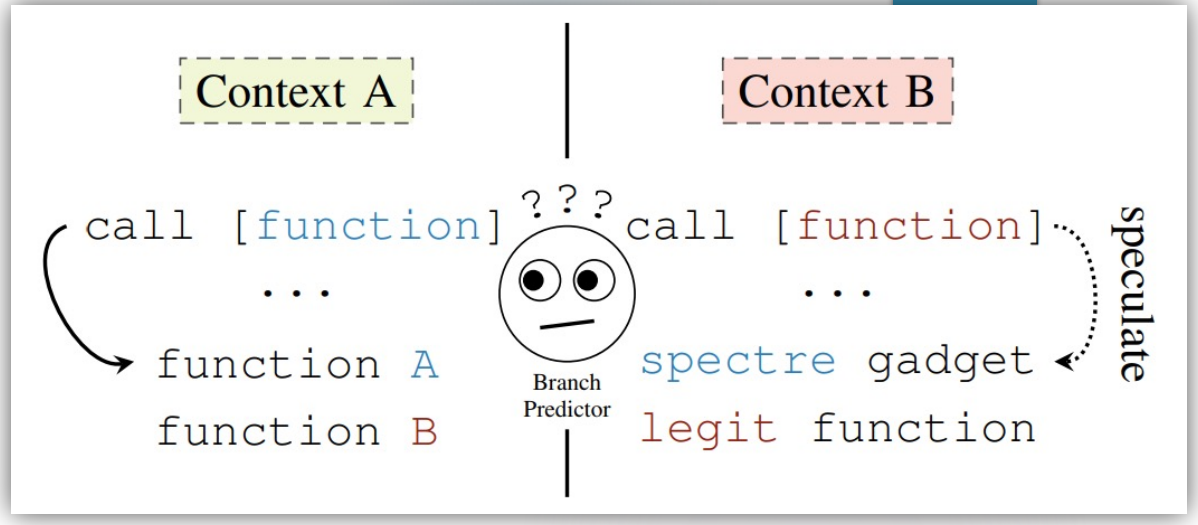


Figure taken from: https://meltdownattack.com/meltdown.pdf

Co-funded by
the European Union

Spectre: Variant 2



- The second variant exploits indirect branch poisoning
  - Indirect branching: jumping to code at some arbitrary location:

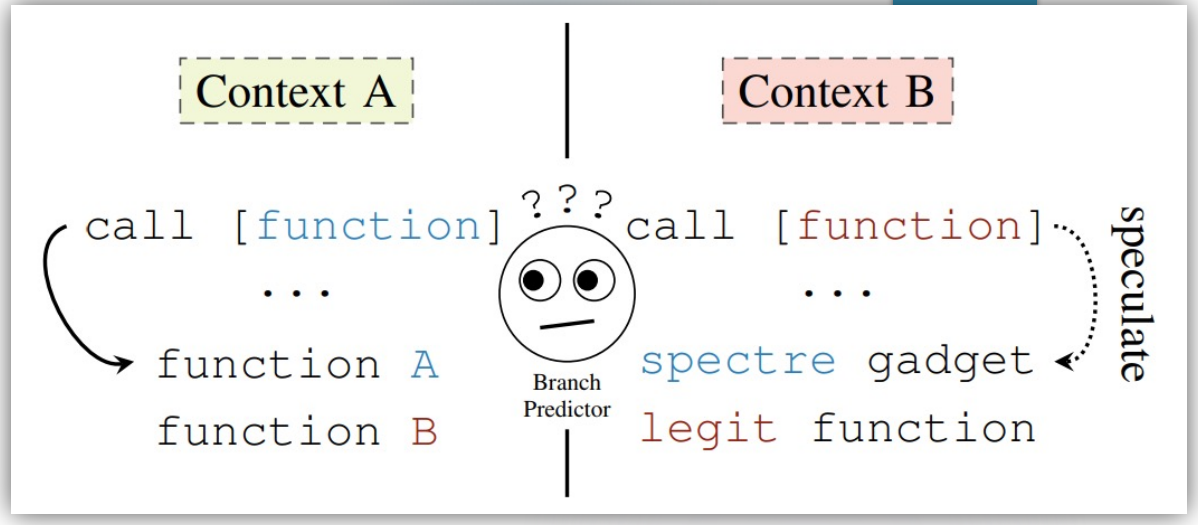
    `jmp[eax]` : jump to instruction stored at the address in register **EAX**

- As shown in the above picture, the mistraining in contex A brings the CPU to speculatively execute a malicious function in context B

Co-funded by the European Union

Figure taken from: https://meltdownattack.com/meltdown.pdf

Spectre: Variant 2

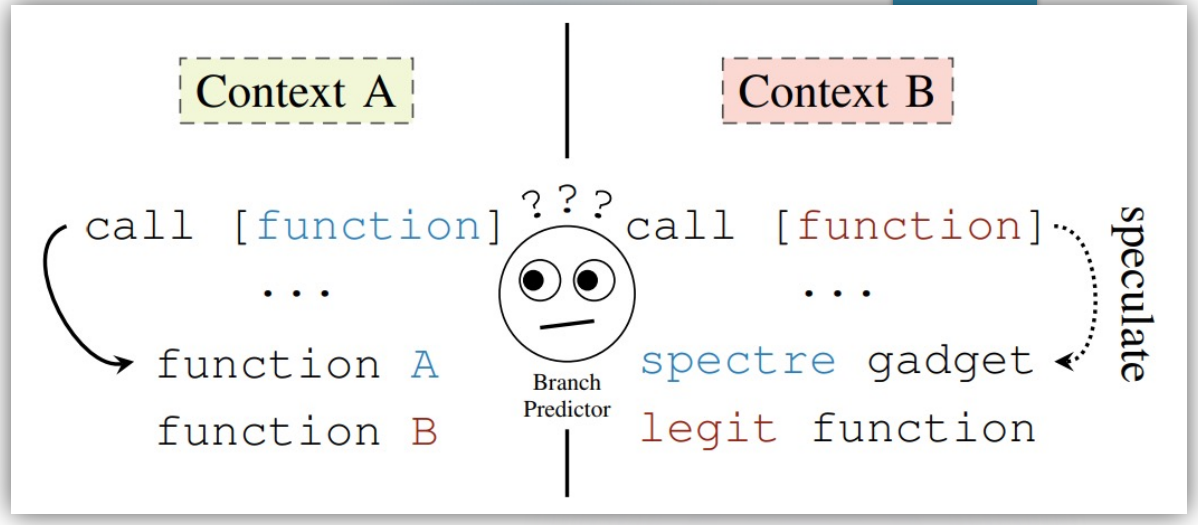


- Phase One: the Setup
  - A suitable function, called gadget, is selected: it will allow the secret data to be leaked (similarly to variant 1)
  - The branch predictor is mistrained by repeatedly supplying the address of the Spectre gadget

Co-funded by the European Union

Figure taken from: https://meltdownattack.com/meltdown.pdf

Spectre: Variant 2

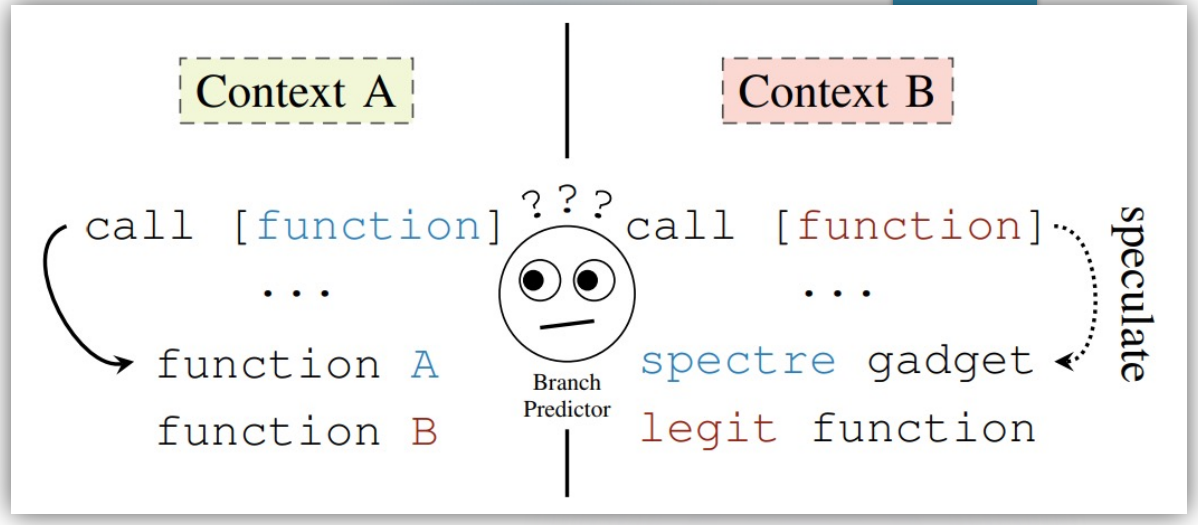


- Phase Two: the Execution
  - Assuming a gadget made up of just two instructions utilizing two registers R1 and R2
  - The secret data k is added to the value of R1 and stored in R2
  - R2 is used to access the memory address in R2
  - The data stored at address R2 is loaded in cache

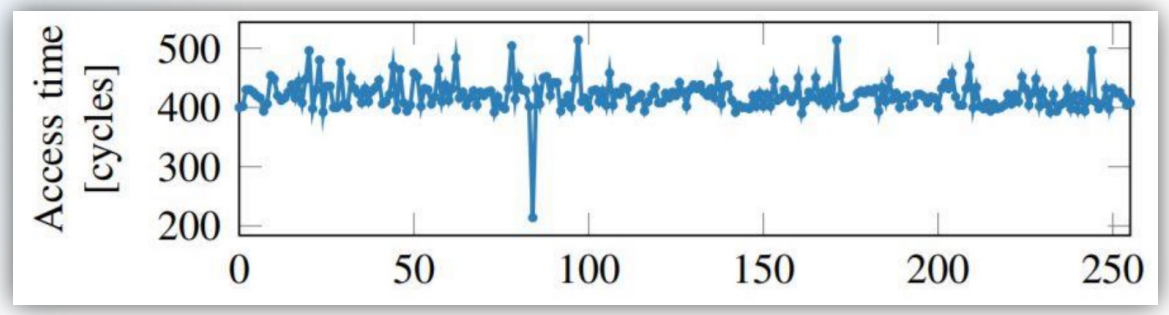
Co-funded by the European Union

Figure taken from: https://meltdownattack.com/meltdown.pdf

Spectre: Variant 2



- Phase Three: the Recovery
  - The content of the cache remains affected by the misprediction even after the correct execution resumes
  - By measuring access times of the cache, it's possible to identify the affected memory location and retrieve the value of the secret k



Co-funded by the European Union

Figures taken from: https://meltdownattack.com/meltdown.pdf

Implementation

- The aim of our project was to implement one attack on a real-world platform. We chose:
  - Spectre: conditional branch misprediction
  - Raspberry Pi 3 model B: ARM Cortex-A53

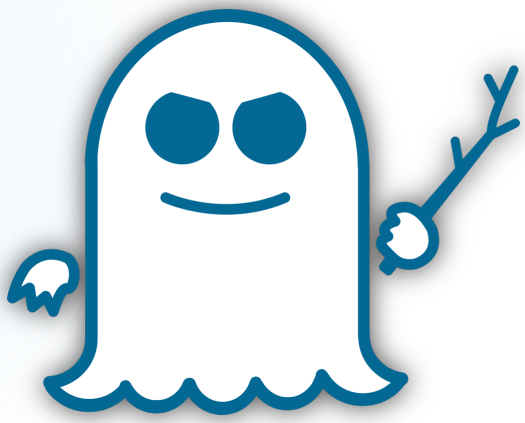


Figure taken from: https://meltdownattack.com/

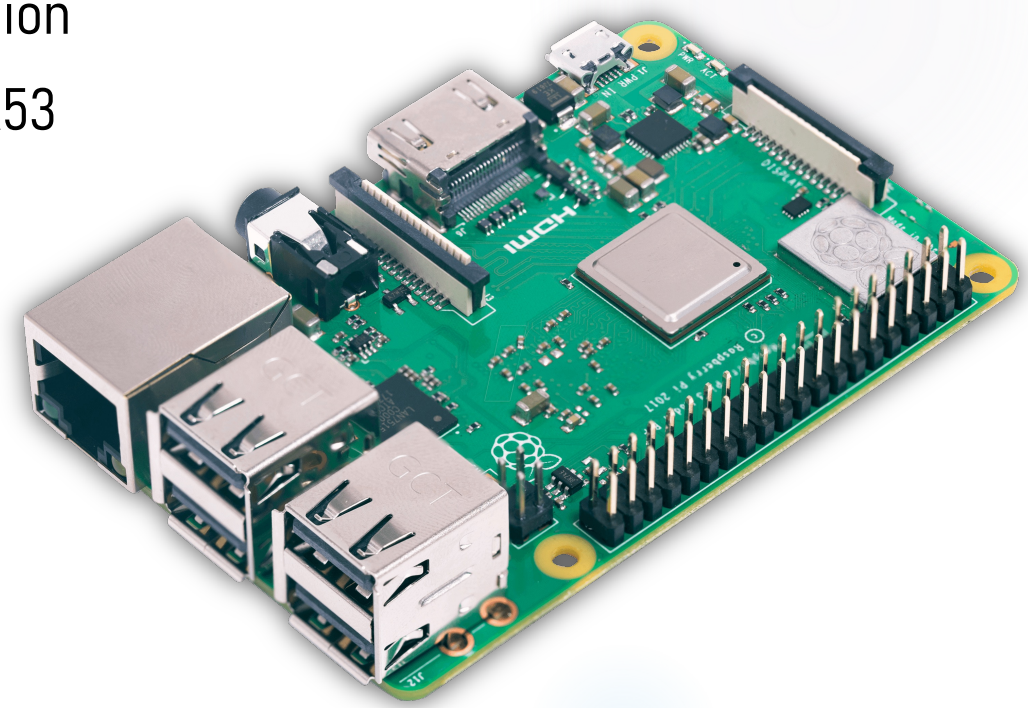


Figure taken from: https://www.pngaaa.com/detail/2839047

Co-funded by the European Union

Inside the Raspberry Pi 3B

- First, let's see in detail the parts of the system that we'll need to interact with during the attack:
  - Cache memory
  - Branch History Registers
  - Pattern History Tables

Co-funded by
the European Union

Inside the Raspberry Pi 3B

- First, let's see in detail the parts of the system that we'll need to interact with during the attack:
  - Cache memory
  - Branch History Registers
  - Pattern History Tables

32KB of L1 cache made up of lines of 64 bytes.
L2 cache is present but disabled on our system.
The mapping is 4-way set associative and exploits a pseudo-random replacement policy within each set

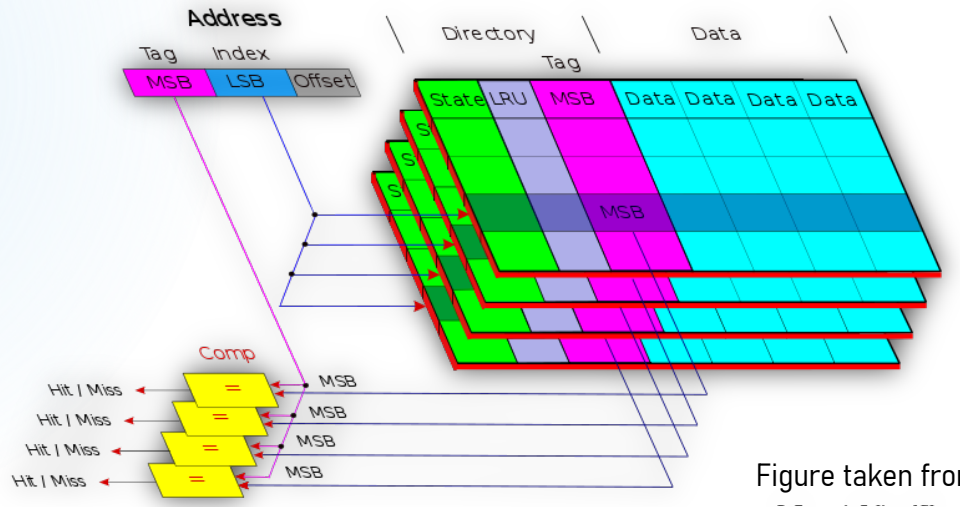


Figure taken from: https://en.m.wikipedia.org/wiki/File:Set_Associative_Cache.svg

Co-funded by the European Union

Inside the Raspberry Pi 3B

- First, let's see in detail the parts of the system that we'll need to interact with during the attack:
  - Cache memory
  - Branch History Registers
  - Pattern History Tables

Abbreviated as BHRs, an array of 3 10-bit shift registers storing the outcomes of the last branches. Each branch operation is mapped to one of the 3 registers by performing the operation: `address mod 3`
When the outcome of a branch is known, the register content is shifted left and a new bit representing to the branch outcome is inserted as the LSb

Co-funded by the European Union

Inside the Raspberry Pi 3B

- First, let's see in detail the parts of the system that we'll need to interact with during the attack:
  - Cache memory
  - Branch History Registers
  - Pattern History Tables

Abbreviated as PHTs, an array of 3 tables of 1024 elements each. Each BHR corresponds to one PHT. The elements of the PHTs are 2-bit saturating counters in which the LSb implements a confidence level (strong/weak) related to the outcome of the branch (taken/untaken) stored in the MSb

Co-funded by the European Union

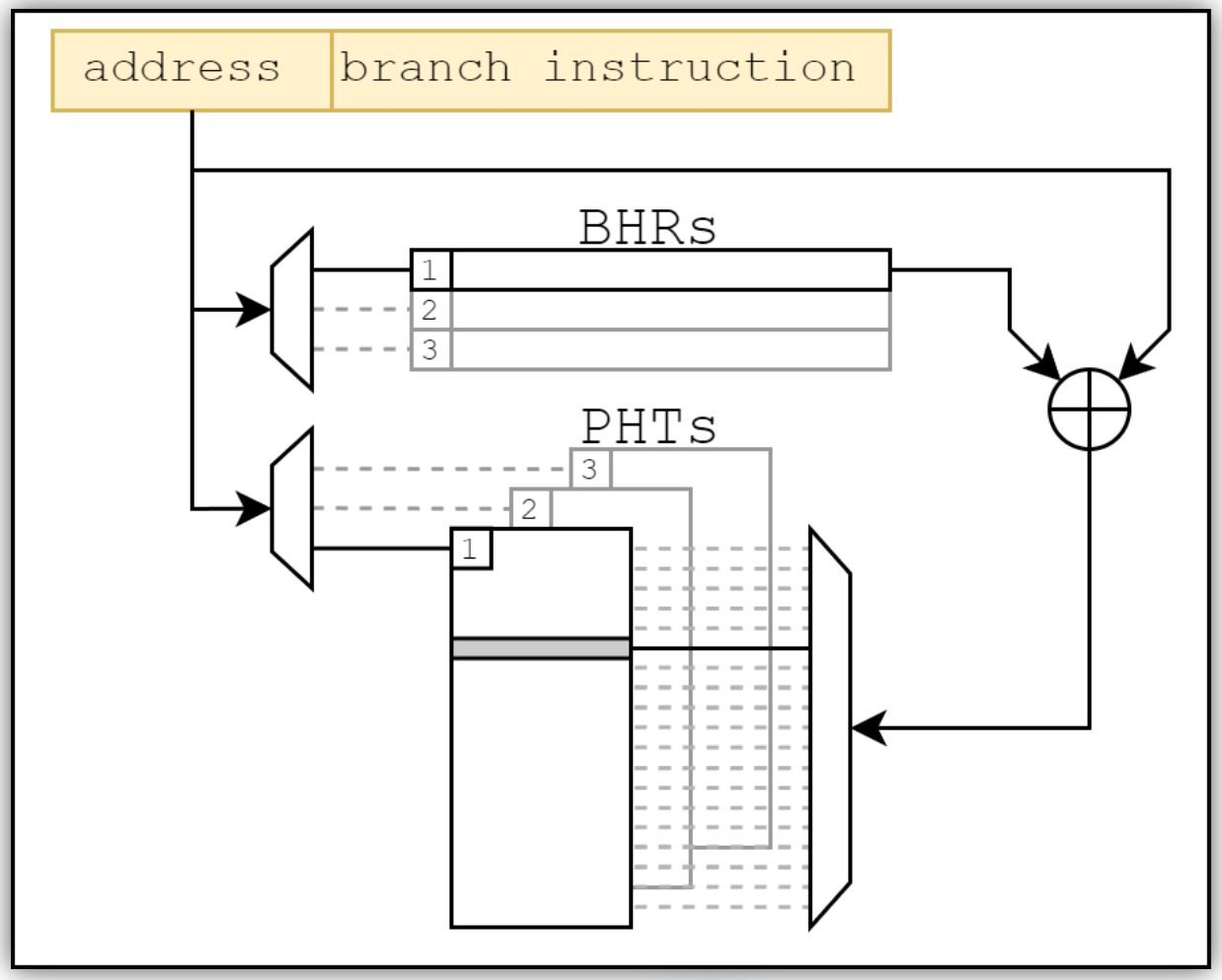
The Attack: Background

- For the Spectre attack to work we must focus on the following characteristics:
  - The address of the branching instruction points to a specific BHR register and a specific PHT table
  - The entry of the chosen PHT is determined by the function $\texttt{branch\_addr} \oplus \texttt{BHR}_{\texttt{branch\_addr}}$ so the BHR value must be forced to a known value
  - The value of the specific PHT entry accessed decides the prediction of the branch, so it must be forced to a known value

Co-funded by the European Union

The Attack: Background

- The dependency of the previous blocks can be visualized in this diagram:
  - The BHR and PHT are selected by the address of the branch instruction (mod 3)
  - The entry in the selected PHT is calculated thanks to the address and BHR



Co-funded by the European Union

The Attack: Setup

- The first step is to set the contents of the BHR and PHT related to the branch address to the needed values to carry out the malicious cache read:
  - The BHR with unknown content, is overwritten to an all-zero value by calling the victim code 10 times. The result is $BHR_{branch\_addr} = 0x00$
  - At this point, the PHT entry can be easily identified: $branch\_addr \oplus BHR_{branch\_addr} = branch\_addr$
  - 3 more calls are needed to migrate the value of the PHT entry from state `11` (strongly taken) to the state `00` (strongly untaken)

Co-funded by the European Union

The Attack: Setup

- The Branch Prediction Unit is now poisoned but that's not enough:
  - The cache is now flushed of the values that would disrupt the branch prediction speculation
  - An additional 10 dummy instructions are called to make sure that the contents of the BHR weren't altered during the cache flush

Co-funded by
the European Union

The Attack: Execution

- At this point the setup phase is complete, and the Spectre attack can be executed:
  - The malicious code reading the secret data is called
  - Thanks to the value of the BHR and PHT, the wrong prediction is made
  - Thanks to the cache flush, the mispredicted branch has enough time to access the secret data
  - The secret data is retrieved and used to modify the cache

Co-funded by
the European Union

The Attack: Recovery

- The attack is almost complete, the attacker can now retrieve the secret data thanks to a timing analysis:
  - The performance timer is started using the function `rdtsc32()`
  - The cache lines are accessed
  - The performance timer is stopped
  - By calculating the time taken to access the cache data, the secret data can be discovered

Co-funded by
the European Union

The Attack: What else?

- Our implementation of the attack works well only on a single-core system. This is because of the lack of multiple layers of the cache.
- We managed to create an additional attack aimed at a multi-core system with an additional constraint:
  - The process must be bound to a specific core for the attack to work correctly

Co-funded by
the European Union

The Attack: What else?

- Unfortunately, the multi-core attack just mentioned is not successful as the single-core described before:
  - The setup, similar to the single-core version, goes as planned
  - The cache read works correctly
  - The speculative access to the data fails
- The exact nature of this issue is unknown and further research and testing are needed

Co-funded by
the European Union

Setup of the lab environment

Some preliminary steps are required for the experience (all the code is provided in a folder):

- Install an OS with memory protection on the Raspberry (Raspbian) or build a custom image using Poky
- Set the CPU clock speed to a fixed value and adjust the memory frequency (1 GHz both, it also works with different settings and dynamic memory settings)
- Install a kernel module to abilitate access to PMU from thread mode (EL0) (add it to `/use/lib/$(uname -r)/kernel/`..., add its name to `/etc/modules/module.conf`, run "`sudo dmprobe`", make sure the module can be loaded manually using modprobe)
- Retrieve the code from the folder that we provide and compile it using the makefile included
- Adjust the alignment of the dummy branches in the code by inspecting the objdump of the project: make sure that you insert a number of NOPs before the dummy branches such that the address of the first branch modulo 3 is equal to the address of the branch in the victim code modulo 3
- To play a bit with the code: try to add some strings after array1 (or change the contents of string s) to see if they are read!

Co-funded by
the European Union

Self evaluation exercises (1)

Try to replicate the original Spectre attack in ARM by replacing the code in the victim with:

```
LDR X5,[X0]
CMP X4,X5
BGE wrong_x
LDRB W1,[X1,X4]
AND X1, X1, X7
LSR X1, X1, X3
LSL X1, X1, #12
LDRB W2,[X2,X1]
```

(in this way we are putting both levels of memory accesses inside the speculation, thus replicating the idea of the original Spectre)

Does the code work? Why or why not?

Co-funded by
the European Union

Self evaluation exercises (2)

Try to change the following characteristics of the code and comment the results obtained:

- Number of mistraining cycles : choose a number greater than 13 and another one smaller, what happens in the two cases?
- Number of alignment NOPs
- Level of optimization for the compiling process
- Try to increase the operations done inside the speculation (for example by adding some unoptimized NOPs): does the code still work or the time is not enough for the read from memory to happen?

Co-funded by
the European Union

Self evaluation exercises (3)

- Try to implement a mechanism able to read multiple bits from the selected bytes. This requires another strategy, different from the one we've explained during the experience (the one able to read only one bit).

Hint: try to randomize the accesses within a single memory page, so that the prefetch algorithm can't keep track of the accessed lines. If possible, try to find a better randomization algorithm than the one presented in the paper.

Co-funded by
the European Union

Self evaluation exercises (4)

- Try to find an attack able to measure the access latency that avoids creating misses. Among the ones that are possible there is the Flush+Flush, introduced as one of the possible attacks for implementations of Spectre.

- Try to implement another attack among the ones described in the Spectre paper, paying attention to which of them are applicable in the Raspberry architecture.

Co-funded by
the European Union

Self evaluation exercises (5)

- Try to change prefetcher settings by modifying `CPUACTLR_EL1` (it must be executed after a reset and before the MMU is enabled, you need to change settings in the boot sequence). Try to change the number of bits read in each iteration of the attack to check the effects on the prefetching.
- Is it possible to increase the number of bits read if we increase the length of the stride that triggers prefecthing?
- What happens if we completely disable prefetching?

Co-funded by
the European Union

Conclusions

- We illustrated the theory behind two powerful hardware-based attacks:
  - Meltdown
  - Spectre
- We successfully implemented a Spectre attack exploiting the conditional branch prediction on a widely used real-world platform:
  - Raspberry Pi 3 model B

Co-funded by
the European Union

THE END

THANK YOU FOR YOUR ATTENTION

Co-funded by
the European Union