Spectre and Meltdown Attack Variants

N V S S Hari Krishna Nama - 170050077
Dileep Kumar Reddy Chagam - 170050080
Tarun Somavarapu - 170050093

Problem Statement/Goals:

In appetite for speed, we lost ____?

Security

Spectre and Meltdown attacks exploit the crucial vulnerabilities in modern processors. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running program. This might include your passwords stored in a password manager or browser, your personal photos, emails, instant messages, and even business-critical documents.

The main goal of the project is to understand this attacks and implementing them. At the moment, there are many variants available to these attacks, we are going to show some of these variants in the demo.

Background

- Speculative Execution
- Out-of-order Execution
- Branch Prediction
- Microarchitectural Side-Channel Attacks
- Return-Oriented Programming

We are dealing with

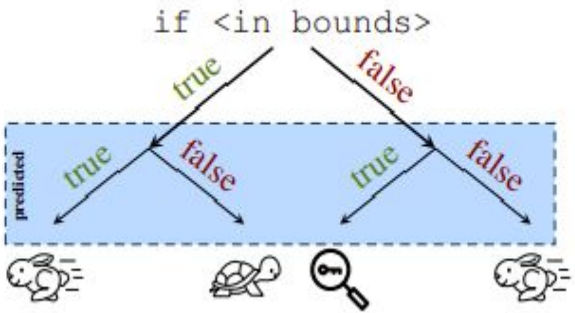
- Spectre variant 1
- Spectre variant 2
- Meltdown

Spectre v1

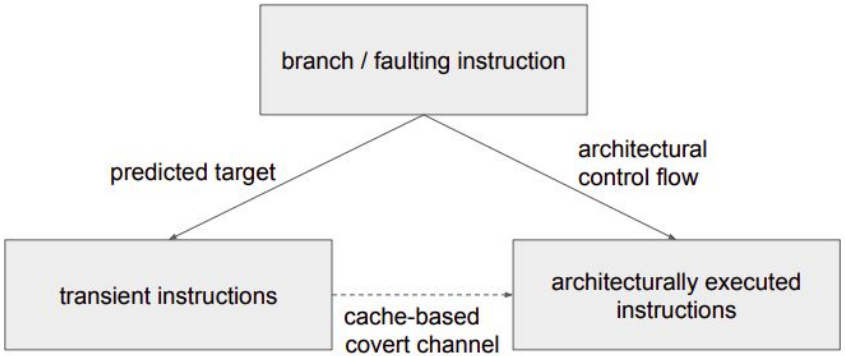
Spectre v1: : Conditional Branch Example

In this variant of Spectre attacks, the attacker mistrains the CPU’s branch predictor into mispredicting the direction of a branch, causing the CPU to temporarily violate program semantics by executing code that would not have been executed otherwise.

Misspeculation



Covert channel out of misspeculation



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

- Execution without speculation is safe
  - CPU will never read array1[x] for any x ≥ array1_size
- Execution with speculation can be exploited
  - mistrain branch predictor to assume ‘if’ is likely true by providing valid or in-bound values of x
  - make array1_size and array2[] uncached; thus cpu can wait or speculate for speed up
- Invokes code with malicious/out-of-bounds x such that array1[x] is a secret
  - NOTE: This read changes the cache state in a way that depends on the value of array1[x]
  - recognizes its error when array1_size arrives, restores its architectural state, and proceeds with ‘if’ false
- Attacker detects cache change (e.g. basic FLUSH+RELOAD or EVICT+RELOAD)

Spectre v2

Spectre v2 :: Poisoning Indirect Branches

- Indirect branches are commonly used in programs across all architectures
- Indirect branches can be poisoned by an attacker and the resulting misprediction of indirect branches can be exploited to read arbitrary memory from another context, e.g., another process
- Here the adversary mistrains the branch predictor with malicious destinations, such that speculative execution continues at a location chosen by the adversary

Simply, in this attack, the attacker does not rely on a vulnerability in the victim code. Instead, the attacker trains the Branch Target Buffer (BTB) to mispredict a branch from an indirect branch instruction to the address of the gadget, resulting in speculative execution of the gadget

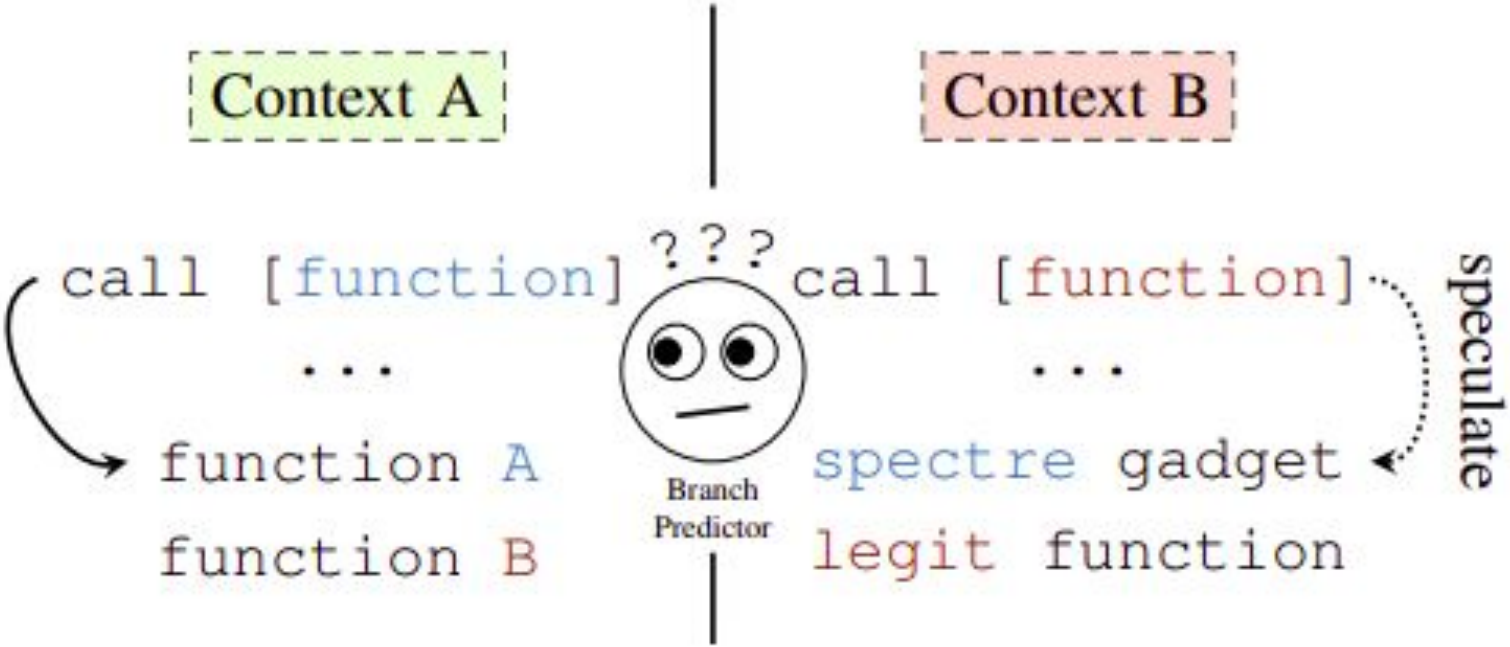


Fig: The branch predictor is (mis-)trained in the attacker-controlled context A. In context B, the branch predictor makes its prediction on the basis of training data from context A, leading to speculative execution at an attacker-chosen address which corresponds to the location of the Spectre gadget in the victim's address space

For a simple example attack, consider an attacker seeking to read a victim's memory, who has control over two registers when an indirect branch occurs. The attacker also needs to locate a "Spectre gadget", i.e., a code fragment whose speculative execution will transfer the victim's sensitive information into a covert channel

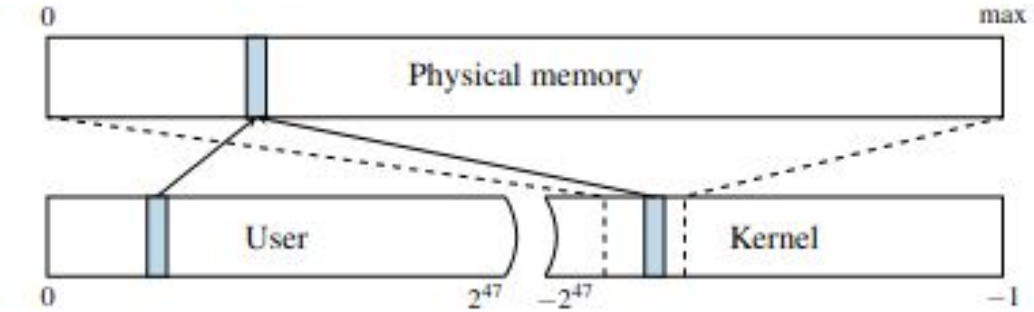
```
int result;
// call *target
__asm volatile("callq *%1\n"
               "mov %%eax, %0\n"
               : "=r" (result)
               : "r" (*target)
               : "rax", "rcx", "rdx",
"rsi", "rdi", "r8", "r9", "r10",
"r11");
return result & junk;
```

```
int gadget(char addr*) {
     return array[*addr * 4096];
}
```

Meltdown

Meltdown:

- Modern operating systems always map the entire kernel into the virtual address space of every user process and also the entire physical memory is directly mapped in the kernel at a certain offset .
- For memory accesses ,CPU checks the User/Supervisor bit to check the **privilege level,** whether it can be accessed



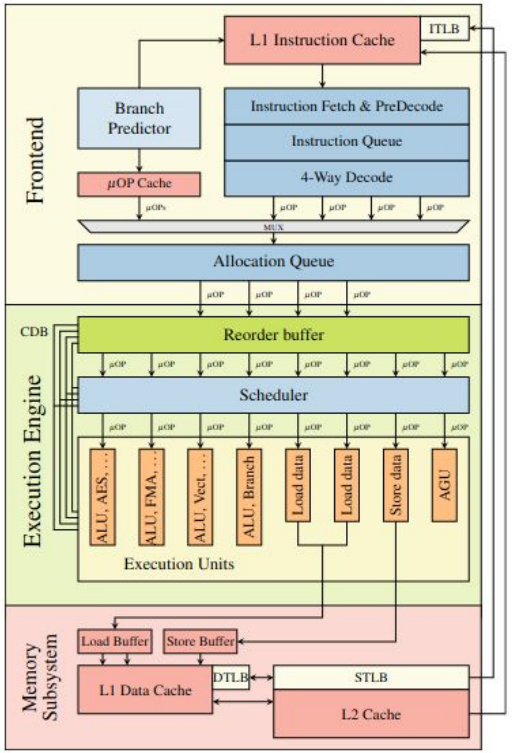
Out-of-Order Execution

Instructions are

- Fetched and decoded in the Frontend, then dispatched to be processed by execution units
- Executed out-of-order as prior instructions might wait until their dependencies are resolved
- Retire in-order
  - any exceptions occurred are checked during retirement

Out-of-Order Execution leave microarchitectural traces

Such instructions are called transient instructions.

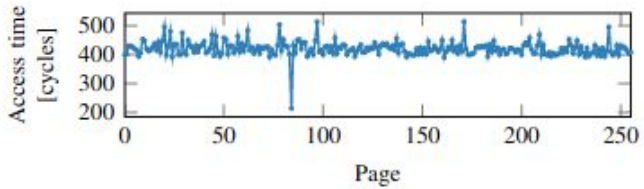


Setting Exploit

- Loading inaccessible addresses triggers an exception that terminates the process
- **Exception handling** / **Exception suppression** to be set up to avoid crashes
- Transient instruction sequence:

```
char data = *(char *) addr ;
user_array[data * 4096] = 0 ;
```

- Leverage techniques from Cache Attacks (Flush + Reload) to deduce the value by checking which part of user_array is in cache



Comparison between Spectre and Meltdown

	SPECTRE	**MELTDOWN**
Architecture	Intel, Apple, ARM, AMD	Intel, Apple
Entry	Must have code execution on the system	Must have code execution on the system
Method	Branch Prediction + Speculative Execution	Trap + Out-of-order execution
Impact	Read contents of memory from other user running program	Read kernel memory from userspace
Action	Software patching for now	Software patching for now (KAISER)

Mitigations

Several countermeasures for Spectre and meltdown attacks have been proposed. Each addresses one or more of the features that these attacks rely upon

- Preventing speculative execution
- Preventing access to secret data while speculatively executing
- Preventing branch poisoning using IBRS, STIBP, and IBPB mitigates spectre variant2
- Limiting data extraction from covert channels
- The KAISER patch for meltdown implements stronger isolation between kernel and userspace. KAISER does not map any kernel memory in the userspace, except for some parts required by the x86 architecture (e.g., interrupt handlers). Thus, there is no valid mapping to either kernel memory or physical memory in the user space

Trials and Tribulations

- Original paper's ATTACK PATTERN is easily predictable by the system's stride prediction. So we had to randomise it further and increase the number of training loops.
- After disabling the KAISER patch on Linux(PTI), we tried dumping memory from another process but in vain. Later tried leaking Kernel Debugging Symbols, which was successful. Now we are working on leaking memory of other processes/kernel modules.
- There are a lot of software patches available for meltdown. Some attacks are no longer working, no matter what we try

Future Works

- Implementing Spectre attack/ Meltdown attack which involves the attacker and the victim running in different processes on same CPU core
- Exploiting the browsers' vulnerability using spectre attack in javascript
- Implementing Spectre/Meltdown attack to fetch secret information from kernel memory like wifi passwords

References

1. Spectre Attacks: Exploiting Speculative Execution, 2019, https://spectreattack.com/spectre.pdf
2. Meltdown: Reading Kernel Memory from User Space, 2018, https://meltdownattack.com/meltdown.pdf
3. Reading privileged memory with a side-channel, Jann Horn, Project Zero, 2018, https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
4. Differences Between Meltdown and Spectre, https://danielmiessler.com/blog/simple-explanation-difference-meltdown-spectre/
5. Spectre variant 2 Proof of Concept, https://github.com/Anton-Cao/spectrev2-poc
6. Meltdown variant Proof of Concept, https://github.com/paboldin/meltdown-exploit

Demo

Thank you