Spectre and Meltdown

Qui Hao (Frank) Yu and Haseeb Choudhary

What is Spectre and Meltdown?

- Two variants of a vulnerability in modern processors
- Could allow attackers access to data presumed to be protected
- Certain PoCs have been shown to be able to:
  - Access memory which otherwise should not be accessible (sometimes between processes)
  - Access memory from the kernel memory space (from an unprivileged program execution)
- Exploit two major designs in modern processors: **caching** and **speculative (/out-of-order) execution**

Who and What do they Affect?

- Affects *some/all* modern processors, servers, mobile phones (Apple SoCs)
- Meltdown
  - Intel, ARM, IBM ...
- Spectre:
  - Intel, AMD, ARM, IBM ...
- Affects all operating systems
  - MacOS, Linux, Windows

Side-Channel Attacks

- Attack based on information gained from the implementation of a computer system.
    - **Caches:** attack which monitors how quickly data accesses take and infer whether or not said data was in the cache
    - **Timing:** attack which monitors time it takes for machine to do various computations
    - **Power-monitoring:** attack which monitors power consumption of hardware on varius computations
    - etc.

In this case...

The side channel comes from monitoring how quickly data can be accessed from the cache.

Data which is accessed quickly => stored in the cache

Data which is accessed slowly => stored in main memory

What do they exploit?

- Speculative (/out-of-order) Execution
- Caching

Speculative Execution

- Goal: we want to use the CPU as much as possible
- A processor will try to compute something even before it is asked to do so
- Ex.

```
bool A = some condition
if A is true:
     then execute function F1()
else:
     execute function F2()
```

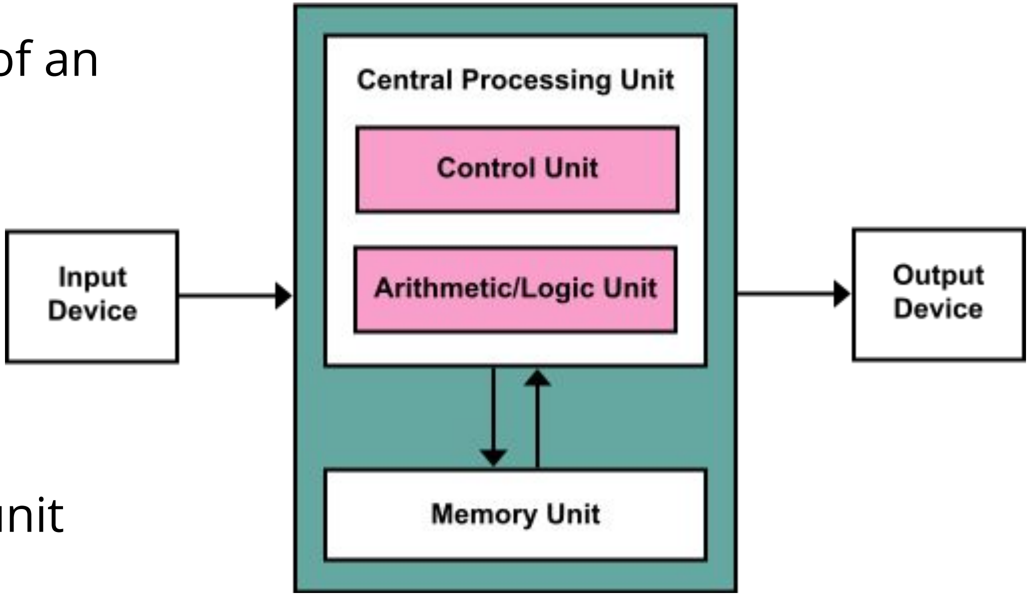
- Processor will do some work ahead of time to compute functions F1 and F2 even though we don’t know which will be executed

Speculative Execution cont.

- Predict what will happen in the future
- If there is *branching* (i.e. if statements, some loop forms), the processor will execute some code which it thinks it will need to do later on
- If a branch is taken, the processor can continue like normal, it guessed correctly
- If a branch is not taken, the work **must** be discarded, it guessed incorrectly (work still stored on cache)

Inside the CPU

If one processing unit is free
the CPU will try to assign a piece of an instruction to use that unit.



E.g. One instruction uses the ALU
while another uses the memory unit

High Level Example

Note: Ignoring the specifics of how a CPU works.

An `add` instruction may be currently using the **ALU** to compute the addition of two values stored in separate registers. The instruction is probably already done using the memory unit...

The CPU can look ahead in the instruction sequence and see if it finds a `load` (from memory) instruction. If it does, it can start executing that instruction because the memory unit is available.

Speculative Execution - A Real Life Example

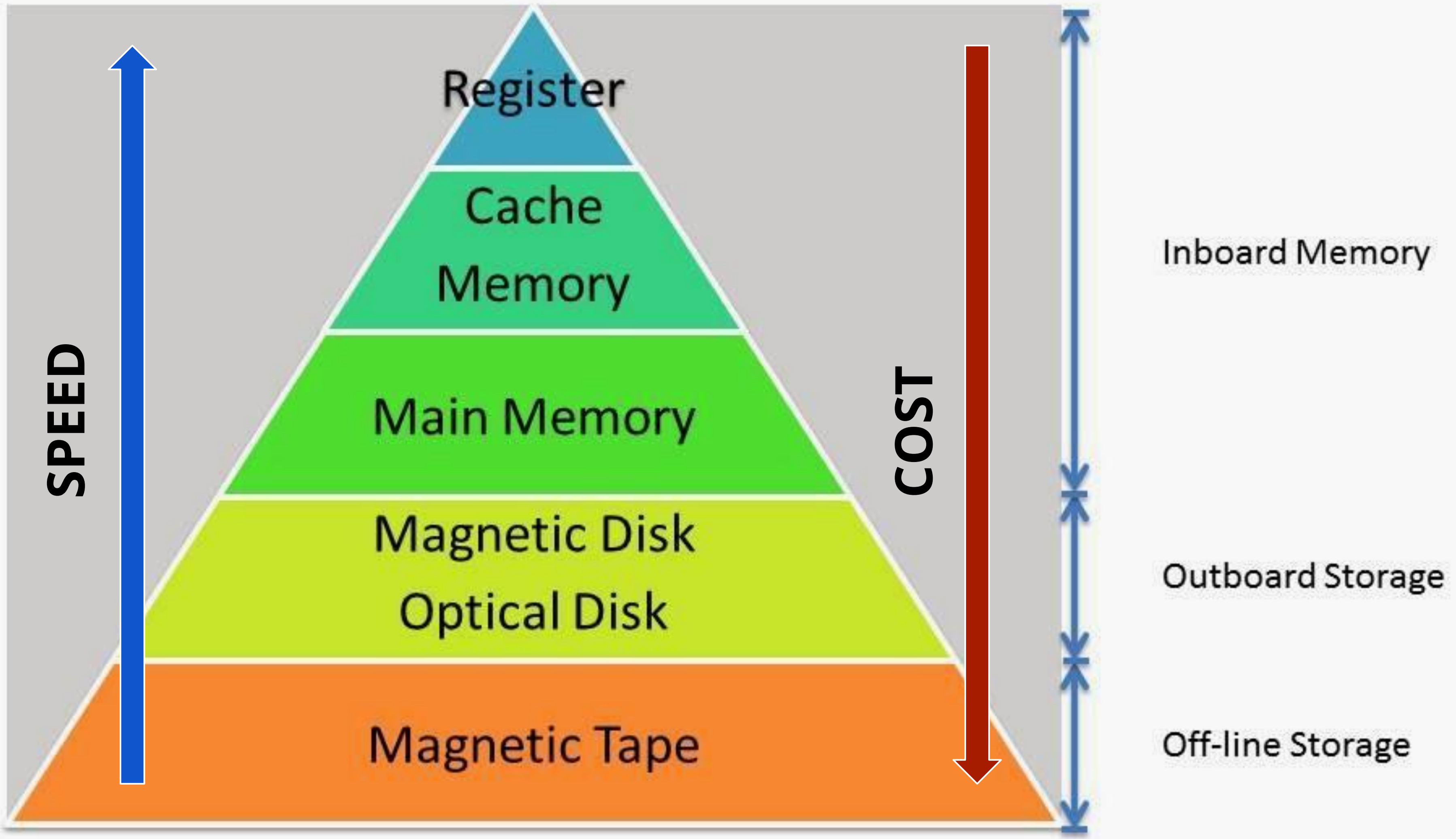
Pusheen.com

OOPS!!

1. Hanging out with a group of friends
2. The group wants to order some pizza
3. A member of the group calls the pizza store to place an order
4. The order that that group member places is based on what the group usually orders
5. The member comes back to join the group and realizes the group wants to order something completely different
6. The member calls back the pizza store to change the order.
   a. The pizza store has to start over and throw away the previous wrong order
   b. Wasted time making the previous order so now they need to start fresh
7. The group has to wait for the store to finish the new order which is what will be delivered to them

Caching

- The CPU requests data from memory which is stored in a cache
- Speeds up memory access
- Temporal locality: something which was accessed recently from memory *might be* accessed again soon
    - Ex. a counter in a loop
- Spatial locality: something which is close to another thing which was accessed recently *might be* accessed soon
    - Ex. elements in an array


Register
Cache Memory
Main Memory
Magnetic Disk
Optical Disk
Magnetic Tape
SPEED
COST
Inboard Memory
Outboard Storage
Off-line Storage

Attacker - Exploiting Cache

- Flush + Reload
  - Flush any access of memory for data you control from the cache (by *clflush*)
  - Lets malicious (or user program) run and access memory you control with secret
  - Try reloading elements from the controlled memory and see how quickly they are accessed
- Evict + Reload (mostly used if *cflush* is unavailable)
  - Evicting memory access of data you control by loading other (possibly random) data into the cache
  - Due to limited size of cache $\Longrightarrow$ evict the specific cache line
  - Let victim program run and access memory using secret, reload data and measure access time

Mitigation

- Funny: https://web.archive.org/web/20180104032628/https://www.kb.cert.org/vuls/id/584653
- Best solution is to redesign modern CPUs

Meltdown

MELTDOWN

Impact

- Modern processors
  - Intel (mostly!)
  - IBM and ARM (minimally)
- Cloud providers
  - Specifically those that rely on Intel CPUs
- Containers such as Docker, OpenVZ, and LXC

What is Meltdown?

- An unprivileged user program can read kernel memory
  - Use of out-of-order instruction execution and memory-to-cache access in processor
  - Abuse privilege check of memory access after data has been brought from memory

- One major variant
  - Rogue data cache load (CVE-2017-5754)

Recall

- The operating system provides a user process with the illusion that they have a large (e.g. infinite) amount of memory to use
  - In reality the actual physical memory is limited
- The kernel address space is mapped to the user address space for efficiency and performance (and possibly more reasons)
  - E.g. if a system call happens, context switching into the kernel address space will just take too long otherwise
- Protection with hardware protection bit which usually traps to the operating system

Meltdown - One Major Variant

- Modern CPUs enforce a privilege check of a program accessing kernel memory
    - This privilege check sometimes occurs too late during speculative execution, i.e. once the data has already been read
- The CPUs knows that this occurs so anything unprivileged which was executed will be forgotten and an exception will be raised (usually SIGSEGV)
- So what’s the problem? Recall: memory accessed recently is still stored in cache

Meltdown - Walkthrough

- Allocate an array with some size = 256 * cache line size (e.g. 4K, 8K)
    - Large size ensures that only specific indices from the array will be cached
- Read from kernel memory at the index in the array
    - E.g. array [read(kernel_memory)]
    - This will store array [read (kernel_memory)] in the cache
- The privilege check occurs once the read instruction is thrown out (i.e. has been fully executed and the CPU is done with it)
- If we can store the data from kernel memory before the privilege check is performed we can read data from kernel memory through a “covert channel”

Attacker controls an array...

```
create $array[256 * 4096]
```

256 is key to distinguish the character stored in kernel memory

4096 is a variable value which represents the cache line size. The cache line size stores data accessed from memory (think: temporal and spatial locality). We want this to be as big as the cache line so that other possible kernel data values are not brought into the cache.

The attacker makes an access to kernel memory…

```
access $kernelMemory, $register
```

This reads data from address *$kernelMemory* and stores the data in *$register*

The attacker preps the value to be stored in the array…

```
shiftleft $register, 0x0C
```

Shifts the value stored in *$register* by 12… value `x` 4096 (2^12)

Attacker stores the value into their controlled array…

```
store $array, read[ $register + $array ]
```

This will store a value into the *$array* at index *$register*

Meltdown - One Major Variant (Example)

Pseudo-assembly:

```
     create $array[256 * 4096]          /* done beforehand  */

 1.  access $kernelMemory, $register
 2.  shiftleft $register, 0x0C
 3.  store $array, read[ $register + $array ]
```

This instruction sequence is executed out-of-order, the CPU will set up and execute pieces of instructions 1-3, until the read from kernel memory is done. There is a race condition between instructions 2 and 3 and the CPU raising an exception for the unprivileged access to kernel memory.

Now the attacker can go through their array iterating over values *1*4096, 2*4096,, 3*4096 … etc.*

The read of the array which is the fastest corresponds to the index into the array which was cached.

The data read from kernel memory can be extracted by taking the index of the fast array access and breaking it up as: *k * 4096* where *k* corresponds to the character in kernel memory.

Exception Handling vs Exception Suppression

We can improve the rate of success either by handling or suppressing the exception raised by the program once the access to kernel memory is done.

Exception Handling

Exception Handling: catch the exception when it occurs (e.g. fork application => parent process will not be affected by exception, install your own signal handler etc.)

Exception Suppression

Exception Suppression: prevent the exception from being raised in the first place (e.g. place attacker code after a branch and force CPU to speculatively execute code, if the CPU guessed wrong it will “roll back”* and the exception will not be raised)

*roll back: erase things such as the instruction execution in registers, CPU pipeline etc.

Mitigation

- Change how CPU handles privileged memory access
- CPU can improve privilege check for memory accesses *before* an instruction is executed (some draw back in performance?)
- Linux kernel: KAISER/KPTI [with KASLR]
  - KAISER: Limits how much of the kernel space is mapped to a processes address space
  - KASLR: randomizes the location of the kernel address space on boot up (can be brute forced though)
  - Variations also implemented in Windows and macOS (also: iOS and tvOS)

Proof of Concept

https://github.com/IAIK/meltdown

Linux Kernel source code mitigation for Meltdown:

```
/* Assume for now that ALL x86 CPUs are insecure */
setup_force_cpu_bug(X86_BUG_CPU_INSECURE);
if (c->x86_vendor != X86_VENDOR_AMD)
        setup_force_cpu_bug(X86_BUG_CPU_INSECURE);
```

Source: https://lkml.org/lkml/2017/12/27/2

Spectre

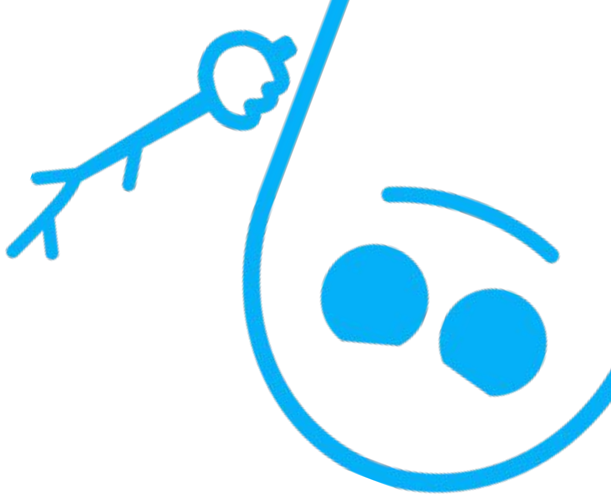
Impact

- Almost every computer system
  - Intel
  - AMD
  - ARM-based
  - IBM-processors
  - etc.
- Browsers
  - Chrome
- Cloud Providers
  - Stronger impact than Meltdown

What is Spectre?

- Reading data from a user processes address space by...
  - Tricking the CPU into speculatively executing instructions that would otherwise not be executed
- Result: leaks the information (from cache) via a side channel to the adversary
- Two major variants:
  - Bounds check bypass, Spectre-V1 (CVE-2017-5753)
  - Branch Target Injection, Spectre-V2 (CVE-2017-5715)

Major Difference to Meltdown



- Meltdown exploit:
  - Out-of-order instruction execution to read kernel data
- Spectre exploit:
  - Speculative instruction execution through indirect and conditional branching to read user process data (can be applied to kernel memory as well)

Variant 1 - Exploiting Conditional Branches

- Conditional Branch Prediction (refer to speculative execution)

- Train the CPU’s predictor into mispredicting the direction of a branch
  - Give branch predictor a certain amount of good values
  - Then give an evil value
  - Predictor will execute the code before the condition has been checked

Variant 1 - Exploiting Conditional Branches cont.

- The result will be discard, but the memory access will remain in cache.

- Check the **memory latency** for accessing specific data and analyze the results
  - Accessing cache is way more faster than accessing main memory...

Variant 1 - Exploiting Conditional Branches cont.

- For example:

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

- x = (address of a secret byte to read) - (base address of array1)

Variant 1 - Real Life Example

- Background: you're working at a company and you want to know if your boss will be away for a specific week

Variant 1 - Real Life Example cont.

1. You prep by calling the administration team every week and asking them to confirm your contact information

2. The week before you want to know whether or not your boss is away you call and ask the administration team to confirm your contact information **iff the boss will be away the next week**

Variant 1 - Real Life Example cont.

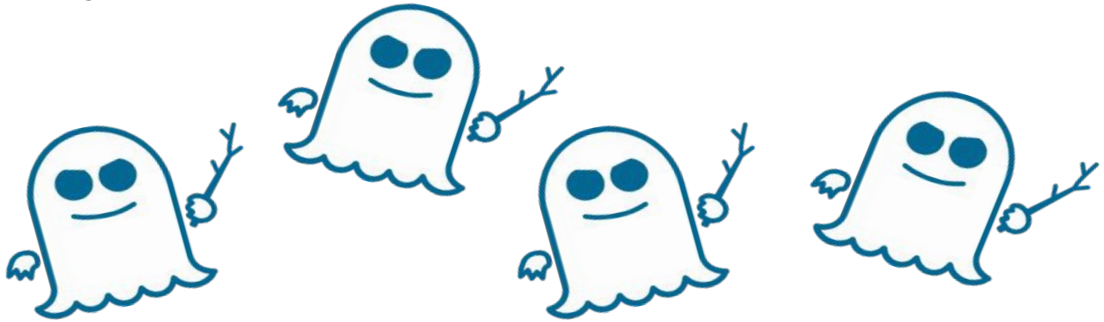
3. The team loads up the boss’ schedule and your information based on the boss’ presence next week. Only after they are done do they remember the boss told them not to tell anyone about his schedule.

4. They tell you “We can’t tell you that information” and you respond with “Okay but can you tell me if my phone number is still listed as ###-#####”

Variant 1 - Real Life Example cont.

One of two things can happen...

- If they respond quickly [with a “yes”] then the boss **is highly likely** (almost certainly) going to be away next week
- If they take some time to respond then the boss **is highly likely** (almost certainly) **not** going to be away next week



Variant 2 - Exploiting Indirect Branches

- **Gadget**: a machine code snippet found in code of victim

- **Indirect Branch:** jumping to code at some memory location
  - e.g. `jmp [eax]` => jump to instruction stored at memory address in register `eax`

- Attacker chooses a **gadget** from the victim's address space and then forces the CPU to speculatively execute the **gadget**
  - Not reliant on the vulnerability of victims code.
  - Attacker has to find the virtual address of gadget

Variant 2 - Background

Exploiting Branch Target Buffer (BTB) ...

Branch Target Buffer (BTB)

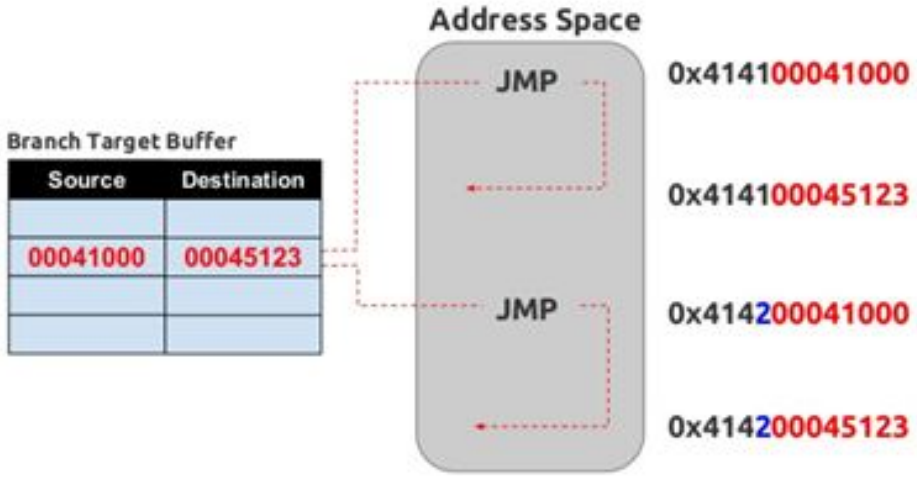
- Unit inside the CPU which stores a mapping of source addresses of recently executed branch instructions to destination addresses
- Used for processors to predict future code addresses without even decoding the branch instruction
    - How it improves performance? - Speculative Execution
- Only the **31 least significant bits** of the branch address are used to index the BTB (the major reason for why spectre works)

Branch Target Buffer (BTB)

- Why is this useful?
- Allows the CPU to speculatively execute code at *predicted* indirect branch target without actually having decoded the branch instructions
- Can improve performance if indirect branch prediction is correct by allowing processor to execute ahead
    - Likewise can harm performance if prediction is incorrect.

Variant 2 - Exploiting Indirect Branches cont.

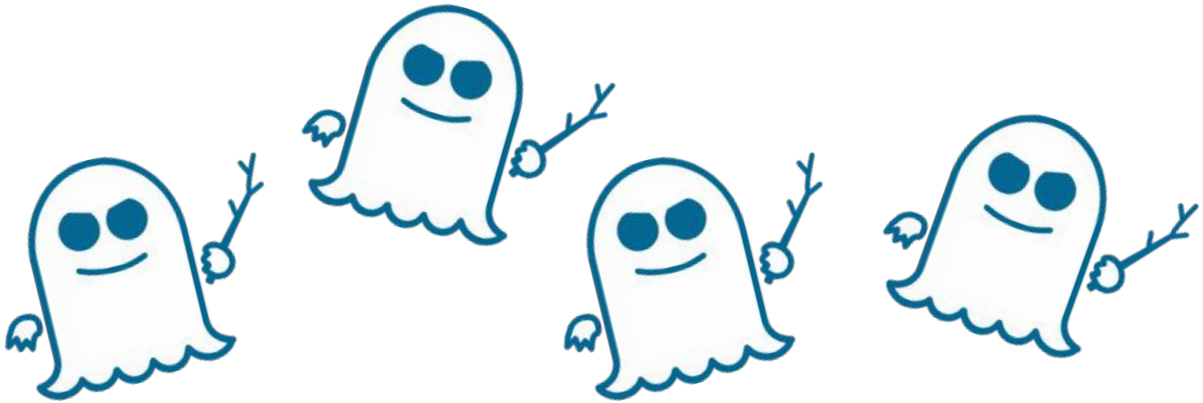
- Attacker trains the Branch Target Buffer (BTB) to mispredict a branch from an indirect branch instruction to the address of the gadget.


Branch Target Buffer

Source	Destination
00041000	00045123

Variant 2 - Exploiting Indirect Branches cont.

- Gadget was run by speculative execution because of branch misprediction
  - Some states in the CPU may be wiped clean of the operations ever having been executed
  - However, the results will still be in the cache



Advanced...

- Mistrain Branches History Buffer (BHB)
- Mistrain return instruction - Return Stack Buffer
- Contention on arithmetic units
- V3c Composition Attack
- Javascript is also vulnerable to the Spectre
    - Enable `#shared-array-buffer` in `chrome:///flags`
    - https://xlab.tencent.com/special/spectre/spectre_check.html
    - https://github.com/cgvwzq/spectre

Mitigation Options

- Preventing Speculative Execution:
  - Ensure control flow leads the instruction
  - Software using serialization or speculation blocking
  - Causing a significant degradation in the performance
- Preventing Access to Secret Data (more for JIT compiler)
  - Chrome: each website per process
- Limiting Data Extraction from Covert Channels
- Preventing Branch Poisoning

Mitigation Options cont

- Preventing Data from Entering Covert Channels
  - Future processors (no such design is currently available) ...
- KAISER/KPTI does not help for Mitigation

Proof of Concept

https://www.exploit-db.com/exploits/43427

https://www.youtube.com/watch?v=0kHFvUcQsWQ

References

https://meltdownattack.com/meltdown.pdf

https://spectreattack.com/spectre.pdf

https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

http://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/spectre_meltdown/index.html

https://www.kb.cert.org/vuls/id/584653/

https://www.slideshare.net/GavinGuo3/spectrev12-fv22fv4-vs-meltdownv3-102527086

https://github.com/IAIK/meltdown

https://www.csoonline.com/article/3247868/vulnerabilities/spectre-and-meltdown-explained-what-they-are-how-they-work-whats-at-risk.html

https://www.reddit.com/r/sysadmin/comments/7ot0ke/genius_explanation_of_meltdownspectre_malware/dscjd68/

http://www.cis.syr.edu/~wedu/seed/Labs_16.04/System/Spectre_Attack/Spectre_Attack.pdf

http://www.cis.syr.edu/~wedu/seed/Labs_16.04/System/Meltdown_Attack/Meltdown_Attack.pdf