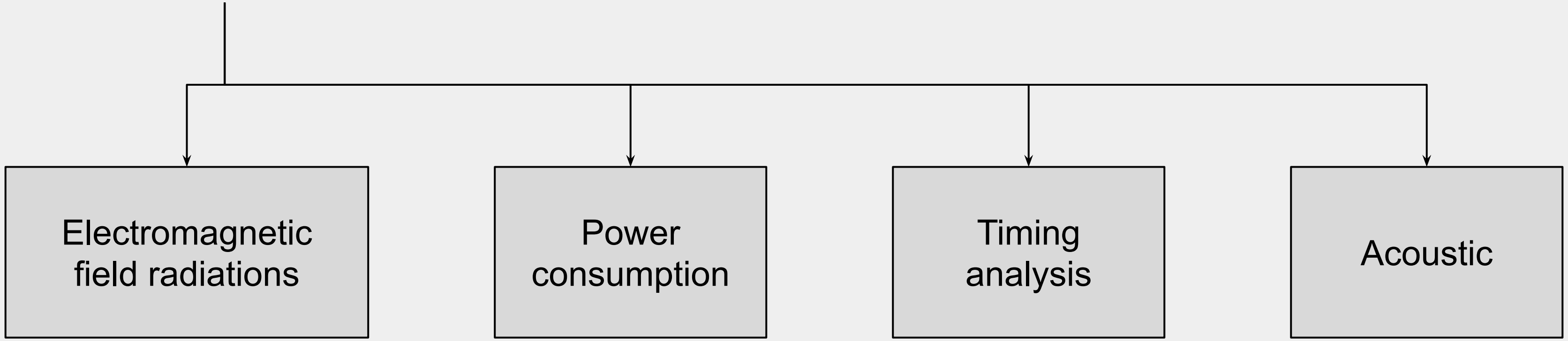
Spectre Attacks: Exploiting Speculative Execution

P. Kocher et al.

*2019 IEEE Symposium on Security and Privacy***.**

Presented by Vishakh Suresh Babu

Side channel attacks



TEMPEST

(**T**ransient **E**lectro**m**agnetic **P**ulse **E**manation **ST**andard)

Differential power analysis

Meltdown

Spectre

SonarSnoop

Why speculative execution?

- What if OOO execution reaches a branch whose direction depends on a value involved in a RAW dependency with a preceding instruction that has not completed yet?

```
if (A = 0)
   A = B
else
   A += 2
```

A is uncached

→

A is at 0(R2)
B is at 0(R3)

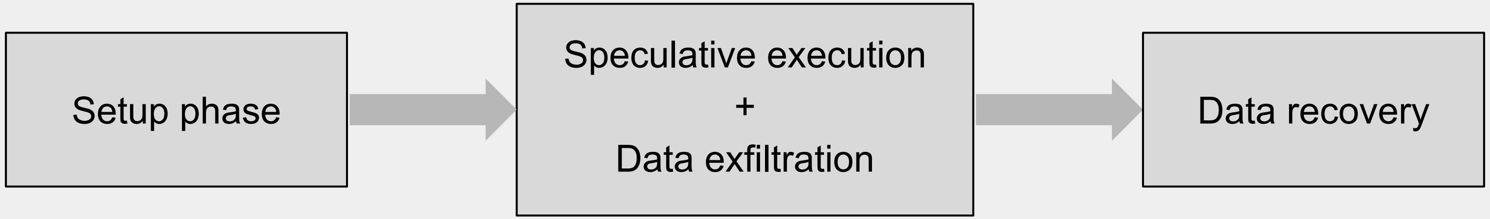
```
      LD   R1,0(R2)
           ↓
      BNEZ R1,L1
      LD   R1,0(R3)
      J    L2
L1:
      ADDI R1,R1,#2
L2:
      SD   R1,0(R2)
```

Why speculative execution? (cont’d)

- *Naive option* : CPU idles waiting for results
- *Better* : Guess execution path and proceed speculatively
  - Save a checkpoint of the register states
  - Predict branch direction -- Branch predictor
  - Predict target address -- Branch Target Buffer (BTB)
    - Don’t even have to wait until ID (instruction decode) stage
  - When value comes in from DRAM, check if guess was correct
    - Commit speculative work/ Discard faulty work

Spectre attacks

- Proceeds in 3 phases :



Locate sequence of instructions capable of acting as the covert channel transmitter

(Mis)train CPU into speculatively executing them

Speculatively execute the instructions that leak information through some side channels

Retrieve the data leaked over the covert channel

Exploiting conditional branches

CVE-2017-5753

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

Attack scenario :

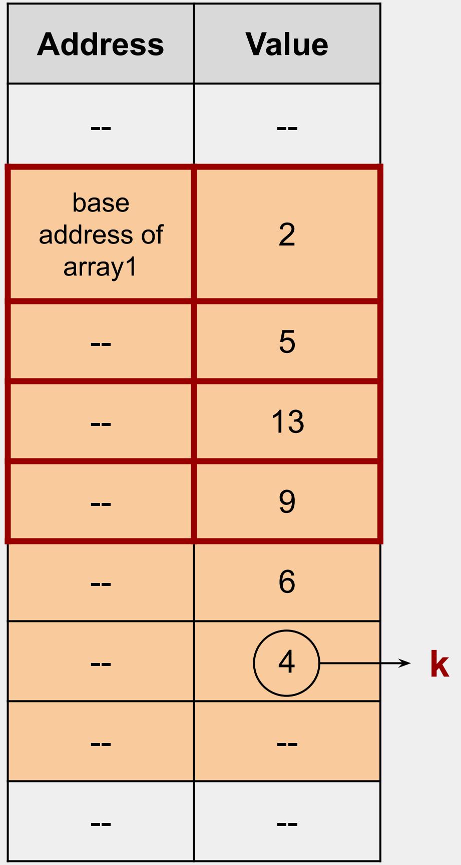
- Adversary controls unsigned integer `x`
- Bounds check to prevent access to sensitive memory outside `array1`
- Branch predictor (mis)trained to predict TAKEN
- `array1_size` and `array2` uncached
- `array1[x]` cached

Exploiting conditional branches (cont'd)

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

Attacker calls the snippet with `x > array1_size`:

- Cache miss on `array1_size` access
- Speculative execution while waiting for `array1_size`
- Predict that branch is taken
- Read what's at `(base addr. of array1 + x)`
- Read returns secret byte k (fast)



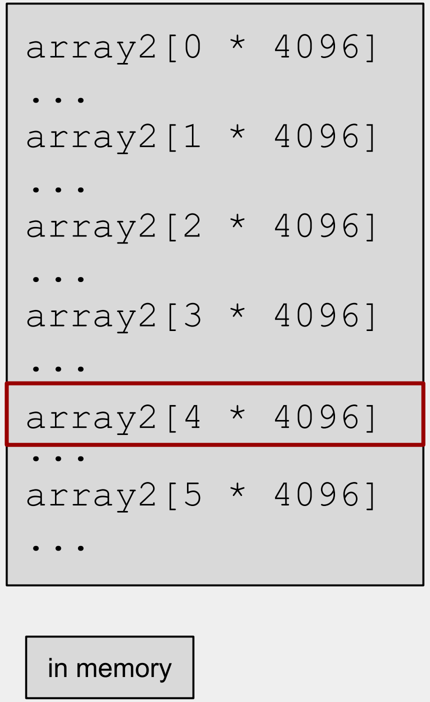
Address	**Value**
--	--
base address of array1	2
--	5
--	13
--	9
--	6
--	$\boxed{4} \rightarrow k$
--	--
--	--

Exploiting conditional branches (cont'd)

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

Attack cont'd :

- `array2[`**`4`**` * 4096]` is not in cache
- Issue a read for memory at address `(base addr. of array2 + `**`4`**` * 4096)`
- `array1_size` fetched & branch direction clear (false)
  - Discard work done in speculative mode
  - Changes to the cache survive state reversion



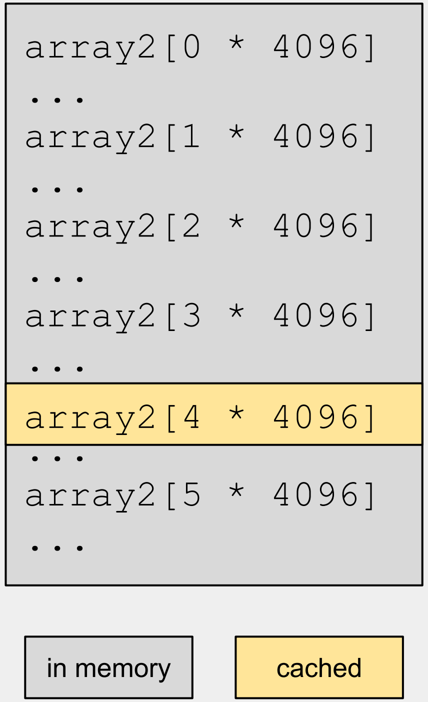
Exploiting conditional branches (cont'd)

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

Recovering k :

- Contents of array2 do not matter ⇒ Only status in cache matters!
- If adversary has access to `array2`, time reads to `array2[i * 4096]`
- If read to `array2[j * 4096]` is faster than other i's

  ⇒ j = **k**, revealing the secret byte



Discussion

In the conditional branch example below, what happens if the value of `x` is chosen such that `array1[x]` causes `array1[x] * 4096` to be outside of the range of array2?

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

Discussion (cont’d)

In the conditional branch example below, what happens if the value of `x` is chosen such that `array1[x]` causes `array1[x] * 4096` to be outside of the range of array2?

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

- `array2` needs to be fetched from the DRAM
- fetch the contents at address

  ```
  base address of array2 + array1[x] * 4096
  ```

Poisoning indirect branches

CVE-2017-5715

- (Mis)train branch target buffer with malicious destinations
- Speculative execution resumes at the location fixed by the adversary
- Locate spectre gadgets to transfer victim’s information
- Example gadget :

```
xor     %ebx,%ebx
add     %eax,%ebx
mov     %ebx,-0x8(%ebp)
```

- Unlike ROP, these gadgets don’t have to terminate in a `ret`

Discussion

I wonder if this attack can be used in conjunction with the work from “Innocent Flesh on the Bone”. Specifically, if the gadgets the author found there can be exploited to perform arbitrary computation. Of course, it depends on which attack is easier to set up: a stack attack or a speculative execution attack. Combining the two works makes most sense if the latter is easier.

Discussion (cont’d)

How can we identify code vulnerable to Spectre attack?

Discussion (cont'd)

How can we identify code vulnerable to Spectre attack?

- Pretty much all non-sequential code
- Run on processors that use speculative execution

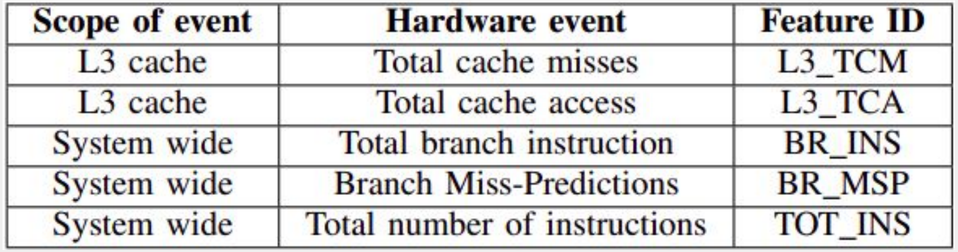
Discussion (cont’d)

How can we detect if such an attack is happening at any given moment?

Discussion (cont’d)

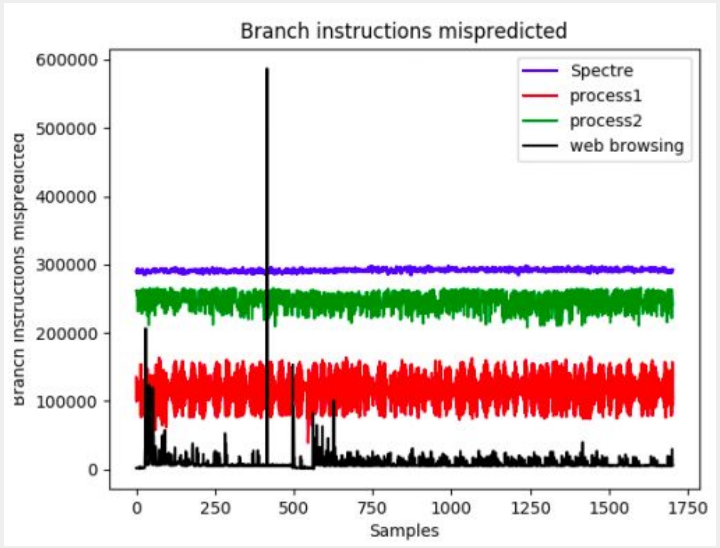
How can we detect if such an attack is happening at any given moment?

- Hardware counters and software events to monitor system activity



Scope of event	Hardware event	Feature ID
L3 cache	Total cache misses	L3_TCM
L3 cache	Total cache access	L3_TCA
System wide	Total branch instruction	BR_INS
System wide	Branch Miss-Predictions	BR_MSP
System wide	Total number of instructions	TOT_INS

- Use various ML models (LogR, SVM, CNN) to analyze these events



Source → Reference [3]

Discussion (cont’d)

Spectre attacks make use of side channels to leak sensitive information. What are some other side channels that can be used?

Variations

- *Evict + Time attack*

  ```
  if (condition) // (mis)predicted taken
     read array1[R1]
  read [R2]
  ```

  - If `array1[R1]` is a cache hit
    - Nothing goes onto the memory bus
    - `read[R2]` starts quickly
  - Selectively evict words from the cache (by causing contention)

  ⇒ analyze the timing of operations

Variations (cont'd)

- *Contention on the register file*
  - Processor sets aside a finite number of registers for saving checkpoints
  - Checkpoints saved if branch is predicted taken

    ```
    if (R != 8)// R = 8 -- (mis)predicted taken
       // block 1
    // block 2
    ```

  - Reduction in speculative execution

    ⇒ Shortage of space on the register file

    ⇒ Leaks info about the variables involved in the condition

Mitigation

- *Turn off speculative execution*
  - Huge performance impact
  - Use an `lfence` instruction
    - Instructions after the barrier need to wait for previous loads to finish
    - Insert before both T & F branches

      ⇒ Disables speculative execution

```
    LD   R1,0(R2)
    lfence
    BNEZ R1,L1
    LD   R1,0(R3)
    J    L2
L1:
    ADDI R1,R1,#4
L2:
    SD   R1,0(R2)
```

Mitigation (cont’d)

- *Preventing speculation execution on potentially sensitive execution paths*
    - Statically analyze code for security-critical code paths
    - Block speculation on such paths
    - Issues?
        - Need restrictions on non-security-critical code in the same process
        - Modern compilers are not capable of doing this automatically

Mitigation (cont’d)

- *Restricting access to secret data*
    - Replacing bounds checking with index masking
        - access `array1[index AND bitmask]`
        - `index` = 19 (10011) AND `bitmask` = 10 (1010) ⇒ 2 (00010)
    - XORing pointers with pseudo-random poison values
        - Adversary cannot use the poisoned pointer directly

Mitigation (cont’d)

- *Preventing data from entering covert channels*
  - Track data fetched while in speculative mode
  - Prevent use in subsequent operations until branch direction is resolved

Mitigation (cont'd)

- *Retpolines*
  - Protection against branch target injection
  - Replace indirect branch with ret to an endless loop
  - Once branch target is known ⇒ push it onto stack & return to it

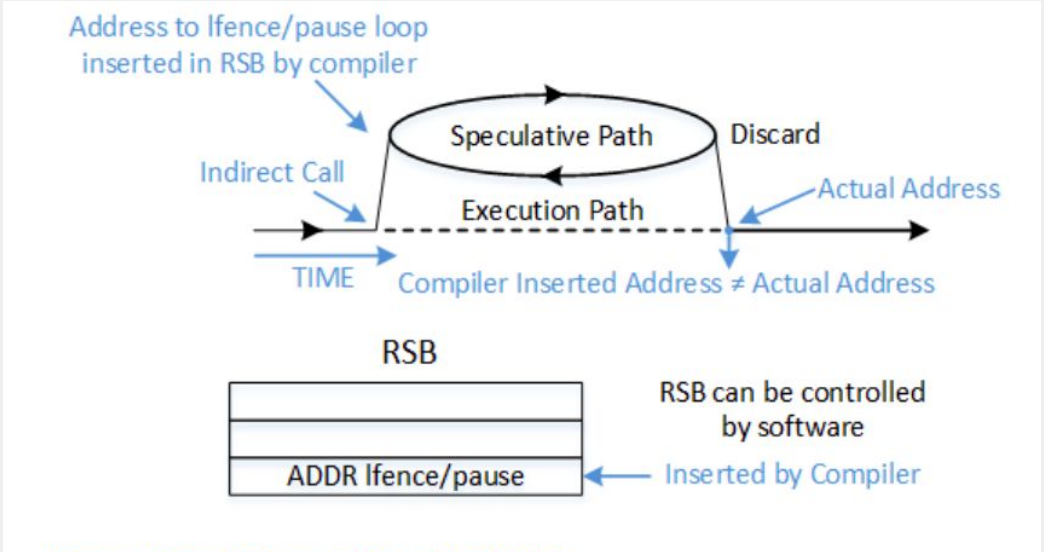

Address to lfence/pause loop inserted in RSB by compiler
Speculative Path
Discard
Indirect Call
Actual Address
Execution Path
TIME
Compiler Inserted Address ≠ Actual Address
RSB
RSB can be controlled by software
ADDR lfence/pause
Inserted by Compiler

Figure 2: Speculative execution with retpoline

Image source → Retpoline: A Branch Target Injection Mitigation White Paper

Discussion

Would a suitable solution to Variant 1 (Exploiting Unconditional Branches) be to simply clear the cache during the CPUs reset phase? What are the challenges with this? This simple addition could potentially prevent an attacker from reading secret information from the speculative attack.

Discussion (cont’d)

Would a suitable solution to Variant 1 (Exploiting Unconditional Branches) be to simply clear the cache during the CPUs reset phase? What are the challenges with this? This simple addition could potentially prevent an attacker from reading secret information from the speculative attack.

- Clearing all the cache contents may be problematic
  - What about contents that were brought to cache by a benign code sequence?

- How about buffering speculative transactions in a cache side buffer?
  - Flush out only the buffer if needed

Discussion (cont’d)

Could there be a self-aware, “speculative” defense system to combat speculative execution exploits? For example if an index or multi-register function is used within a speculative block, can the cpu just delay processing it until the execution reaches that?

Discussion (cont’d)

Could there be a self-aware, “speculative” defense system to combat speculative execution exploits? For example if an index or multi-register function is used within a speculative block, can the cpu just delay processing it until the execution reaches that?

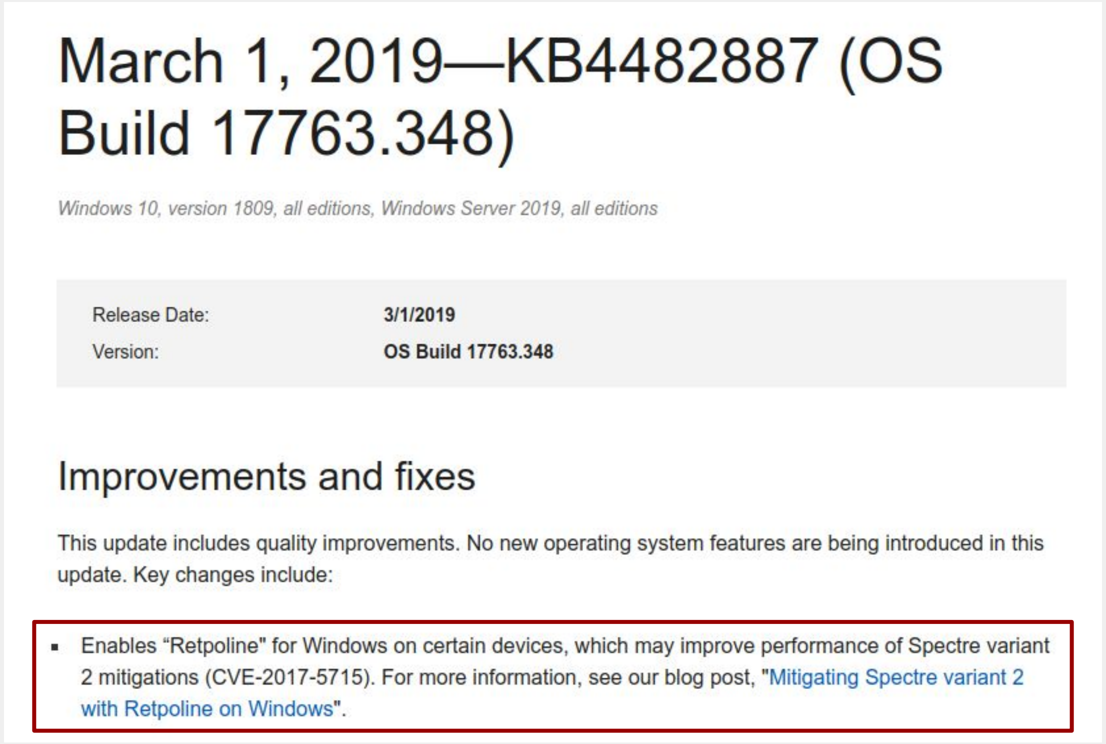
- Intel and ARM processors use lfence instructions
- Can be used to force a wait until branch direction is clear

Discussion (cont'd)

Which defenses proposed by this paper will continue to be implemented in future systems?

Discussion (cont'd)

Retpoline support for Windows



March 1, 2019—KB4482887 (OS Build 17763.348)

Windows 10, version 1809, all editions, Windows Server 2019, all editions

Release Date:	**3/1/2019**
Version:	**OS Build 17763.348**

Improvements and fixes

This update includes quality improvements. No new operating system features are being introduced in this update. Key changes include:

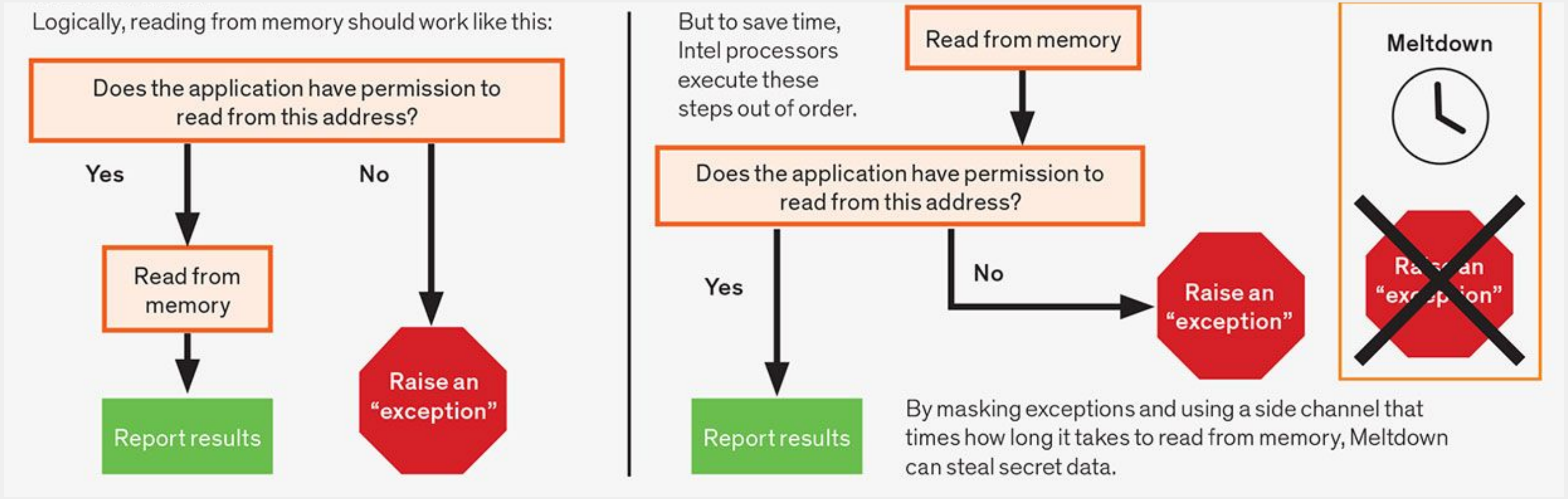
- Enables "Retpoline" for Windows on certain devices, which may improve performance of Spectre variant 2 mitigations (CVE-2017-5715). For more information, see our blog post, "Mitigating Spectre variant 2 with Retpoline on Windows".

Image source → https://support.microsoft.com/en-us/topic/march-1-2019-kb4482887-os-build-17763-348-f7a9f207-0627-1fb9-cca7-29bb7b26027fr

Meltdown

CVE-2017-5754

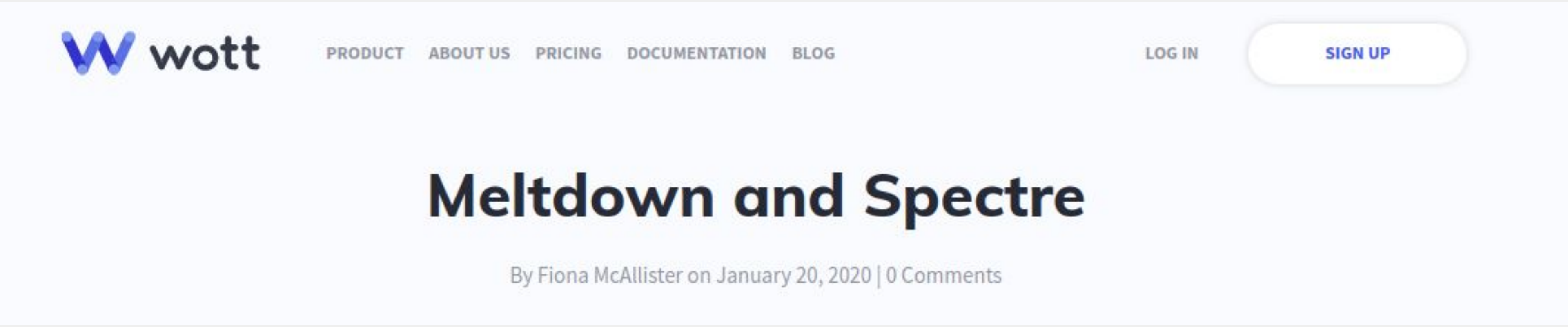
- Exploits a race condition between memory accesses & privilege check
- Bypass privilege level checks ⇒ breaks process isolation



Logically, reading from memory should work like this:
Does the application have permission to read from this address?
Yes
No
Read from memory
Report results
Raise an "exception"
But to save time, Intel processors execute these steps out of order.
Read from memory
Does the application have permission to read from this address?
Yes
No
Raise an "exception"
Report results
Meltdown
Raise an "exception"
By masking exceptions and using a side channel that times how long it takes to read from memory, Meltdown can steal secret data.

Image source → https://spectrum.ieee.org/computing/hardware/how-the-spectre-and-meltdown-hacks-really-worked

In the wild...



Meltdown and Spectre

By Fiona McAllister on January 20, 2020 | 0 Comments

“The good news is that no actual attacks have been recorded ‘in the wild.’ *However, this may be due to the fact that recording such an attack would be unlikely as the effects would not be recorded in any measurable way. Fortunately, the risk and likelihood of such attacks is relatively low given the difficulty of execution. Right now, it would be seen as a very user-targeted attack. However, that isn’t to say it isn’t possible, and as hardware and computing speeds continue to become more sophisticated, the likelihood of such attacks increase.”*

Source → https://wott.io/blog/tutorials/2020/01/20/meltdown-and-spectre

References

1. P. Kocher *et al*., "**Spectre Attacks: Exploiting Speculative Execution**," 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 1-19, doi: 10.1109/SP.2019.00002.

2. Moritz Lipp, M. Schwarz, D. Gruss, Thomas Prescher, W. Haas, A. Fogh, Jann Horn, S. Mangard, P. Kocher, Daniel Genkin, Yuval Yarom, & Michael Hamburg (2018). "**Meltdown: Reading Kernel Memory from User Space**." In *USENIX Security Symposium*.

3. Ahmad, Bilal. (2020). "**Real time Detection of Spectre and Meltdown Attacks Using Machine Learning**."

4. How the Spectre and Meltdown Hacks Really Worked