15. Hardware Security (Spectre and Meltdown Attacks)

Blase Ur and Grant Ho

February 26th, 2024

CMSC 23200

THE UNIVERSITY OF CHICAGO

Attacks that exploit processor vulnerabilities

Can leak sensitive data

Relatively hard to mitigate

Lots of media attention

Relevant Ideas in CPUs

- **Memory isolation**: Processes should only be able to read their own memory
  - Virtual (paged) memory
  - Protected memory / Protection domains
- CPUs have a relatively small, very fast cache
  - Loading uncached data can take >100 CPU cycles

Relevant Ideas in CPUs

- **Out-of-order execution**: Order of processing in CPU can differ from the order in code
  - Instructions are much faster than memory access; you might be waiting for operands to be read from memory
  - Instructions **retire** (return to the system) in order even if they executed out of order

Relevant Ideas in CPUs

- There might be a conditional branch in the instructions
- **Speculative execution**: Rather than waiting to determine which branch of a conditional to take, go ahead anyway
  - **Predictive execution**: Guess which branch to take
  - **Eager execution**: Take both branches

Relevant Ideas in CPUs

- When the CPU realizes that the branch was mis-speculatively executed, it tries to eliminate the effects
- A core idea underlying Spectre/Meltdown: The results of the instruction(s) that were mistakenly speculatively executed will be cached in the CPU *[yikes!]*

Example (not problematic as written)

Consider the code sample below. If `arr1->length` is uncached, the processor can speculatively load data from `arr1->data[untrusted_offset_from_caller]`. This is an out-of-bounds read. That should not matter because the processor will effectively roll back the execution state when the branch has executed; none of the speculatively executed instructions will retire (e.g. cause registers etc. to be affected).

```
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...;
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 ...
}
```

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

Example (really bad!!!)

However, in the following code sample, there's an issue. If `arr1->length`, `arr2->data[0x200]` and `arr2->data[0x300]` are not cached, but all other accessed data is, and the branch conditions are predicted as true, the processor can do the following speculatively before `arr1->length` has been loaded and the execution is re-steered:

- load `value = arr1->data[untrusted_offset_from_caller]`
- start a load from a data-dependent offset in `arr2->data`, loading the corresponding cache line into the L1 cache

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

Example (really bad!!!)

```
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
/* >0x400 (OUT OF BOUNDS!) */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 if (index2 < arr2->length) {
   unsigned char value2 = arr2->data[index2];
 }
}
```

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

Example (really bad!!!)

After the execution has been returned to the non-speculative path because the processor has noticed that `untrusted_offset_from_caller` is bigger than `arr1->length`, the cache line containing `arr2->data[index2]` stays in the L1 cache. By measuring the time required to load `arr2->data[0x200]` and `arr2->data[0x300]`, an attacker can then determine whether the value of `index2` during speculative execution was 0x200 or 0x300 - which discloses whether `arr1->data[untrusted_offset_from_caller]&1` is 0 or 1.

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

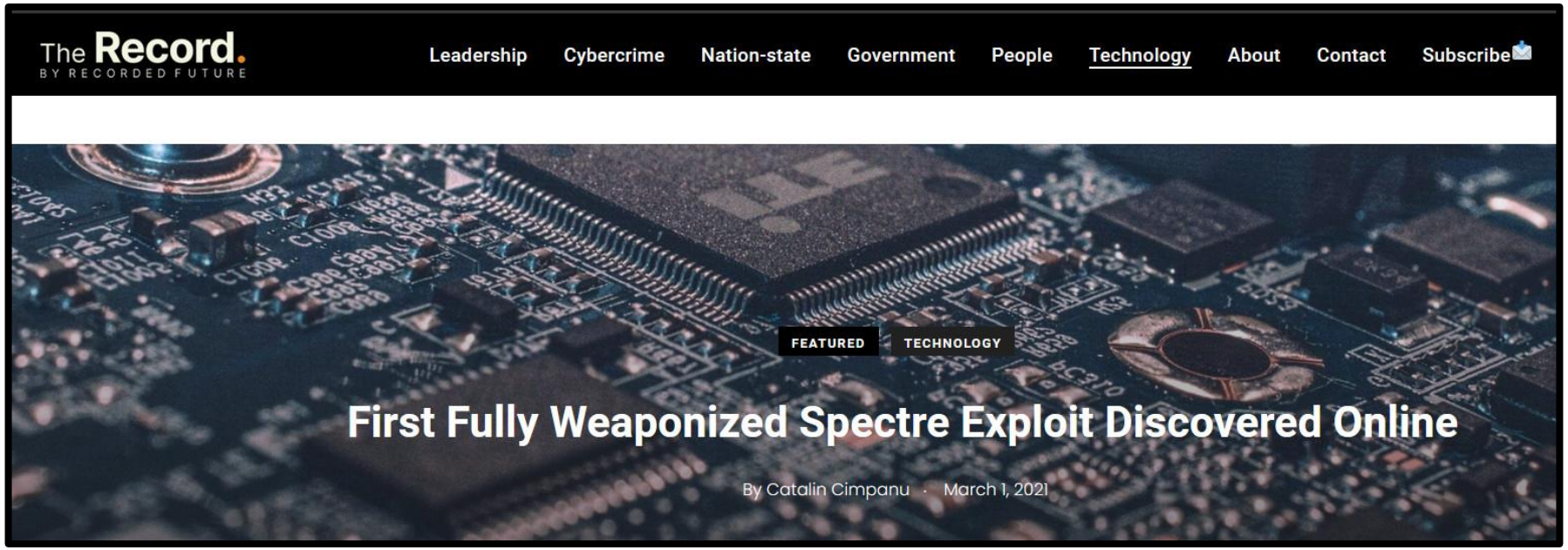
Spectre: Key Idea

- Use branch prediction as on the previous slide
- Conducting a timing side-channel attack on the cache
- Determine the value of interest based on the speed with which it returns
- Spectre allows you to read any memory from your process **for nearly every CPU**

Spectre: Exploitation Scenarios

- Leaking browser memory
- JavaScript (e.g., in an ad) can run Spectre
- Can leak browser cache, session key, other site data

Spectre: Exploitation Scenarios



"But today, Voisin said he discovered new Spectre exploits—one for Windows and one for Linux—different from the ones before. In particular, Voisin said he found a Linux Spectre exploit capable of dumping the contents of ***/etc/shadow***, a Linux file that stores details on OS user accounts"

https://therecord.media/first-fully-weaponized-spectre-exploit-discovered-online/

Meltdown: Key Ideas

1. Attempt instruction with memory operand (Base+A), where A is a value forbidden to the process
2. The CPU schedules a privilege check and the actual access
3. The privilege check fails, but due to speculative execution, the access has already run and the result has been cached
4. Conduct a timing attack reading memory at the address (Base+A) for all possible values of A. The one that ran will return faster

Meltdown: Impact

Meltdown allows you to read **any memory in the address space (even from other processes)** but only on some (unpatched) Intel/ARM CPUs

Meltdown: Timing Side Channel

- Now the attacker reads each page of probe array
- 255 of them will be slow
- The X^{th} page will be faster (it is cached!)
- We get the value of X using cache-timing side channel

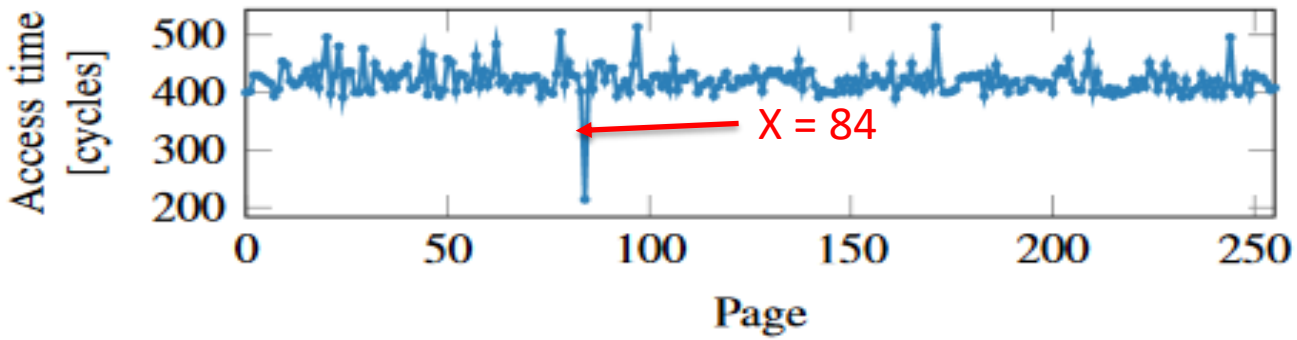


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Meltdown: Mitigation

- KAISER/KPTI (kernel page table isolation)
- Remove kernel memory mapping in user space processes
- Has non-negligible performance impact
- Some kernel memory still needs to be mapped