Meltdown

Overview of a security vulnerability

•••

Stefano Ottolenghi @ Binary Analysis and Secure Coding
Università degli Studi di Genova
December 3rd, 2018

Overview

- Meltdown breaks memory isolation and allows a process to read the entire kernel memory.
  It is a side-channel attack that leverages out-of-order execution of modern CPUs and caching mechanisms.
- Basically any Intel CPU since 1995 is vulnerable. Think about cloud computing and shared hosting.

- Discovered in January 2018.
- Authors claim to be able to be able to dump physical memory with up to 503 KB/s. My tests did not confirm this (optimistic?) result.

Hardware details (1/4) - Pipeline

Several stages to execute an instruction:

Fetch

↓

Decode

↓

Fetch operands

↓

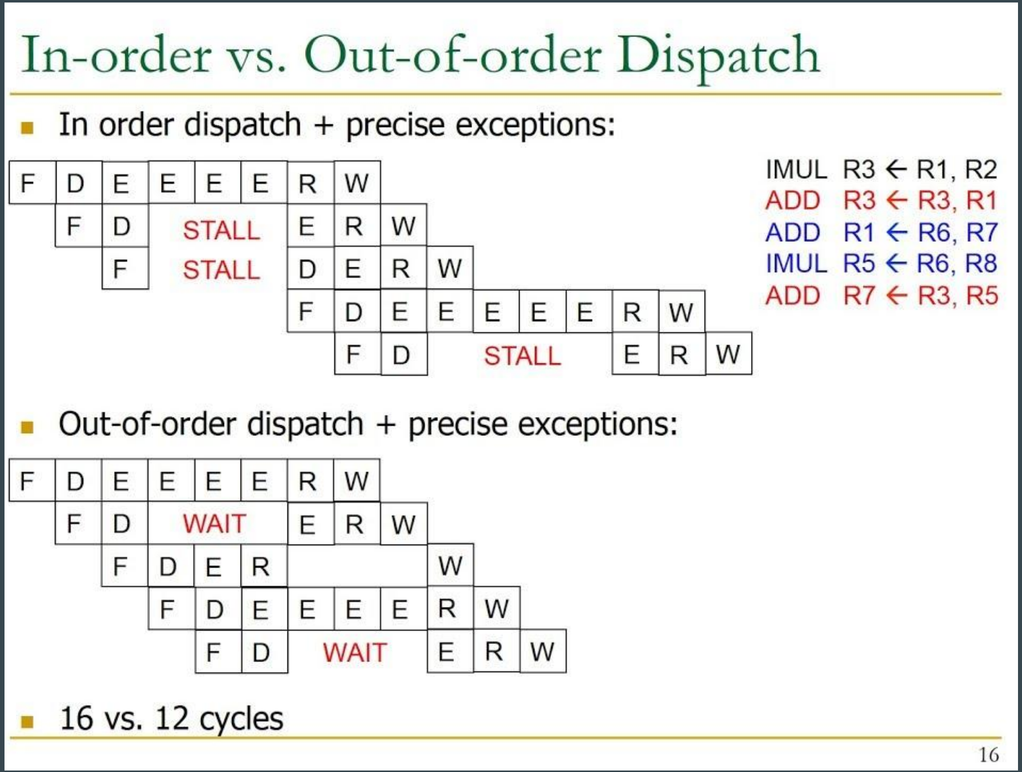
Execute

↓

Memory write

Stadio >>	S1	S2	S3	S4	S5
T0					
T1	1				
T2	2	1			
T3	3	2	1		
T4	4	3	2	1	
T5	5	4	3	2	1
T6	6	5	4	3	2
T7	7	6	5	4	3
T8	8	7	6	5	4
T9	9	8	7	6	5
T10	10	9	8	7	6
T11	11	10	9	8	7

Hardware details (2/4) - Out-of-order execution



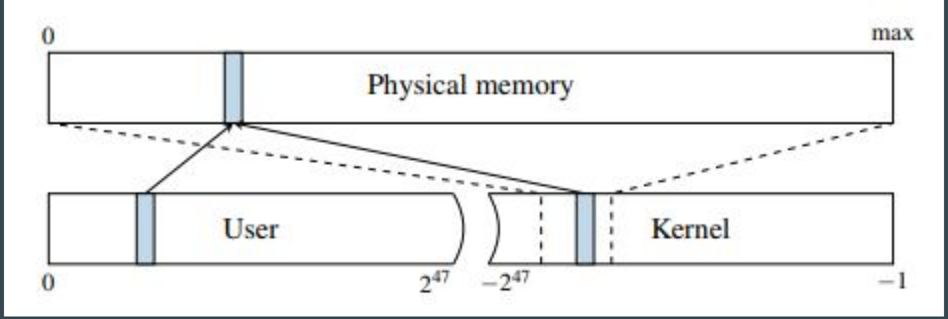
Hardware details (3/4) - Speculative execution

- When several execution flows are possible for a program, the CPU goes ahead "speculating" on which to take while waiting to evaluate the branch condition.
- If the wrong branch is taken, the CPU will clear the executed instructions and rollback.

```
x = y * 5;
if( x > pow(z, 2) )
  // Instruction A
else
  //Instruction B
```

Hardware details (4/4) - Memory mapping

- Each process has a (huge) virtual address space, split in kernel and user space.
- Kernel maps the whole physical memory.
- Isolation between kernel and user space is enforced by the CPU on read.
- The CPU is responsible for translating virtual addresses to physical ones and checking permissions.
- Hardware-based isolation is considered secure and advised by vendors.



What does it happen when a process accesses a kernel memory address?

- Both a request to check permissions AND to fetch data are sent **at the same time.**
- If permission check fails, an exception is raised and the pipeline/status flushed.
- **Returned data is cached anyway** (although future requests fetching from cache will also require the permissions check).
- (Permission check is only evaluated when data is committed.)

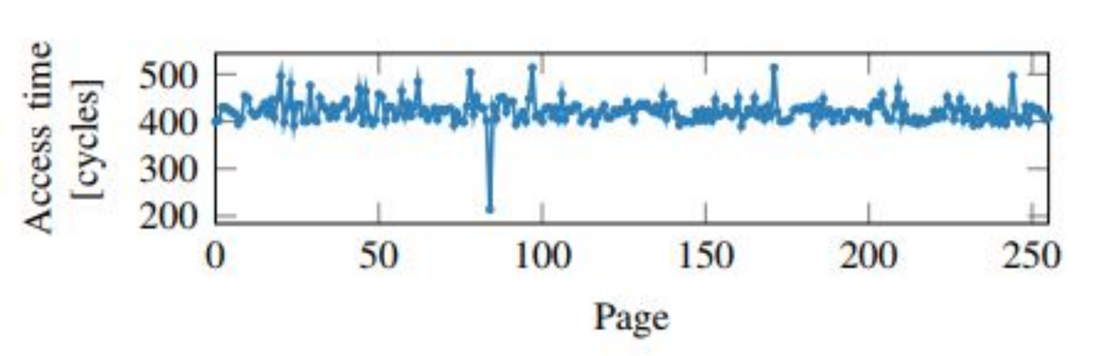
Can we exploit the cached data to read the secret value?

Read (one byte of) the secret value

- Use the secret value as index of an array we can access!

  load(program_array[secret_value]);

- With up to 256 probes with different values for the secret, we will find a page that loads faster (from cache), discovering one byte of the secret value!



The Flush+Reload (2013) technique is used to execute the cache-timing attack.

Meltdown attack

1. Set up a private array of 256 entries.
2. Flush the array from the CPU cache (*clflush*).
3. Fork.

CHILD

a. Load one byte of the secret address secret.
b. Access the array using the secret byte as index of an array we can access: *array[secret]*.
c. However, a. will trigger an exception, killing the process (but b. is likely to be executed).

PARENT

I. Wait until child is killed.
II. Access entry *n* of the array and measure timing.
III. Set n += 1 and repeat from 2.

Meltdown attack code (1/3)

```
; rcx = kernel address, rbx = probe array
xor rax, rax
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

Listing 2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

Meltdown attack code (2/3)

Line 5 scatters array accesses with strides of 4 KB = 2^12 bytes, the typical size of memory pages.

This is to prevent the prefetecher from loading subsequent array entries and caching them.

The prefetecher does not work across pages, so we scatter reads w.r.t the page size.

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

Meltdown attack code (3/3)

Line 6 retries if a zero is read.

When the CPU realizes a read from an inaccessible address happened, it zeroes out the corresponding register (to avoid seeing the value in a core dump). This could trick deceive the attack.

Meltdown assumes the secret byte was indeed ‘0’ only if there is no cache hit at all.

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

What makes Meltdown possible?

- Speculative execution can lead to out-of-order execution of undesired instructions, which may have micro-architectural side effects.
- Micro-architectural side effects can be used to infer a secret value.
- The CLFLUSH instruction can be used at all privilege levels (although it is subject to all permission checking and faults associated with a byte load).

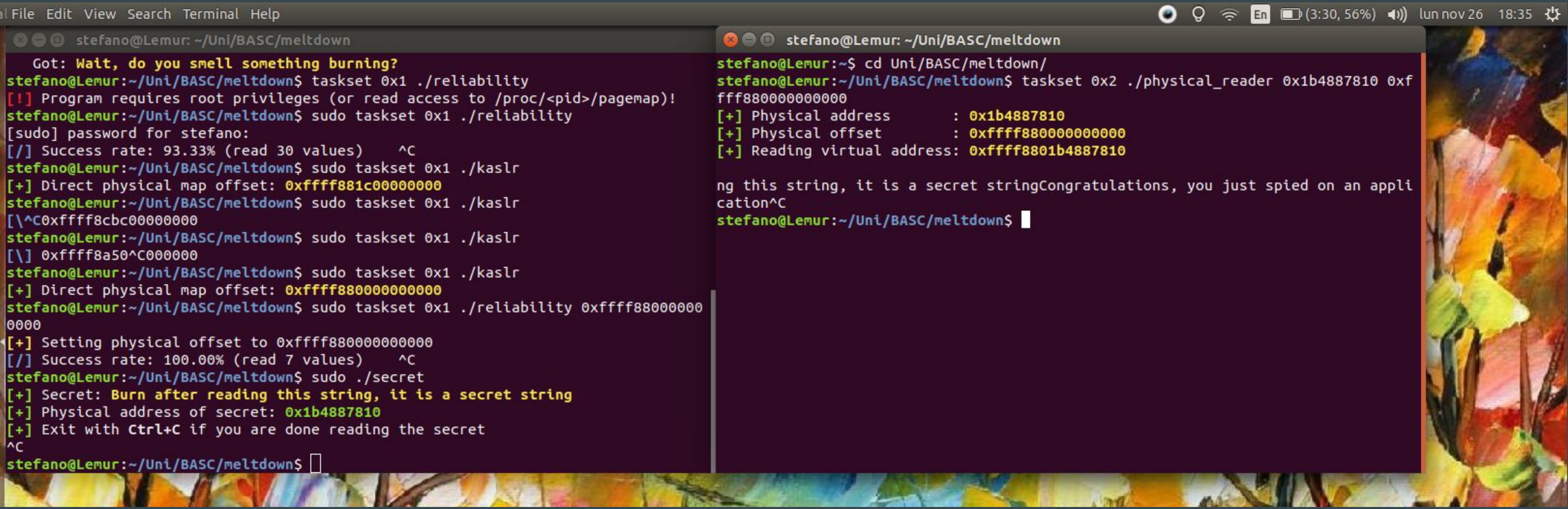
- Former head of TAO Rob Joyce "*NSA did not know about the flaw, has not exploited it and certainly the U.S. government would never put a major company like Intel in a position of risk like this to try to hold open a vulnerability.*" And other funny conspiratorial theories.

Countermeasures

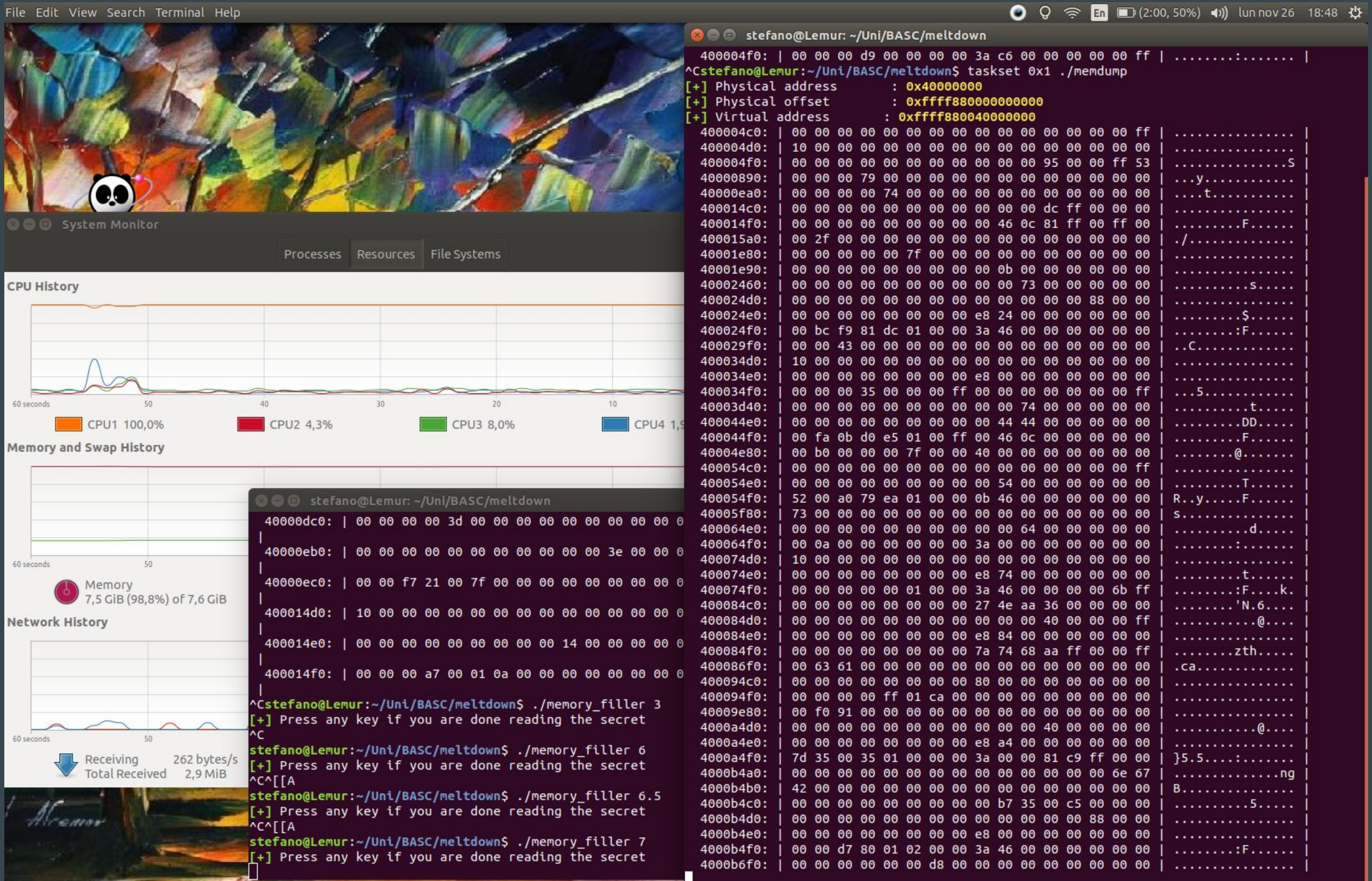
- Put permission checks before the memory access and make it blocking.
  No major slowdown should happen (page info are found together with the physical address, although L1 cache are generally virtually indexed).
  Probably what AMD does already.
- (Flushing cache lines on invalid memory access would not work, at least not straightforwardly.)

- Do not map all kernel memory in a process address space: KAISER and KPTI on Linux, but the slowdown can be non-zero.
- Encode kernel and user memory distinction in the address itself (for ex. with the left-most bit), to allow the CPU to quickly determine whether access should be allowed.

The exploit in action (1/2)



The exploit in action (2/2)



A peek into Spectre

- Still based on out-of-order execution, but relies on fooling the CPU Branch Predictor to speculatively execute the wrong branch, and read secret data.

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

- Significantly more difficult to exploit than Meltdown, but more broadly applicable. All modern CPUs are affected, and software patches are tricky (slowdown).

Resources

- Meltdown original paper: https://meltdownattack.com/meltdown.pdf
- Meltdown @ Wikipedia https://it.wikipedia.org/wiki/Meltdown_(vulnerabilit%C3%A0_di_sicurezza)
- Flush+Reload attack paper: https://eprint.iacr.org/2013/448.pdf
- Meltdown discussion on HackerNews: https://news.ycombinator.com/item?id=16107578
- Meltdown exploit repository: https://github.com/IAIK/meltdown
- Cache missing for fun and profit: http://www.daemonology.net/papers/htt.pdf
- Intel Analysis of Speculative Execution Side Channels: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf
- How to check Linux for Spectre and Meltdown vulnerability: https://www.cyberciti.biz/faq/check-linux-server-for-spectre-meltdown-vulnerability/
- Disable Meltdown/Spectre patches on Linux: https://www.phoronix.com/scan.php?page=news_item&px=Global-Switch-Skip-Spectre-Melt