CIS 5710
Computer Organization & Design

Meltdown & Spectre
Hardware Speculative Execution Attacks

Who discovered these bugs?

Who reported Meltdown?

Meltdown was independently discovered and reported by three teams:

- Jann Horn (Google Project Zero),
- Werner Haas, Thomas Prescher (Cyberus Technology),
- Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology)

Who reported Spectre?

Spectre was independently discovered and reported by two people:

- Jann Horn (Google Project Zero) and
- Paul Kocher in collaboration with, in alphabetical order, Daniel Genkin (University of Pennsylvania and University of Maryland), Mike Hamburg (Rambus), Moritz Lipp (Graz University of Technology), and Yuval Yarom (University of Adelaide and Data61)

These bugs were discovered in 2017, and publicized in January 2018

Side and Covert Channels

Side Channels and Covert Channels

- A **side channel** is a mechanism that leads to inadvertent information transfer
  - e.g., going through your neighbor's recycling
- A **covert channel** is a mechanism used for illicit communication between two cooperating parties
  - e.g., a double-agent spy communicating with their handler
- Both channels result in undesired information transfer
- Computer programs can have such channels
  - even programs that utilize secrets, like encryption keys
  - We focus especially on side channels, as they're more dangerous

Example side channel

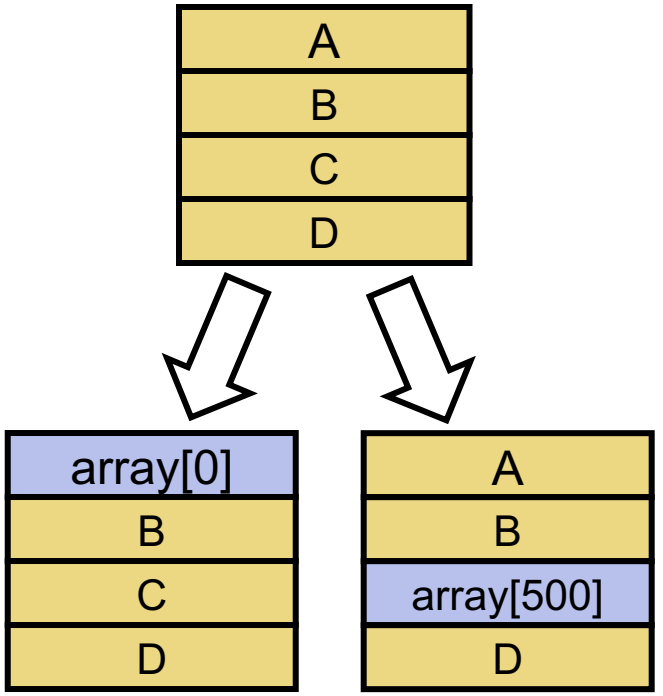
```
boolean secret = true;
int array[1024] = {…};
if (secret) {
  array[0]++;
} else {
  array[500]++;
}
```

- Threat Model: what the attacker can do
  - can't change the hardware or OS, which work correctly
  - can run arbitrary programs at arbitrary times
- How can we learn the secret?
  - go through the program's recycling and see what we find

Example side channel

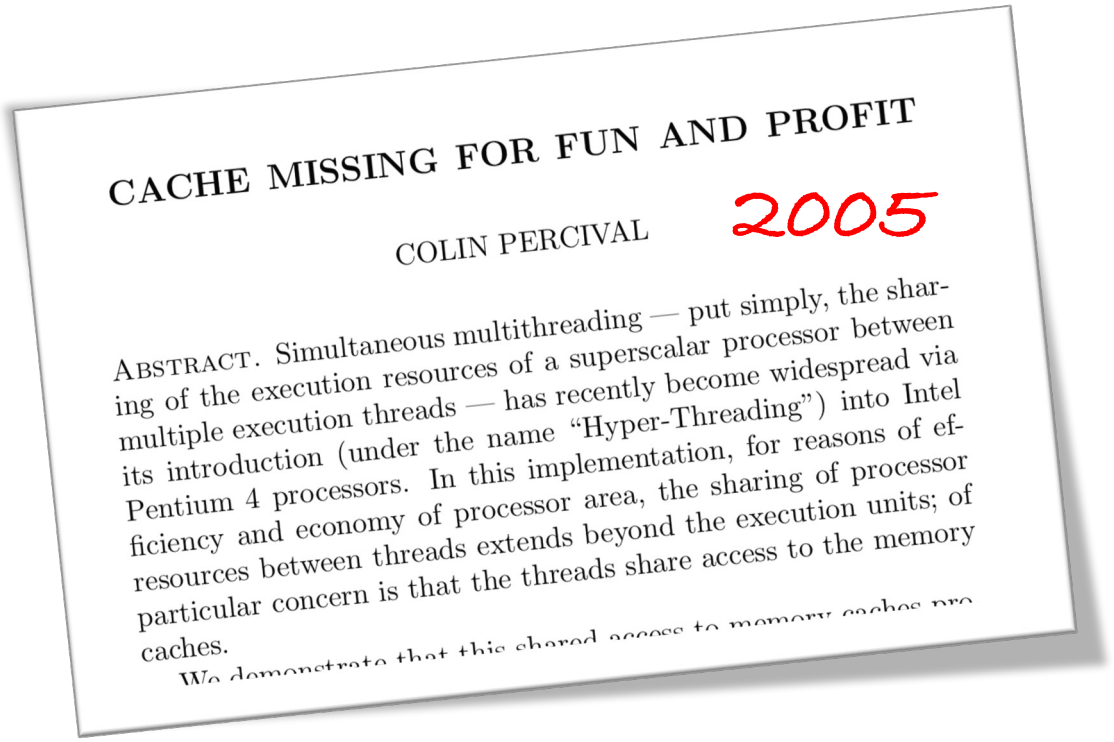
```
boolean secret = true;
int array[1024] = {…};
if (secret) {
  array[0]++;
} else {
  array[500]++;
}
```

- We fill up the cache with known data
- Then, run the secret program
- Then, see what's left in the cache
  - access all the known data, see which accesses are fast and which are slow
- Caches can be a side channel
  - inadvertently revealing information about program execution



Cache Side Channel Attacks

- Can be used for both:
  - **side channel** (unintentional communication)
  - covert channel (intentional communication)
- We can steal SSH private keys this way



CACHE MISSING FOR FUN AND PROFIT

COLIN PERCIVAL

2005

ABSTRACT. Simultaneous multithreading — put simply, the sharing of the execution resources of a superscalar processor between multiple execution threads — has recently become widespread via its introduction (under the name "Hyper-Threading") into Intel Pentium 4 processors. In this implementation, for reasons of efficiency and economy of processor area, the sharing of processor resources between threads extends beyond the execution units; of particular concern is that the threads share access to the memory caches.

We demonstrate that this shared access to memory caches pro

Side channels are delicate

- Might have to go through a lot of recycling to find what you're looking for
  - side channels often require many executions to recover a secret
- Other side channels:
  - power consumption
  - execution time
  - electromagnetic radiation
  - microarchitectural features like contention for execution ports
  - attackers always find new ones

Abusing OoO Execution

Meltdown code

Let's read kernel memory from userspace >:-)

r0, r1: temps
r2: kernel address we wish to read
r3: start of probe_array

```
ldr r0 <- [r2]
sll r0 <- r0, #6
add r1 <- r3, r0
ldr r1 <- [r1]
```

What does this code do?

Based on the **data value** at [r2], we access a particular cache line of probe_array.

Meltdown code on an OoO processor

Let's read kernel memory from userspace >:-)

r0, r1: temps
r2: kernel address we wish to read
r3: start of probe_array

```
ldr r0 <- [r2]
sll r0 <- r0, #6
add r1 <- r3, r0
ldr r1 <- [r1]
```

<= this raises an exception…but not until Commit!

After the exn, we clear the ROB and other OoO structures **but not the caches**, leaving open a side channel!

Speculative Execution Attacks

- An OoO processor speculates aggressively
  - on a mis-speculation, the incorrect instructions are wiped away and have no ISA-level visible impact
    - register writes, memory stores are all cleaned up
    - non-ISA level changes like cache contents **are not cleaned up**
    - insns that roll back can still leak information via a side channel!
- These attacks strike at core principles of OoO execution
  - undiscovered since 1990s?

> Speculative execution can proceed far ahead of the main processor. For example, on an i7 Surface Pro 3 (i7-4650U) used for most of the testing, the code in Appendix A works with up to 188 simple instructions inserted in the source code between the 'if' statement and the line accessing array1/array2.

Meltdown experiment

- Real Meltdown uses pages instead of cache lines, exploiting a TLB side channel

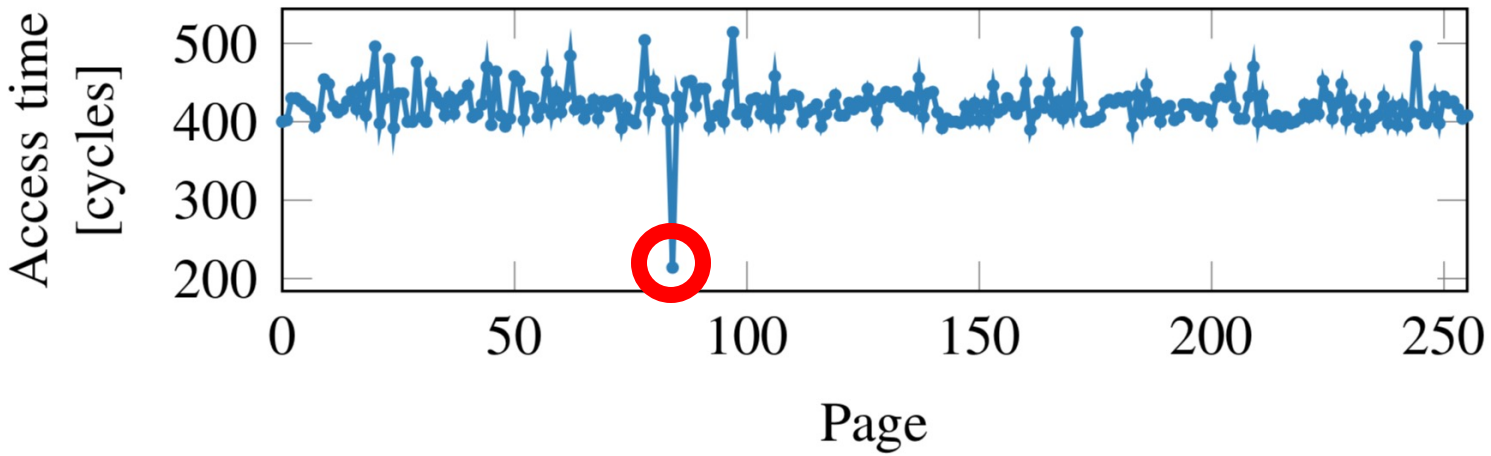


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Meltdown Mitigations

- Meltdown affects Intel OoO and some ARM OoO processors, AMD is immune
  - Intel did page permission checks at Commit, not Execute
  - Intel cores have hardware fixes as of ~2019
- Meltdown can be patched in software
  - all major OSes released patches in 2018
  - performance impact for system calls
  - KPTI/KAISER Linux patch maps minimal kernel code/data

Spectre Attacks

What happens on a Context Switch

- Operating system responsible to handle context switching
  - Hardware support is just a timer interrupt
- Each process has an associated data structure which is used to record relevant state such as:
  - **Architected state**: PC, registers, Page table pointer
    - Save and restore them on context switches
    - Memory state?
  - **Non-architected state**: caches, predictor tables, etc.
    - Ignore or flush
    - ignoring is key aspect of Spectre/Meltdown vulnerabilities!
- OS swaps out values for old process, swaps in values for new process and restarts the system.

Spectre Attack

- Goal: trick another process into leaking information
- Let's say my secure program contains this code:

```
if (x < array1_size) {
  y = array2[array1[x] * 256];
}
```

- If attacker controls x, value of array1[x] can be leaked
  - problem: how to get around the bounds check?

Poison the branch predictor

- How do I poison the BHT/BTB?
  - If I know the BHT/BTB entries that will be used for the bounds check branch, I can train them in advance
    - e.g., branch with the same PC in the attacker process
  - Not hard when the attacker provides their own code
    - JavaScript, VM on EC2
- How do I get the victim to run the vulnerable code?
  - highly victim-specific
  - easiest in JIT environments
  - Spectre paper shows how to read Chrome-internal state from JavaScript

Spectre Mitigations

- Spectre affects OoO chips from everyone
  - confirmed on Intel, ARM, AMD
- Software patches are incomplete, slow
  - Reduce use of indirect branches
  - Disable speculation through special instructions at “critical” code points
- Ultimately, Spectre attacks aren’t that easy to pull off