Meltdown & Spectre

Edward Higgins

Brief overview of Meltdown & Spectre

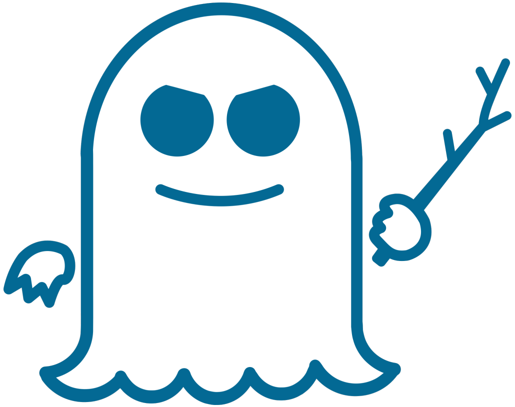
Meltdown

- Exploits side effects of out-of-order execution to read arbitrary kernel memory
- Affects all modern Intel CPUs in recent years

Figure: Meltdown

Spectre

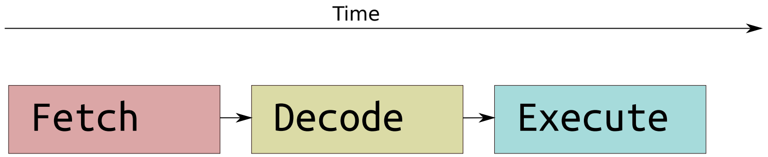
- Induces victim to speculatively perform operations which leak information
- Affects many high-performance CPUS, including Intel, AMD and ARM chips in recent years, and others



Modern CPU Architecture

Traditional Computers

- Next instruction **fetched** from memory
- Instruction is **decoded** and the location any of indirectly referenced memory is interpreted
- Instruction is **executed**, and written to wherever is specified
- Each instruction takes several clock ticks



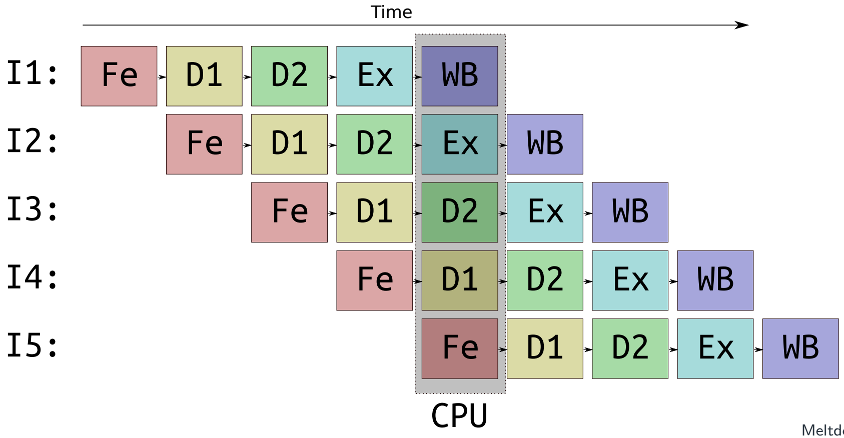
Memory Hierarchy

- Modern computers use multiple types of memory
- Each Various levels present different trade-offs of speed & memory
- When the CPU needs memory, it will move it into a lower-level cache
- Example:

Memory	Size	Latency
L1 Cache	64 KB	4-12
L2 Cache	256 KB	26-31
L3 Cache	4 MB	43-60
RAM	8 GB	100s

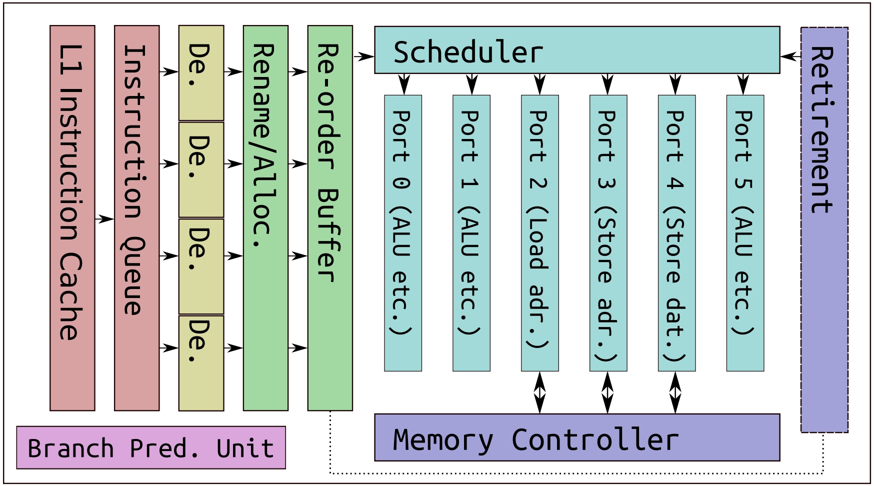
Pipelining

- F-D-E cycle micro-ops can be done in parallel, each is done by different hardware
- Breaking the cycle down into more micro-ops allows more instructions to be processed at once



Out-of-order execution

- Modern CPUs allow micro-ops of many operations to be done **out of order**



Speculative execution

- Allows commands such as loads & stores to be issued before:
  - preceding branches resolve
  - preceding operations complete

Branch prediction

- Branches include:
  - Conditionals
  - Direct calls & jumps
  - Indirect calls & jumps
  - Returns
- Calculated by the Branch prediction unit (BPU), including:
  - Return stack buffer (RSB): A history of recent return addresses
  - Branch target buffer (BTB): Recent outcomes from conditionals/calls

Exploiting the architecture

Transient instructions

- Instructions that:
  - Are executed out of order
  - Leave measurable side effects
- Occur all the time in normal operation
- Exploitable if their operation depends on a secret channel

Example

```
...
if (x < 0)
  call OutOfBounds();
var = array[x];
...
```

Transient instructions

- Instructions that:
  - Are executed out of order
  - Leave measurable side effects
- Occur all the time in normal operation
- Exploitable if their operation depends on a secret channel

Example compiled

```
...
cmp rax, 0             ; Compare register rax to 0
jl OutOfBounds         ; If rax < 0, jump elsewhere
mov rcx, [rbx + rax]   ; Now, move some memory into rcx
...
```

Side-channel attacks

- Usually, multiple programs run on the same hardware
- State of the CPU can be changed by these programs
- Such changes may be detectable by other programs

Example state changes:

- Branch history
- BTB
- **Caches** (e.g. Flush+Reload)

Meltdown attack

Overview

- Allows non-privileged users to read privileged memory

3 Steps to Meltdown

1. The content of a restricted memory location is loaded into a register, throwing an exception
2. A transient instruction accesses an uncached memory address based on the contents of that register, fetching it into cache
3. A side-channel attack (e.g. Flush+Reload) used to determine which memory has been moved to cache, revealing the value of the restricted memory

Steps 1 & 2: Transmission of the secret

- Line 5 attempts to retrieve the secret byte from address `rcx` into `rl`
- CPU checks permission bits of address, and raises an exception
- While that is happening line 8 speculatively fetches some offset from the probe array, caching it
- Once line 5 retires, the exception resolves and the CPU registers and pipeline are flushed

```
1  ; rcx = secret address
2  ; rbx = probe array
3
4  retry:
5  mov a1, byte [rcx]
6  shl rax, 0xc
7  jz retry
8  mov rbx, qword [rbx + rax]
```

Step 3: Receiving the secret

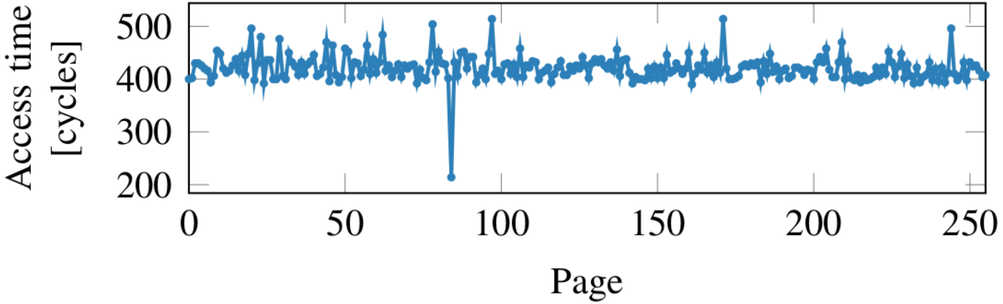


Figure: Cache timing

- Using a Flush+Reload attack, access time to the probe_array can be measured
- By timing access to each entry in the probe array, the entry corresponding to the value of the secret byte becomes apparent, (in this case it was 84)

But which addresses?

- User processes don't know physical addresses, they use a virtualised address space
- User processes may need to access the kernel, so kernel memory is mapped within this space
- Since the kernel manages everything, the entire physical memory is mapped within the kernel address space

Address Space Layout Randomization

- In the past, the address of physical memory was easy to figure out for a given kernel
- Within the past 15 years, ASLR has been implemented in all main OSs to randomize these addresses
- Randomization is limited to 40 bits, so on a machine with 8GB of memory, only 128 tests are needed to find the actual physical memory
- Once found, the attacker can proceed to dump the entire physical memory

Performance

- Since steps 1 and 2 are much faster than step 3, performance can be improved by only reading 1 bit at a time:
- In this case, only one read of the probe array is needed:
  - if it's cached it's a 1
  - else it's a 0
- Using this technique, an attacker can read any portion of physical memory at $>$500KB/s, with an error rate of $<$0.04%

Meltdown Fixes

- The fix for meltdown involves remapping the virtual address space every time a program makes a system call to the kernel
- This means that the kernel memory won't be in unprivileged processes' address spaces, but will slow down certain operations
- This has been patched in all major OSs ("Kernel Page Table Isolation" or KPTI for linux)
- Make sure your computers are up-to-date to minimize the risks

Spectre attacks

Overview

- Allows attacker to trick a victim process into revealing secret memory from their address space
- Involves training the victim code to speculatively execute code it otherwise wouldn't
- 2 approaches involving :
    1. Training the outcome of a conditional branch in the victim
    2. Training the call address of a victim's call

Exploiting conditional branch misprediction

- Consider some victim code:

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

- Calling this code (e.g. through an API) with allowed `x` multiple times trains the CPU to speculatively execute line 2.
- Now, calling with some malicious `x`, line 2 can cache memory based on the target value, as previously mentioned

Exploiting conditional branch misprediction (contd.)

- Selecting appropriate values for `x` allows an attacker to read arbitrary memory from the victim's address space
- For example:
  - Accessing secrets from a cryptographic library
  - Accessing arbitrary browser data from a sandboxed JS environment

Poisoning indirect branches

- In some cases, a victim will make a branch call while the attacker has control over some CPU registers
- E.g. A function making a function call while dealing with externally provided data
- The attacker can train the BTB to branch to some *gadget* code instead of the correct destination
- This way, data from addresses calculated from those registers can be leaked

Spectre fixes

- Much harder to fix than Meltdown, KPTI and similar fixes won't work
- Fixes can include allowing indirect branches to be isolated from speculative execution
- Likely to be an issue for a while

Summary

- Performance optimisations in modern CPUs have left them vunerable to attacks
- Meltdown and Spectre attacks demonstrate some ways in this can be done
- Now we know, we can try to mitigate the risks
- However, this will likely be an issue for a while

Many thanks!

Useful Resources

1. Google project zero post
2. Meltdown paper
3. Spectre paper
4. Intel x86 optimization manual
5. Google post on spectre fix

Appendix

Intel Haswell Microarchitecture

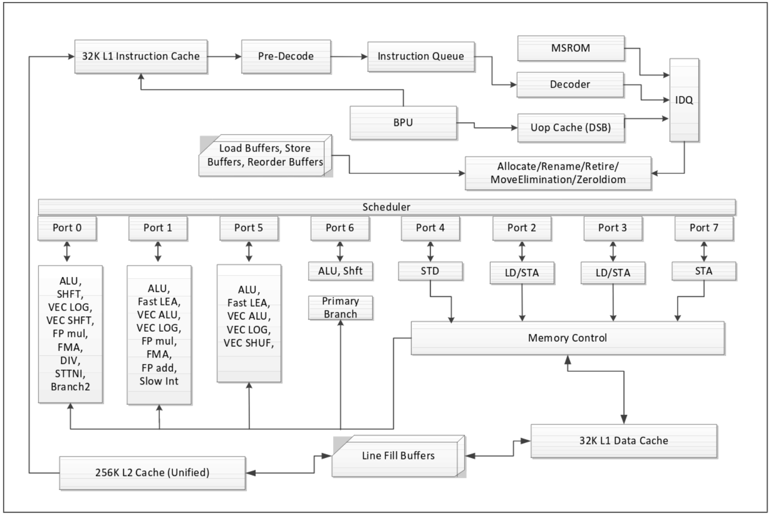

32K L1 Instruction Cache
Pre-Decode
Instruction Queue
MSROM
Decoder
IDQ
BPU
Uop Cache (DSB)
Load Buffers, Store Buffers, Reorder Buffers
Allocate/Rename/Retire/ MoveElimination/ZeroIdiom
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
ALU, SHFT, VEC LOG, VEC SHFT, FP mul, FMA, DIV, STTNI, Branch2
ALU, Fast LEA, VEC ALU, VEC LOG, FP mul, FMA, FP add, Slow Int
ALU, Fast LEA, VEC ALU, VEC LOG, VEC SHUF,
ALU, Shft
Primary Branch
STD
LD/STA
LD/STA
STA
Memory Control
32K L1 Data Cache
Line Fill Buffers
256K L2 Cache (Unified)

Figure: Haswell microarchitecture

AMD Zen Microarchitecture

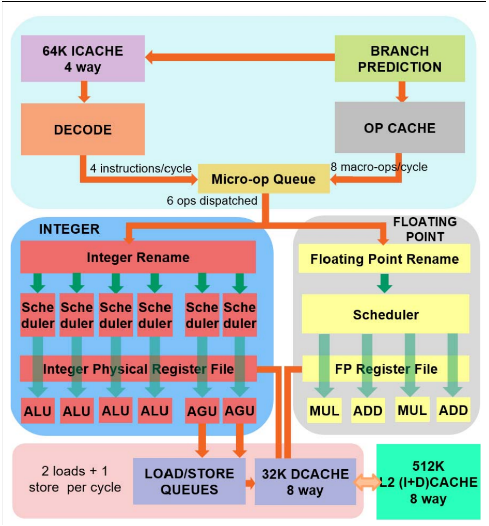
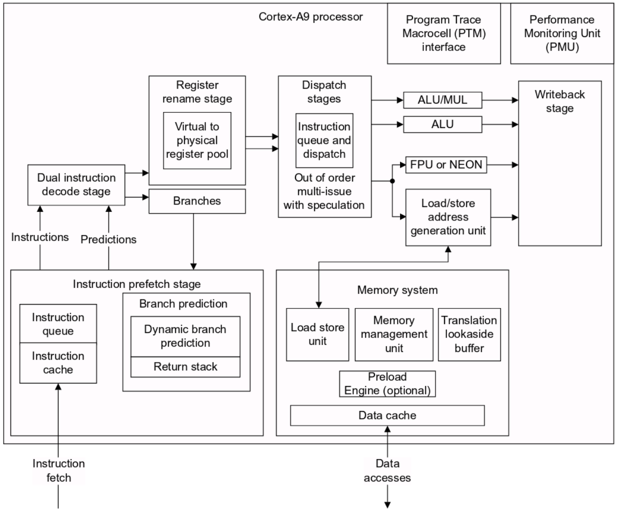


Figure: AMD Zen microarchitecture

ARM Coretex A9 Microarchitecture



Cortex-A9 processor
Program Trace Macrocell (PTM) interface
Performance Monitoring Unit (PMU)
Register rename stage
Virtual to physical register pool
Dispatch stages
Instruction queue and dispatch
Out of order multi-issue with speculation
ALU/MUL
ALU
FPU or NEON
Load/store address generation unit
Writeback stage
Dual instruction decode stage
Branches
Instructions
Predictions
Instruction prefetch stage
Instruction queue
Instruction cache
Branch prediction
Dynamic branch prediction
Return stack
Memory system
Load store unit
Memory management unit
Translation lookaside buffer
Preload Engine (optional)
Data cache
Instruction fetch
Data accesses

Figure: ARM Coretex A9 microarchitecture