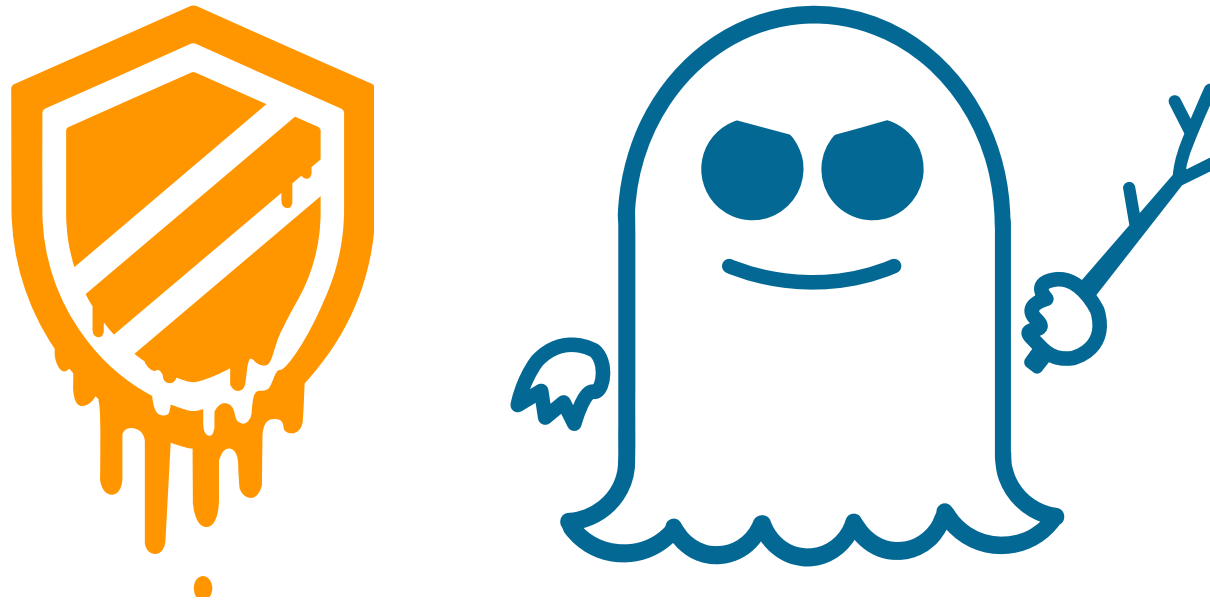


Meltdown & Spectre: Microarchitectural security bugs



<https://meltdownattack.com/>

Meltdown & Spectre

- **Intro**
- **Introduction to Processors**
- **Fast processors**
- **Side-channel attacks**
- **Meltdown**
- **Spectre 1**
- **Spectre 2**
- **Workarounds**

Intro

- **Allow unprivileged programmes to read kernel memory**
- **Demo of it being done via Javascript in a browser through a VM !**

But pretty hairy

- **Not a software bug**
- **Hardware meets specification**
- **Most modern fast processors**

Spectre more common, Meltdown rarer (Intel some ARM)

Some low end ARMs/Atoms are immune

Processors (aka CPUs, Cores)

- **CPUs execute a stream of instructions, reading/writing memory and registers**

- **Instructions:**

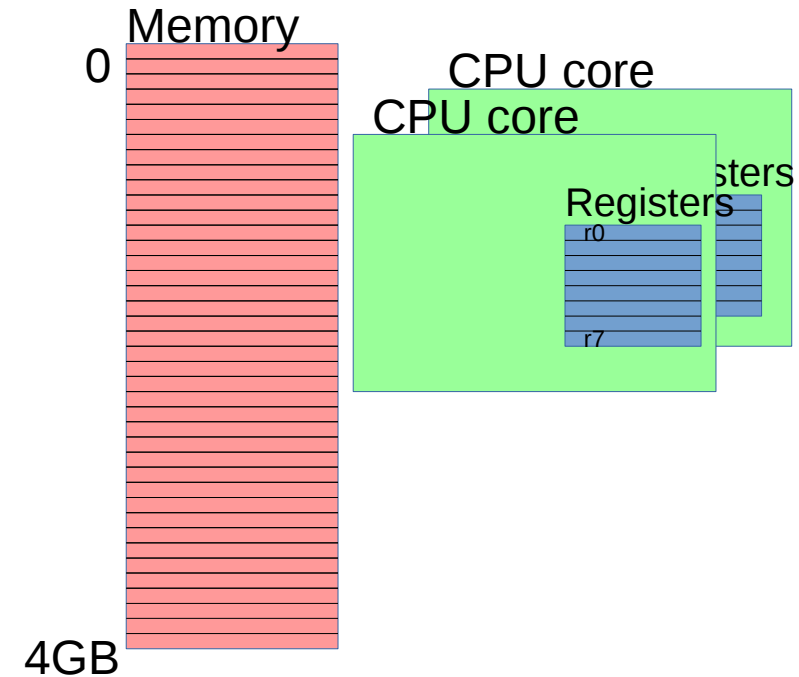
Add, Multiply Load, Store, Compare, Branch; all stored in memory

- **Memory:**

Slow, but big; think of as having a Single large address (e.g. 0...4 billion)

- **Registers:**

Fast, but not many, e.g. 16



Programs

- ◆ **Sequences of instructions**

At a series of memory locations

All just numbers

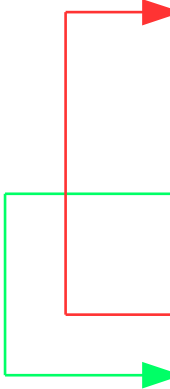
- ◆ **Lets ignore how they get there, how they start running or how they stop**

- ◆ **Some special instructions**

e.g. syscalls to enter OS, return, etc etc

Address

0	5	1	0	Set r1= 0
1	1	2	3	Load r2 ← [r3]
2	a	33	1	Add r3,r3,#1
3	c	2	0	Compare r2,#END
4	b	0	7	Branch if-equal 7
5	a	11	1	Add r1,r1,#1
6	b	7	1	Branch always 1
7	7	55	1	Store result ← r1
8	f	00	0	exit



Simple string length

Fetch, Decode, Execute (Simple)

- **For every instruction:**

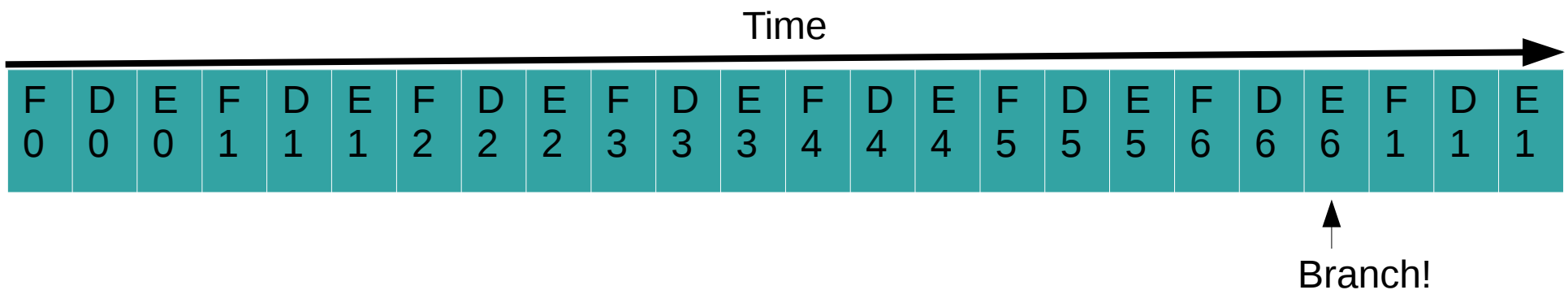
 - Fetch it from memory

 - Decode it (e.g. 5/1/0 means Set r1=0)

 - Execute it (store 0 in r1)

- **At least 3 clocks/instruction**

 - Some times 'executes' take longer (e.g. a multiply)



Pipelines

- **Overlap instructions**

Upto 1 instruction/cycle

- **Branches get complex**

throw away stuff

0	5	1	0	Set r1= 0
1	1	2	3	Load r2 ← [r3]
2	a	33	1	Add r3,r3,#1
3	c	2	0	Compare r2,#END
4	b	0	7	Branch if-equal 7
5	a	11	1	Add r1,r1,#1
6	b	7	1	Branch always 1
7	7	55	1	Store result ← r1
8	f	00	0	exit

F	F	F	F	F	F	F	F	F	F	F	F	F
0	1	2	3	4	5	6	7	8	1	2	3	4
	D	D	D	D	D	D	D	D	D	D	D	D
	0	1	2	3	4	5	6	7	8	1	2	3
		E	E	E	E	E	E	E	E	E	E	E
		0	1	2	3	4	5	6	7	8	1	2

Caches: A fast small chunk of memory

- **CPU clocks got faster**

But memory **latency** lagged

e.g. 0.3ns CPU clock

10ns RAM latency

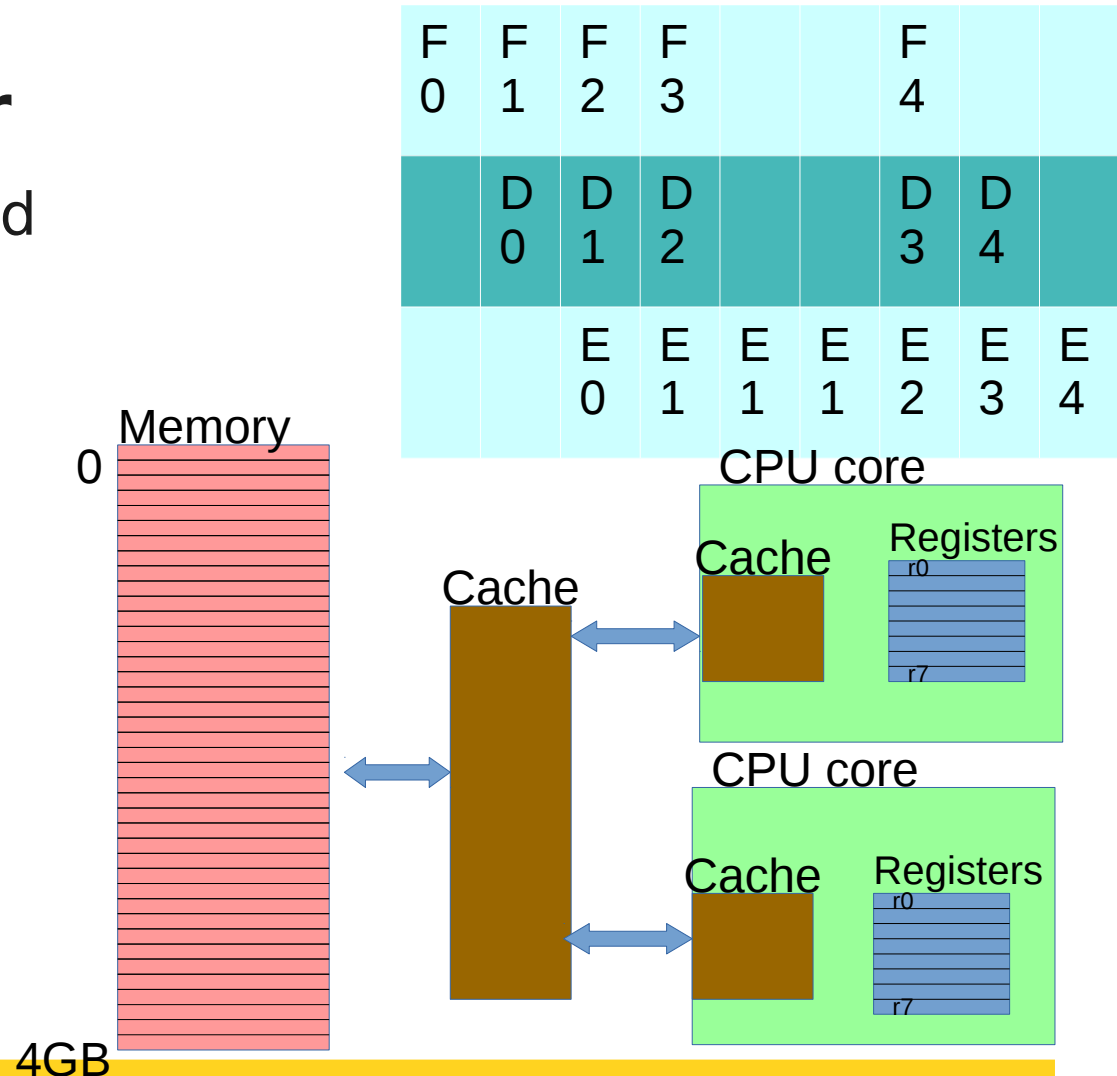
30 cycles to fetch data

- **Caches**

Between core and memory

Often multiple levels

Shared between cores



Caches: Allocation & Replacement

- **Multiple 'lines'**

Hold one chunk of data from RAM

- **Easiest way to find slot**

e.g. address/lines remainder

- **Replace when reused**

Read 3008 - goes in slot 8 - SLOW

Read 3008 - read from slot 8 - FAST

Read 4008 - replaces slot 8 - SLOW

Read 3008 - read RAM again - SLOW

Lines	Actual address	Data
0	Free	
1	Free	
2	2002	
3	2003	
4	Free	
5	Free	
6	Free	
7	2007	
8	3008	
9	5109	

Pipelines: Got more complex

- **Split up more: Faster clocks**
- **Multiple execution units**
- **Lots of instructions in flight**

Maybe over 100

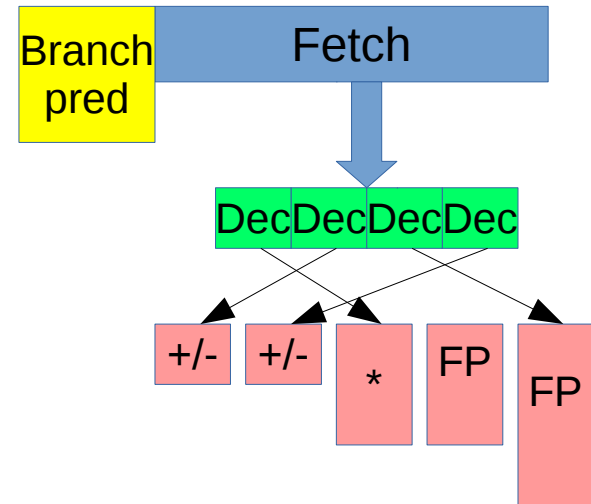
- **Instructions out-of-order**

Skip an instruction while waiting for a result

Put them back at the end

- **Branches even more costly**

Must try and **predict** branches based on what they did before



Hyperthreading/SMT

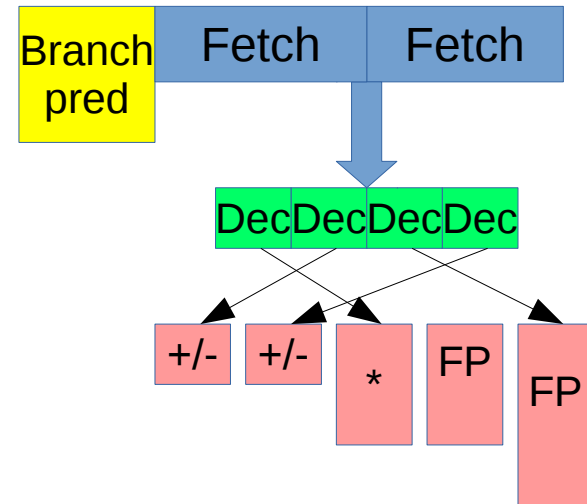
- **One complex pipeline running 2 or more sets of code**

Looks like multiple cores

- **Shares complex things like**

Branch prediction

All the execution units



Virtual memory & permissions

- **User processes can't access each others memory**

One 'page table' per process

- **User processes can't access kernel memory**

'privilege flag' in pages

- **You can swap**

Page tables can have gaps that cause errors when accessed, to cause disk to be read

0	3005	
1	Disk	
2	4000	
3	3001	
4	Disk	
5		
6	8000	Kernel!
7	8001	Kernel!
8	8002	Kernel!
9	8003	Kernel!

Sidechannel attacks

- ♦ **If CPU does what it is supposed to**
Rely on the **timing** of instructions
- ♦ **e.g. if a read is fast it's in the cache**
Which probably means someone else read it first!
- ♦ **Often slow exploits**
Having to time things and wait
- ♦ **Originally demonstrated in special cases**
e.g. finding out if another process was using crypto tables
Previously felt pretty obscure

Meltdown!

- **Read kernel memory!**

- (a) means the rest is just prediction

- (b) loads from the kernel memory – still happens on some systems, but protection guarantees thrown away **eventually**

- But not before (c) that uses it to choose an address to access

- (d) Accesses memory later – at base+.... - finds which one is fast : The one that is fast corresponds to (b)'s value

- Relies on CPU implementation getting data in (b) before it notices permission error; some CPUs do, some don't.

a	Branch
b	Load r1 ← kernel
c	Load r2 ← [base+r1*4096]
d	Time memory access

Spectre v1

- **Get kernel to do it for you**

Find somewhere in kernel that accesses table based on user input

Of course kernel range checks it first

- **Speculation means load happens and contaminates cache**

a	Cmp r1 < table-size ?
b	Branch if too big
c	Load r2 ← [...r1]
d	Load r3 ← [r2...]
e	Time memory access

Spectre v2

- **Mistrain branch predictor to go where you want it!**

Very hard, branch predictor algorithms are complex; always assumed to be unpredictable - they reverse engineered it!

Only *indirect* branches normally

- **Find somewhere in kernel that has the ‘loads’**

- **Hardest to exploit**

Also hardest to fix!

But with public proof-of-concept

- **Almost all complex processors (i.e. with complex branch predictors)**

a	Misuse branch in kernel to...
b	Load r2 ← [...r1]
c	Load r3 ← [r2...]
d	Time memory access

Workaround: Meltdown

♦ 'Page Table Isolation'

Keep two page tables

One for user-space has no kernel pages at all

Switch everytime we go in and out of kernel

Expensive

A bit better on newer CPUs with 'PCID'

dmesg|grep isolation

	User	
0	3005	
1	Disk	
2	4000	
3	3001	
4	Disk	
5		
6	8000	Kernel!
7		
8		
9		

	Kernel	
0	3005	
1	Disk	
2	4000	
3	3001	
4	Disk	
5		
6	8000	Kernel!
7	8001	Kernel!
8	8002	Kernel!
9	8003	Kernel!

Workaround: Spectre v1

- ◆ **Pointer sanitization**

Any user pointer that's checked needs an extra barrier to stop the pipeline.

Special instruction (new?)

Only some architectures have them

Some masking tricks

- ◆ **Need to find every use in kernel!**

Compiler tools

Beware closed source modules

a	Cmp r1 < table-size ?
b	Branch if too big
	BARRIER!
c	Load r2 ← [...r1]
d	Load r3 ← [r2...]
e	Time memory access

Workaround: Spectre v2 [Retpoline]

- ◆ **Avoid all indirect branches!**

Find an instruction that doesn't get branch prediction

'ret' - return from function

- ◆ **Compiler changes**

- ◆ **Manual code needs checking**

- ◆ **Slower**

- ◆ **Doesn't work on latest CPUs**

Because they predicted returns

Needed ways to stop that

Workaround: Spectre v2

[Branch speculation flush/restriction]

- **Before retpoline solution**
- **New microcode**
 - Gives your CPU new instructions!
 - Changes the way branch predictor works
 - Stops predictions from userspace influencing kernel
 - Protects hyperthreads from each other
- **Might be faster on future chips**
 - When they design them in rather than bodge on later
- **Got to wait for microcode for your chip**
- **No microcode (or equivalent) on some RISC chips**

Summary

- ♦ **A new type of attack**

Probably more with similar idea coming

- ♦ **Not currently easy to actually use Spectre**

A few KB/s read

Hard to exploit remotely, but demonstrable

- ♦ **Fixes are all pretty messy**

All slow things down a bit

How much varies vastly depending on task and CPU