We adapt. You succeed.

Speculative Execution HW BUGs, Virtualization & Other Things...

Dario Faggioli <dfaggioli@suse.com>
Software Engineer - Virtualization Specialist, **SUSE**
GPG: 4B9B 2C3A 3DD5 86BD 163E 738B 1642 7889 A5B8 73EE
https://about.me/dario.faggioli
https://www.linkedin.com/in/dfaggioli/
https://twitter.com/DarioFaggioli (@DarioFaggioli)

Myself, my Company, what we’ll cover today…

About myself: Work

- Ing. Inf @ UniPI
  - B.Sc. (2004) *"Realizzazione di primitive e processi esterni per la gestione della memoria di massa"* (Adv.s: Prof. G. Frosini, Prof. G. Lettieri
  - M.Sc (2007) *"Implementation and Study of the BandWidth Inheritance protocol in the Linux kernel"* (Adv.s: Prof. P. Ancilotti, Prof. G. Lipari)

- Ph.D on Real-Time Scheduling @ ReTiS Lab, SSSUP, Pisa; co-authored `SCHED_DEADLINE`, now in mainline Linux

- Senior Software Engineer @ Citrix, 2011; contributor to The Xen-Project, maintainer of the Xen's scheduler

- Virtualization Software Engineer @ SUSE, 2018; still Xen, but also KVM, QEMU, Libvirt. Focuson performance evaluation & improvement

- https://about.me/dario.faggioli , https://dariofaggioli.wordpress.com/about/

About my Company: SUSE

- We're one of the oldest Linux company (1992!)
- We're the "open, Open Source company"
- We like changing name:
  S.u.S.E. → SuSE → SUSE
- We make music parodies
- Our motto is: "Have a lot of fun!"

Academic program:

suse.com/academic/

We're (~always) hiring:
suse.com/company/careers

Spectre, Meltdown & Friends

- **Spectre v1** - Bounds Check Bypass
- **Spectre v2** - Branch Target Isolation
- **Meltdown** - Rogue Data Cash Load (a.k.a. Spectre v3)
- **Spectre v3a**- Rogue System Register Read
- **Spectre v4** - Speculative Store Bypass
- **LazyFPU** - Lazy Floating Point State Restore
- **L1TF** - L1 Terminal Fault (a.k.a. Foreshadow)
- **MDS** - Microarch. Data Sampling (a.k.a. Fallout, ZombieLoad, ...)

Will cover: Meltdown. *Maybe* Spectre. *Maybe* L1TF

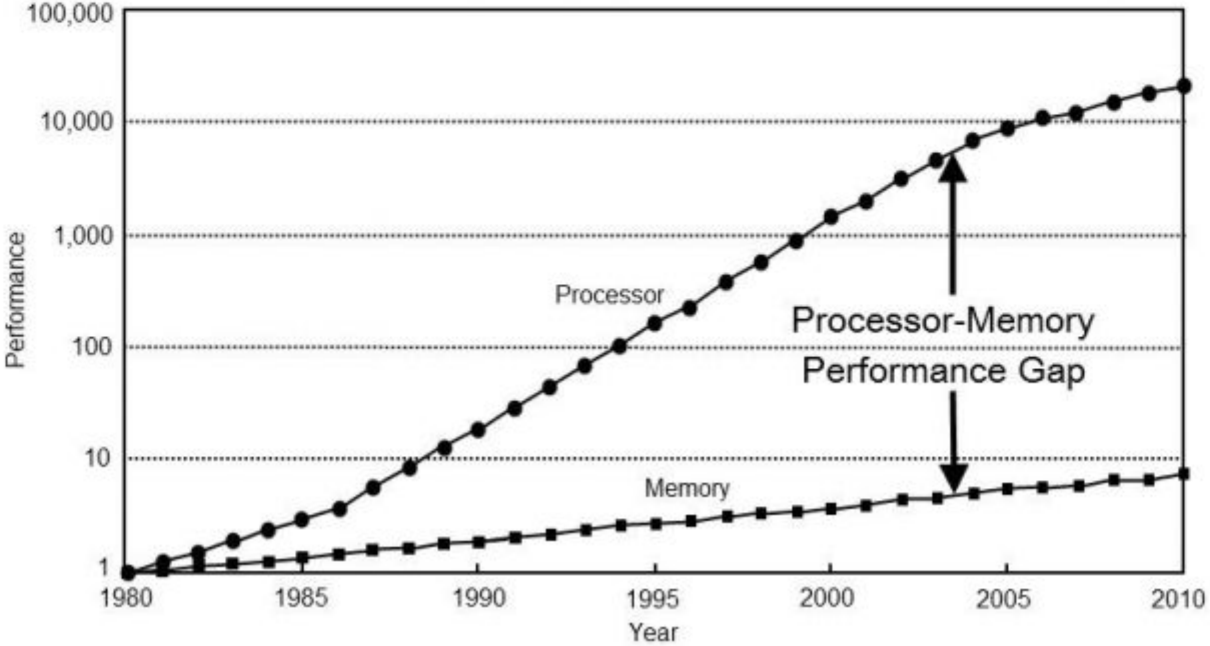
Stop me and ask (or ask at the end, or ask offline)

Spotted a mistake? Do not hesitate point’n out... Thanks! ;-)

CPU, Memory, Caches, Pipelines, Speculative Execution…

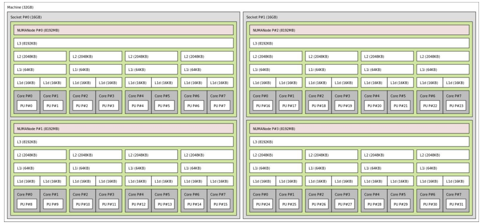
CPU, Memory

CPU are fast, memory is slow



What is Speculative Execution?

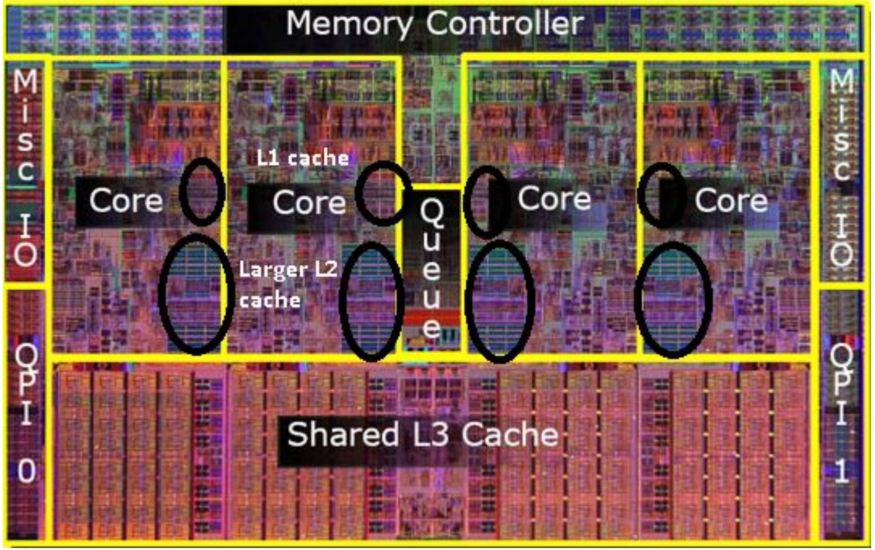
CPU, Memory, Cache(s)



Portable Hardware Locality (hwloc)

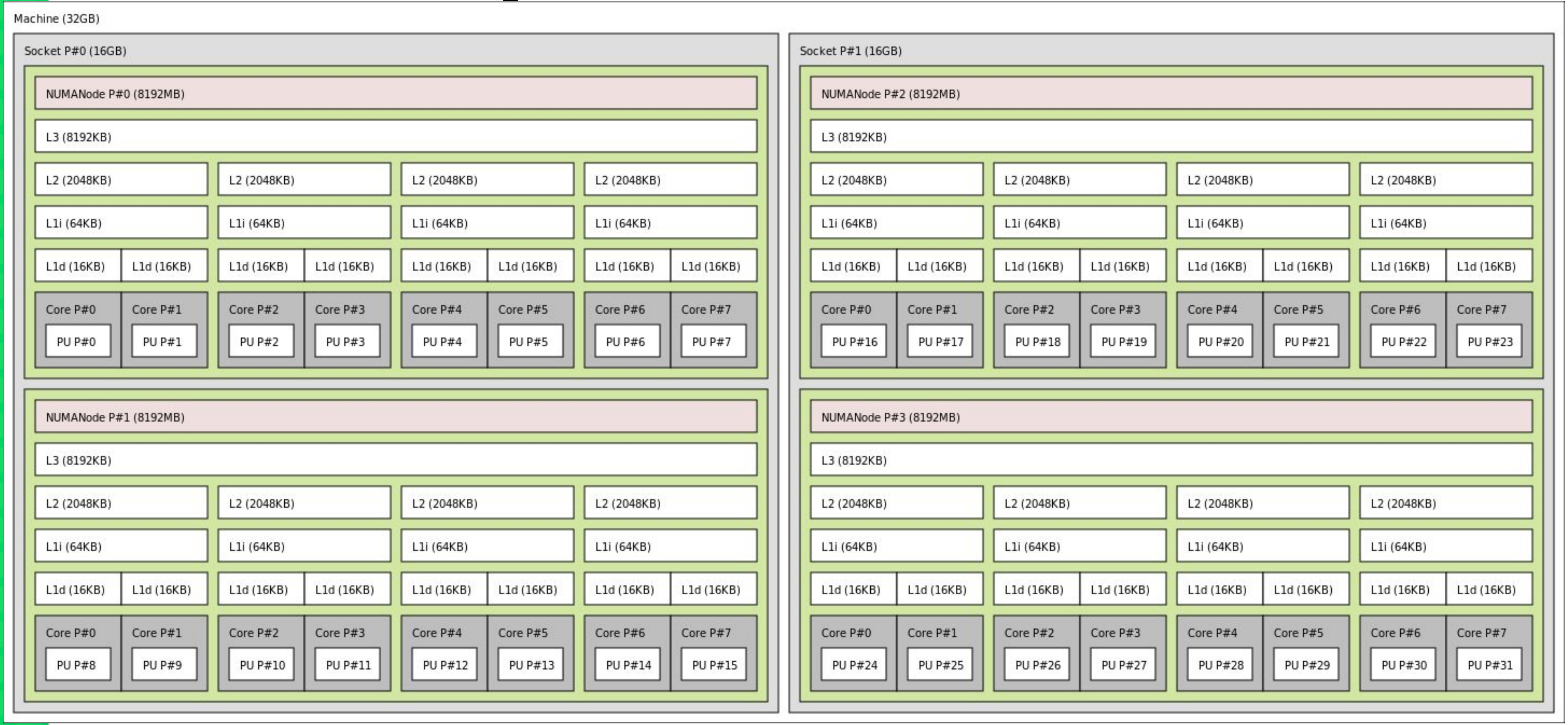
CPU are fast, memory is slow

- Cache == fast memory
- But we can't use it as main memory:
  - takes a lot of space on a chip
  - costs a lot of money
  - consumes a lot of power
  - ...
- On the CPU chip
  - takes most of the space in nowadays CPU CPUs, actually
- It's not cache, it's cache**s**
  - there's more than 1;
  - organized hierarchically

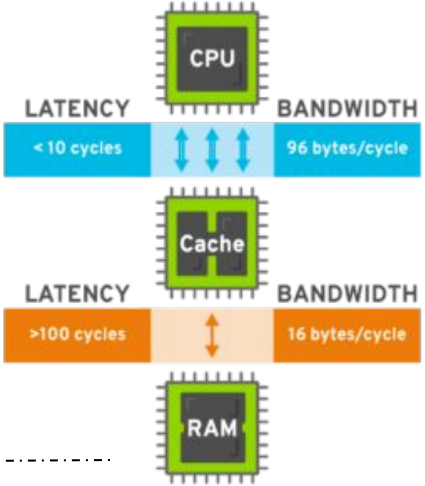


CPU, Memory, Cache(s)

Portable Hardware Locality (hwloc)



Cache(s): how *faster* are we talking about?



i7-6700 Skylake 4.0 GHz access latencies:

- L1 cache : **4 cycles** (direct pointer)
- L1 cache : 5 cycles (complex addr. calculations)
- L2 cache : 11 cycles
- L3 cache : 39 cycles
- Memory : **100 ~ 200 cycles**

Latency Numbers Every Programmer Should Know (do check this website!):

- L1 cache : **1 ns**
- L2 cache : 4 ns
- Memory : **100 ns**
- Tx 2KB, 1 Gbps network : 20,000 ns (20 µs)
- SSD random read : 150,000 ns (150 µs)
- Rotational disk seek : 10,000,000 ns (10 ms)

Cache(s): how *faster* are we talking about?

Real life parallelism:

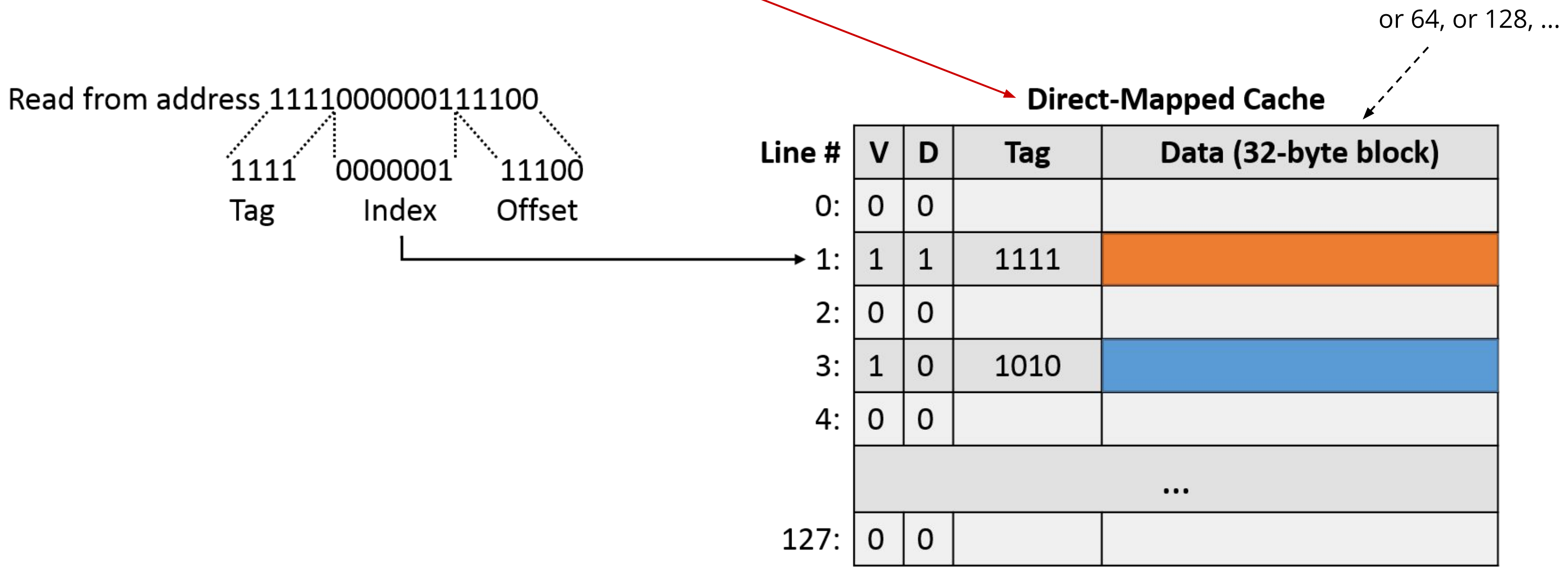
1 CPU Cycle	0.3 ns	1 s
Level 1 cache	0.9 ns	**3 s**
Level 2 cache	2.8 ns	9 s
Level 3 cache	12.9 ns	43 s
Memory	120 ns	**> 6 min**
SSD I/O	50 - 150 us	2-6 days
Rotational disk I/O	1-10 ms	1-12 months

Oh, and do check-out this video too!

http://sgros.blogspot.com/2014/08/memory-access-latencies.html

Caches: how do they work

- Address: splitted `[Index,Tag]`
- Lookup `Index`: gives you one or more tags ⇒ match your `Tag`



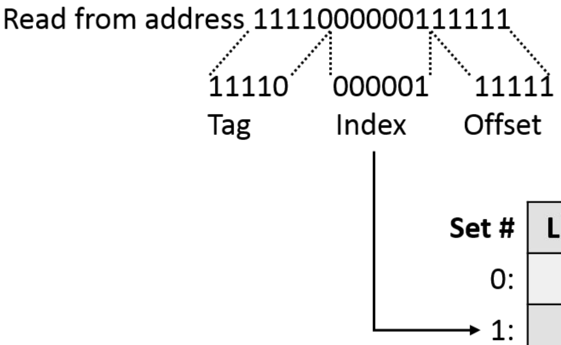
Read from address 1111000000111100

Direct-Mapped Cache

Line #	V	D	Tag	Data (32-byte block)
0:	0	0		
1:	1	1	1111	🟧
2:	0	0		
3:	1	0	1010	🟦
4:	0	0		
	...			
127:	0	0		

Caches: how do they work

- Address: splitted `[Index, Tag]`
- Lookup `Index`: gives you one or more tags ⇒ match your `Tag`



Two-Way Set Associative Cache

Set #	LRU	V_0	D_0	Tag_0	$Data_0$	V_1	D_1	Tag_1	$Data_1$
0:	0	0	0			0	0		
1:	1	1	1	11110	🟦	1	0	00110	🟧
2:	0	0	0			0	0		
3:	0	0	0			0	0		
4:	0	0	0			0	0		
	...								
63:	0	0	0			0	0		

CPU, Memory, Cache(s), TLB(s)

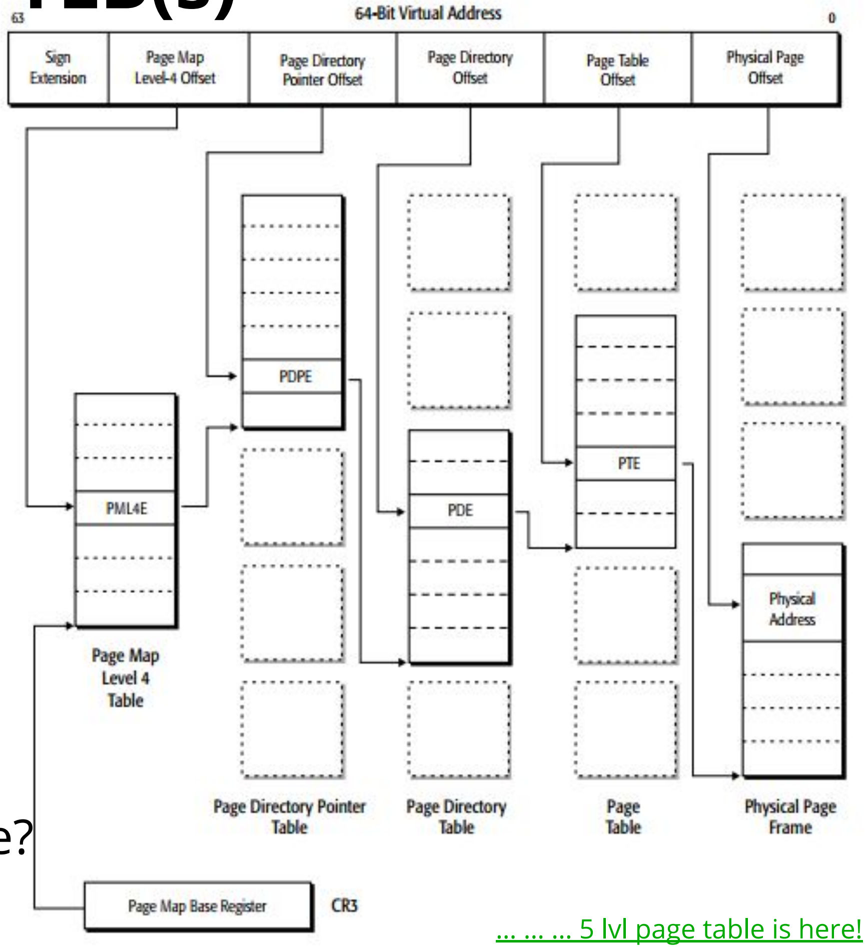
CPU are fast, memory is slow

- Even with caches

CPU, Memory, Cache(s), TLB(s)

Virtual Memory

- Address: virtual ⇒ physical, translated ~~via a table~~
- … via a **set of tables** (we want it sparse!)
- Address split: `[L4off,L3off,L2off,L1off,off]`
- Page Table:
  - Setup by CPU within MMU
  - Translation done by MMU, *walking* the page table
  - A walk for each memory reference? **No! No! No! No!**



… … … 5 lvl page table is here!

CPU, Memory, Cache(s), TLB(s)

Hierarchy of TLBs

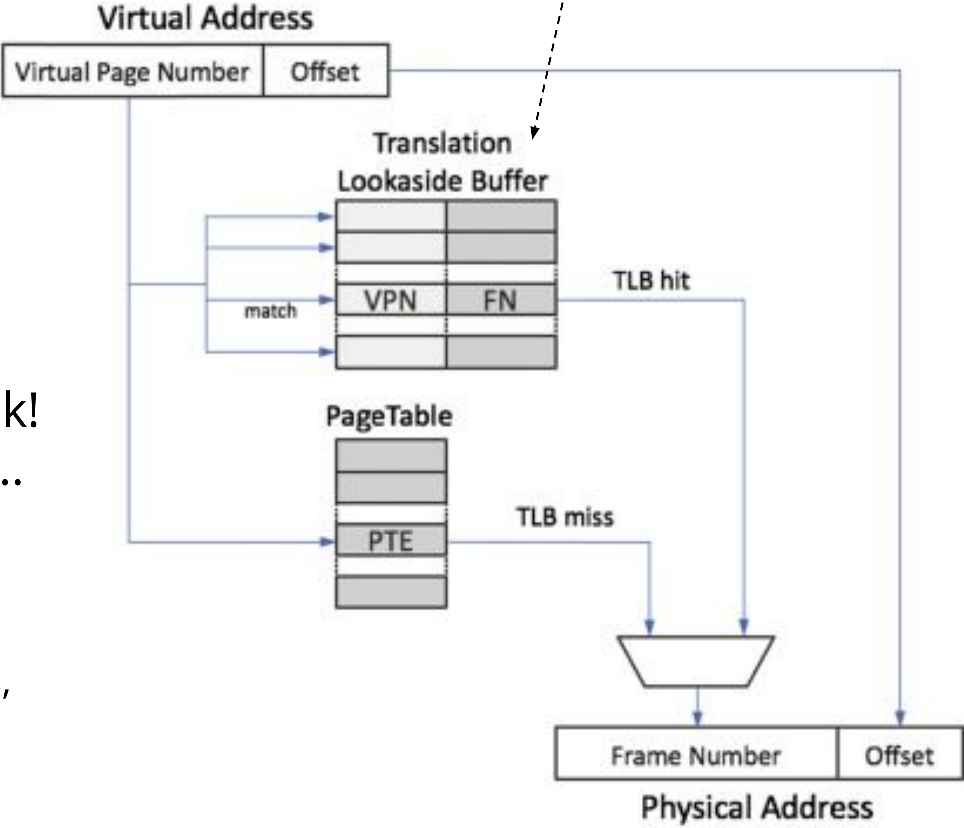
- Instruction L1 TLB
- Data L1 TLB
- I+D L2 TLB (called STLB)

Transitional Lookaside Buffer (TLB)

- A cache for virtual address translations
- On memory reference, check TLB:
  - **Hit:** we saved a page table walk!
  - **Miss:** page table walk needed...

Latency:

- TLB hit: ~ cache hit, 4 cycles / 4 ns
- Page Table Walk: 4~5 memory accesses, 100 cycles / 100ns **each!**



Superscalar Architectures

CPU are fast, memory is slow

- Even with caches
- Even with TLBs

Superscalar Architectures

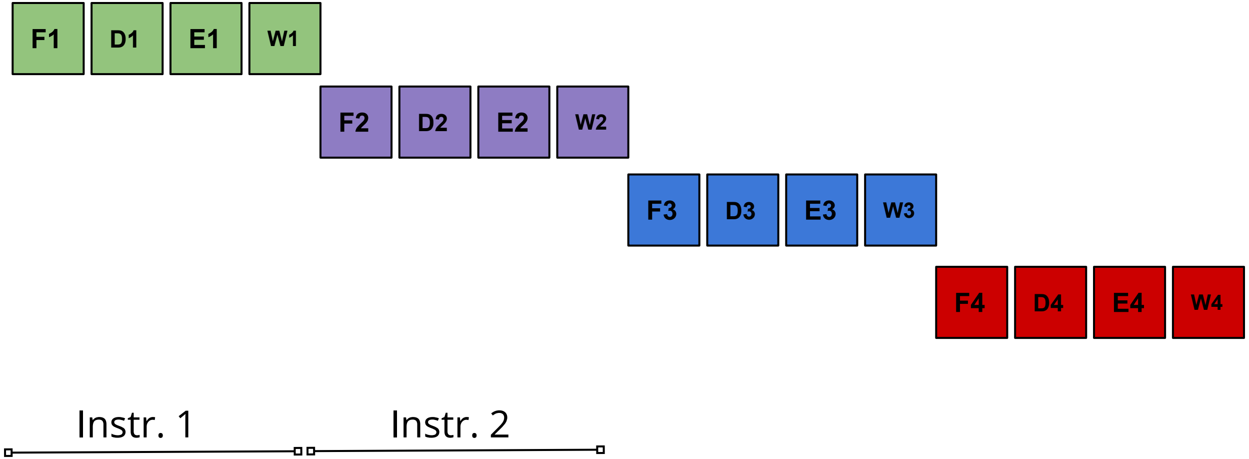
CPU executing an instruction:

F D E W

- **F:** fetch the instruction from memory/cache
- **D:** decode instruction:
  E.g., 01101101b == ADD %eax,*(%ebx)
- **E:** execute instruction
  do it. E.g., do the add, in CPU’s ALU, etc
- **W:** write result back
  update actual registers & caches/memory locations

Superscalar Architectures

CPU executing multiple instructions:



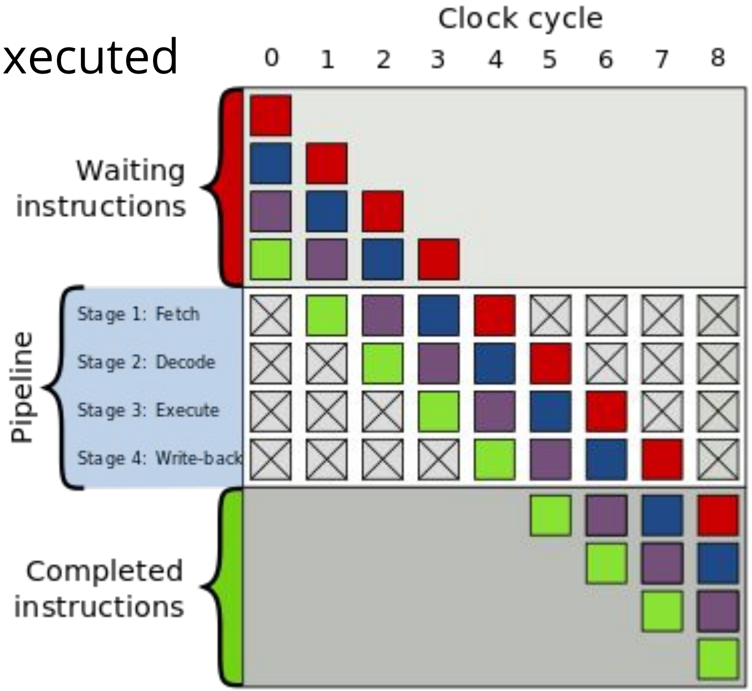
One after the other… … … **slow**! :-/

Somewhat there (pipelining), even in "sixty-five-oh-two" or "six-five-oh-two", the CPU of Apple II and Comodore VIC-20, back in 1975

Superscalar Architectures: pipelining

In-order execution, pipelined

- 0: Four instructions are waiting to be executed
- 1: green enters pipeline (e.g., IF)
- …
- 4: pipeline full
  4 stages ⇒ 4 inst. In flight
- 5: green completed
- …
- 8: all completed

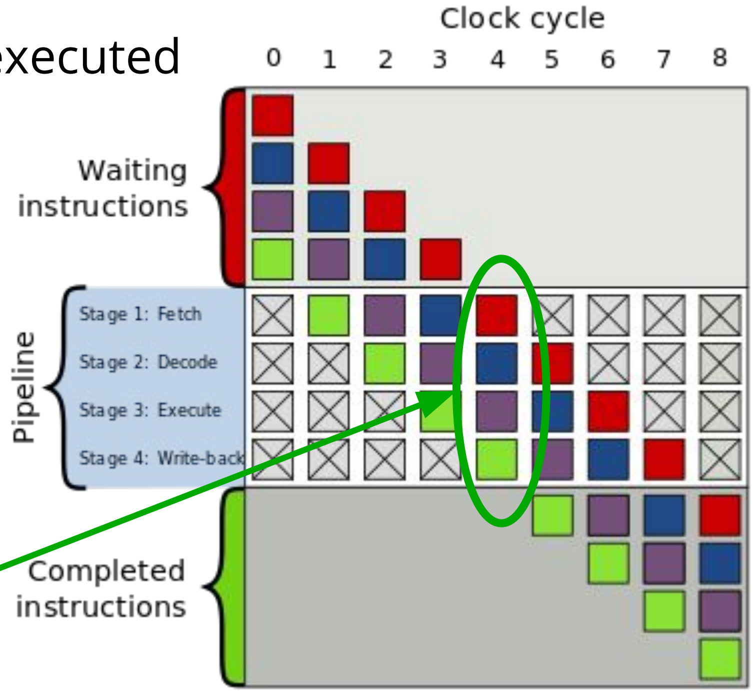


Wikipedia: Instruction Pipelining

Superscalar Architectures: pipelining

In-order execution, pipelined

- 0: Four instructions are waiting to be executed
- 1: green enters pipeline (e.g., IF)
- …
- 4: pipeline full
  4 stages ⇒ 4 inst. In flight
- 5: green completed
- …
- 8: all completed



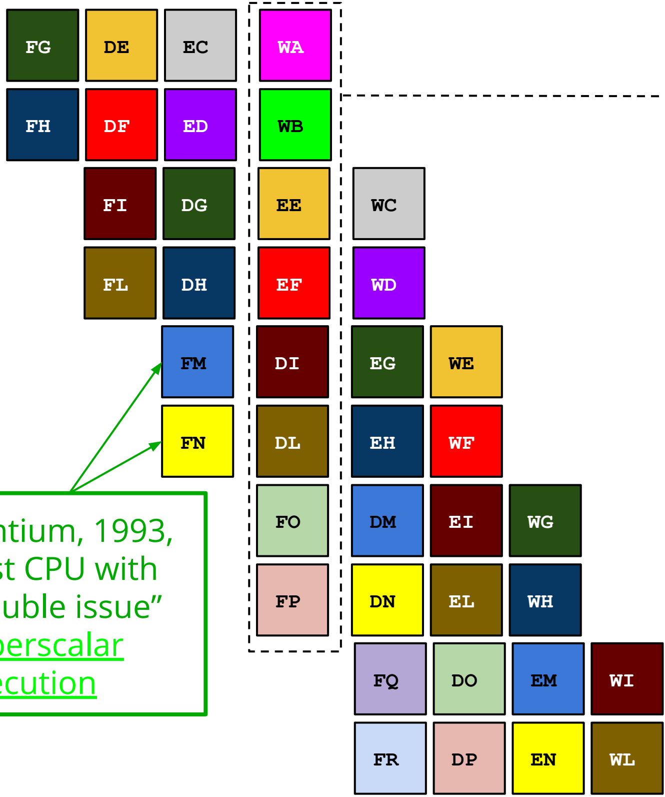
Instruction Level Parallelism

Wikipedia: Instruction Pipelining

King of there back in 1966 already!

Superscalar Architectures: n-th issue

Double the game, ~~double ILP~~, increase ILP



Pentium, 1993, First CPU with “double issue” superscalar execution

4 pipeline stages, 8 instructions in flight:

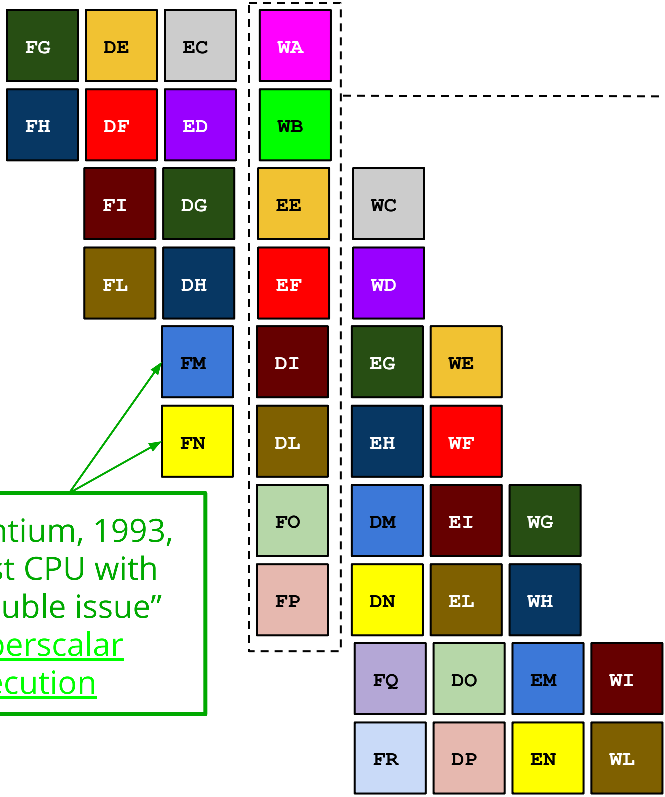
1. **Instr. A**, **instr. B**: write-back
2. E, F: execute
3. I, L: decode
4. F, P: fetch

(… … … theoretically)

King of there back in 1966 already!

Superscalar Architectures: n-th issue

Double the game, ~~double ILP~~, increase ILP



FG	DE	EC	WA				
FH	DF	ED	WB				
	FI	DG	EE	WC			
	FL	DH	EF	WD			
		FM	DI	EG	WE		
		FN	DL	EH	WF		
			FO	DM	EI	WG	
			FP	DN	EL	WH	
				FQ	DO	EM	WI
				FR	DP	EN	WL

4 pipeline stages, 8 instructions in flight:

1. **Instr. A**, **instr. B**: write-back
2. **E**, **F**: execute
3. **I**, **L**: decode
4. **F**, **P**: fetch

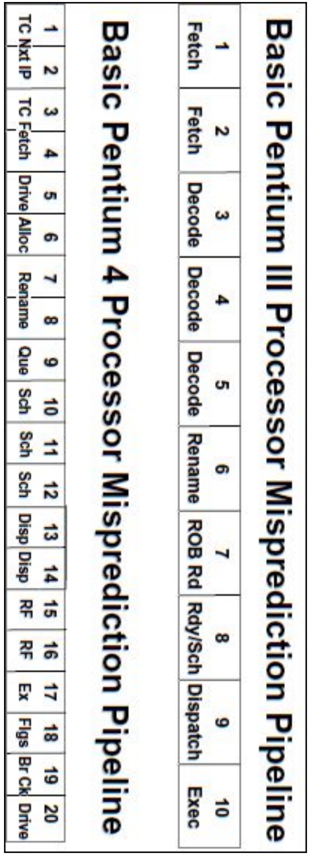
But don't go too far… Or you'll get Itanium! Explicitly parallel instruction computing / Very long instruction word)

Pentium, 1993, First CPU with "double issue" superscalar execution

Superscalar Architectures: deeper pipes

The *smaller* the **stage**, the *faster* **clock** can run

- 486 (1989), 3 stages 100 MH
- P5 [Pentium] (1993), 5 stages, 300 MHz
- P6 [Pentium Pro, Pentium II, Pentium III] (1995-1999), 12-14 stages, 450 MHz-1.4 GHz
- NetBurst, Prescott [Pentium4] (2000-2004), 20-31 stages, 2.0-3.8 GHz
- Core (2006), 12 stages, 3.0 GHz
- Nehalem (2008), 20 stages, 3.6 GHz
- Sandy Bridge, Ivy Bridge (2011-2012) 16, 4 GHz
- Skylake (2015), 16 stages, 4.2 Ghz
- Kaby Lake, Coffee Lake (2016-2017), 16 stages, 4.5 GHz
- Cannon Lake (2018), 16 stages, 4.2 GHz



Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures

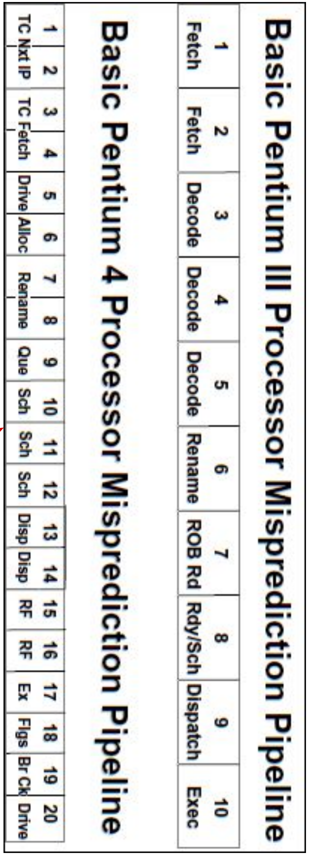
Superscalar Architectures: deeper pipes

The *smaller* the **stage**, the *faster* **clock** can run

- 486 (1989), 3 stages 100 MH
- P5 [Pentium] (1993), 5 stages, 300 MHz
- P6 [Pentium Pro, Pentium II, Pentium III] (1995-1999), 12-14 stages, 450 MHz-1.4 GHz
- NetBurst, Prescott [Pentium4] (2000-2004), 20-31 stages, 2.0-3.8 GHz
- Core (2006), 12 stages, 3.0 GHz
- Nehalem (2008), 20 stages, 3.6 GHz
- Sandy Bridge, Ivy Bridge (2011-2012) 16, 4 GHz
- Skylake (2015), 16 stages, 4.2 Ghz
- Kaby Lake, Coffee Lake (2016-2017), 16 stages, 4.5 GHz
- Cannon Lake (2018), 16 stages, 4.2 GHz

https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures

But don't go too far… or you'll get Pentium4!



Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

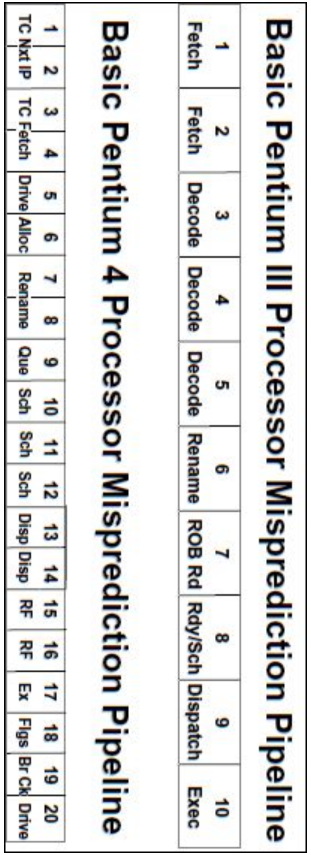
Superscalar Architectures: deeper pipes

The *smaller* the **stage**, the *faster* **clock** can run

- 486 (1989), 3 stages 100 MH
- P5 [Pentium] (1993), 5 stages, 300 MHz
- P6 [Pentium Pro, Pentium II, Pentium III] (1995-1999), 12-14 stages, 450 MHz-1.4 GHz
- NetBurst, Prescott [Pentium4] (2000- stages, 2.0-3.8 GHz
- Core (2006), 12 stages, 3.0 GHz
- Nehalem (2008), 20 stages, 3.6 GHz
- Sandy Bridge, Ivy Bridge (2011-2012) 16, 4 GHz
- Skylake (2015), 16 stages, 4.2 Ghz
- Kaby Lake, Coffee Lake (2016-2017), 16 stages, 4.5 GHz
- Cannon Lake (2018), 16 stages, 4.2 GHz

Current processors

https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures



Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Out-of-Order Execution

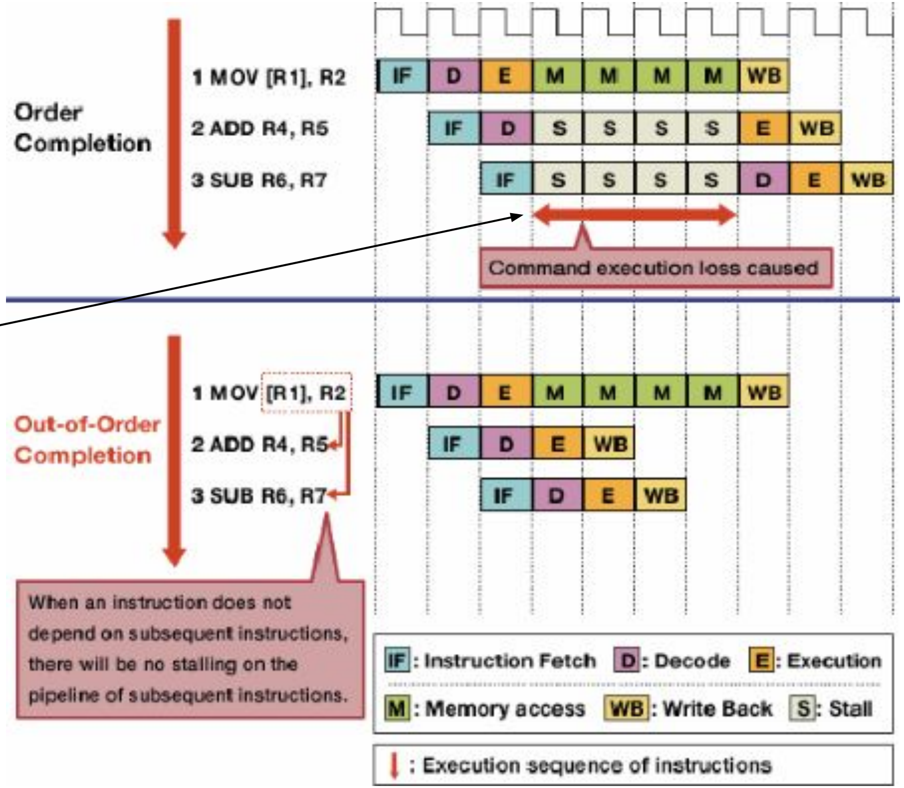
CPU are fast, memory is slow

- Even with caches
- Even with TLBs
- Even with pipeline

Out-of-Order Execution

In-order execution, pipelined

- Instructions takes variable amount of time
- If an (phase of an) instruction takes a lot of time? **Stalls / bubbles**
- Could have I done something else while waiting? **YES!**



Pipeline and out-of-order instruction execution optimize performance

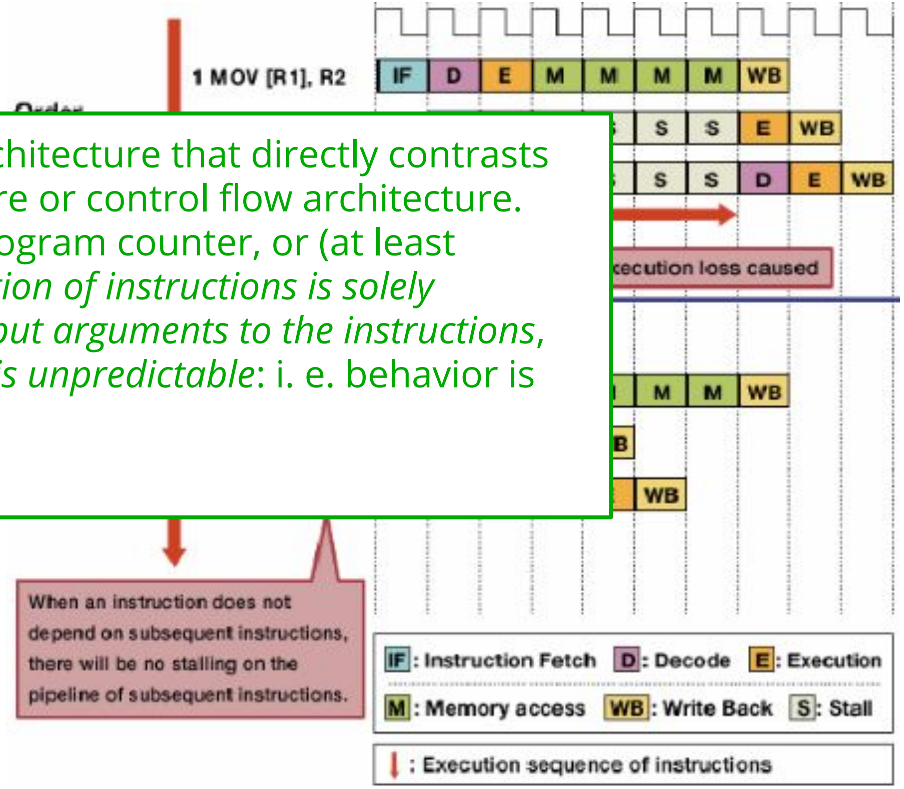
But **not** delay slots! From (old) RISCs, right now only popular in some DSP (probably)

Out-of-Order Execution

In-order execution, pipelined

- In… ar…
- If… ta… **St…**
- C… else while waiting? **YES.**

"Dataflow architecture is a computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. Dataflow architectures do not have a program counter, or (at least conceptually) *the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions*, so that *the order of instruction execution is unpredictable*: i. e. behavior is nondeterministic."



When an instruction does not depend on subsequent instructions, there will be no stalling on the pipeline of subsequent instructions.

Pipeline and out-of-order instruction execution optimize performance

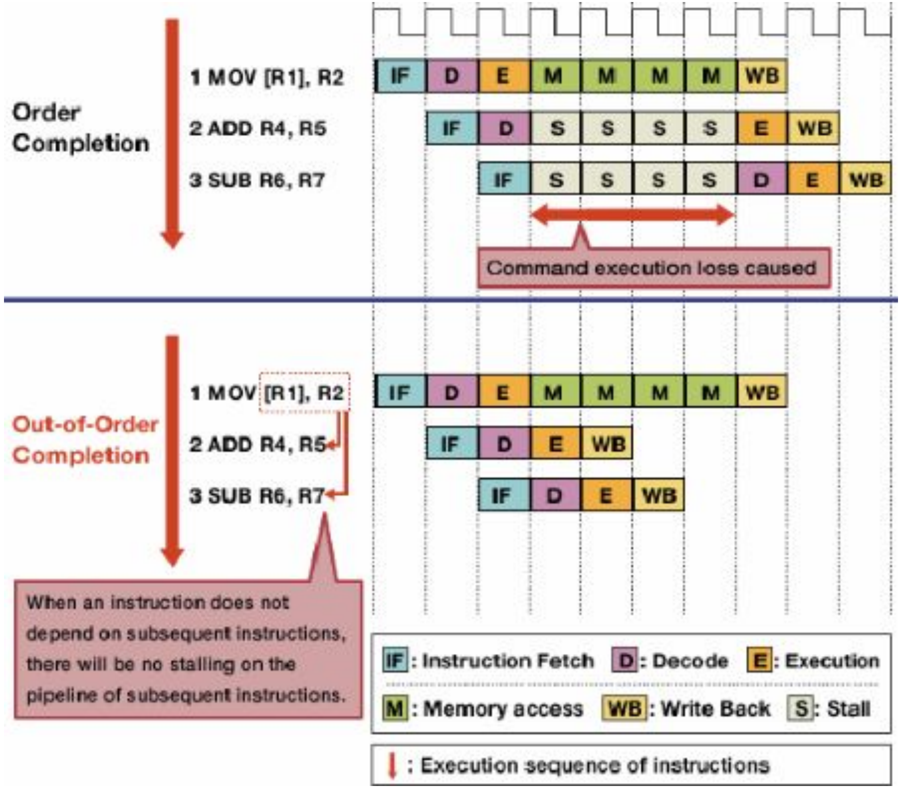
But **not** delay slots! From (old) RISCs, right now only popular in some DSP (probably)

Out-of-Order Execution

1. Fetch **a bunch** of instructions; **stash** them in a queue (Reservation Station)
2. Fetch operands, e.g., from memory
2. Execute instructions from the queue with operands ready ⇒ issued to the appropriate stage
3. Instructions leaves queue (might be before “earlier” instructions) ⇒ results **queued** (Reorder Buffer, ROB)
4. Instruction completes (retires) **after** all earlier instructions also completed

in parallel!

Tomasulo algorithm, IBM, 1967
⇒ adopted by Pentium Pro (P6 family), 1995



Pipeline and out-of-order instruction execution optimize performance

Out-of-Order Execution

In Order

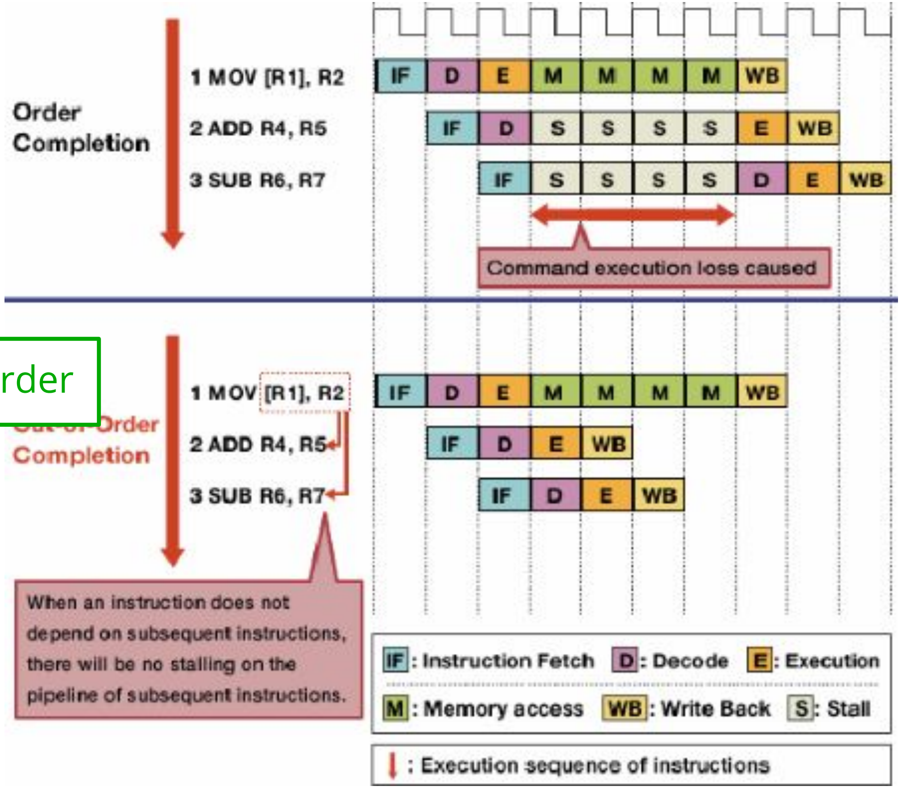
in parallel!

1. Fetch **a bunch** of instructions; **stash** them in a queue (Reservation Station)
2. Fetch operands (e.g., from memory)
2. Execute instructions from the queue with operands ready ⇒ issued to the appropriate stage
3. Instructions leaves queue (might before "earlier" instructions) ⇒ results **queued** (Reorder Bufffer, ROB)
4. Instruction completes (retires) **after** all earlier instructions also completed

Out of Order

Tomasulo algorithm, IBM, 1967
⇒ adopted by Pentium Pro (P6 family), 1995

In Order



Pipeline and out-of-order instruction execution optimize performance

Speculative Execution

CPU are fast, memory is slow

- Even with caches
- Even with TLBs
- Even with pipeline
- Even with out-of-order execution

Speculative Execution

CPU are fast, memory is slow

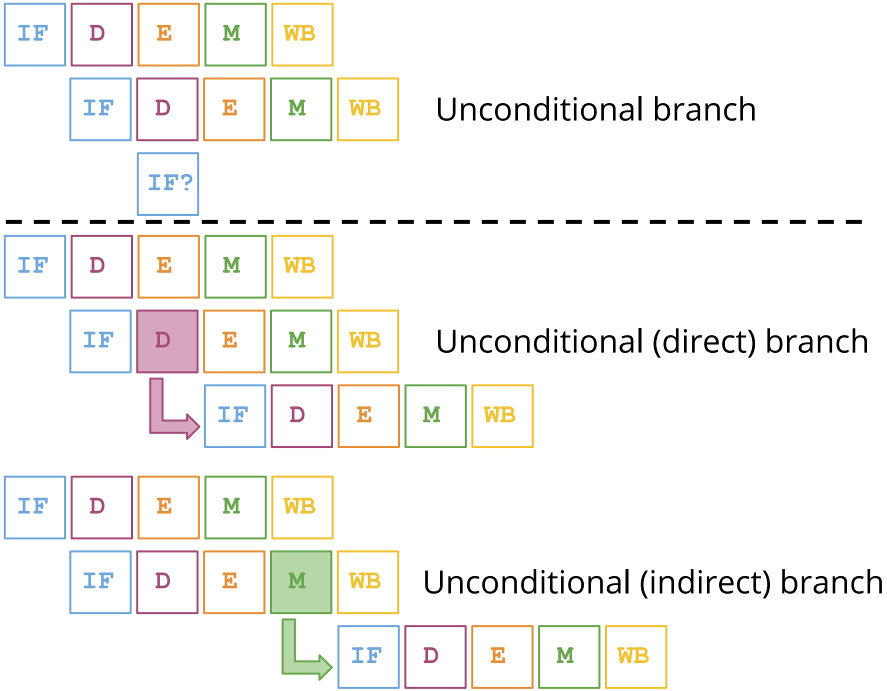
- Even with caches
- Even with TLBs
- Even with pipeline
- Even with out-of-order execution

How come we’re still in trouble?

- Branches: `if`, loops, function calls/returns …
- Out of Order Exec. works **great** for *data-dependencies*
- Branches are “control flow-dependencies”
  - If I don’t know what I’ll execute next, I can’t reorder instructions!

Branches

Unconditional branches (func. call, func. ret, `jmp`, …)



IF D E M WB

IF D E M WB Unconditional branch

IF?

IF D E M WB

IF D E M WB Unconditional (direct) branch

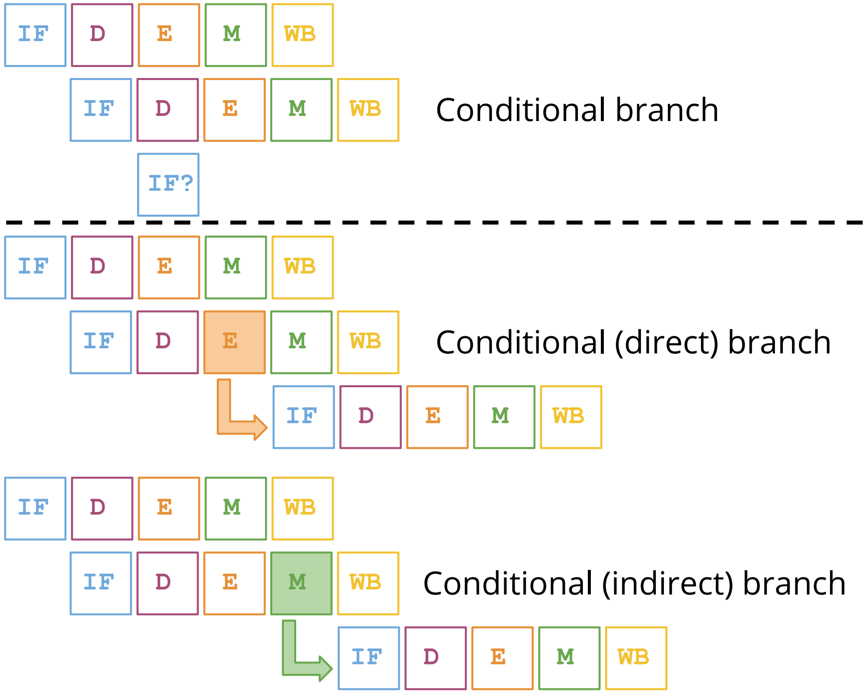
IF D E M WB

IF D E M WB

IF D E M WB Unconditional (indirect) branch

IF D E M WB

Conditional branches (if, loops, …)



IF D E M WB

IF D E M WB Conditional branch

IF?

IF D E M WB

IF D E M WB Conditional (direct) branch

IF D E M WB

IF D E M WB

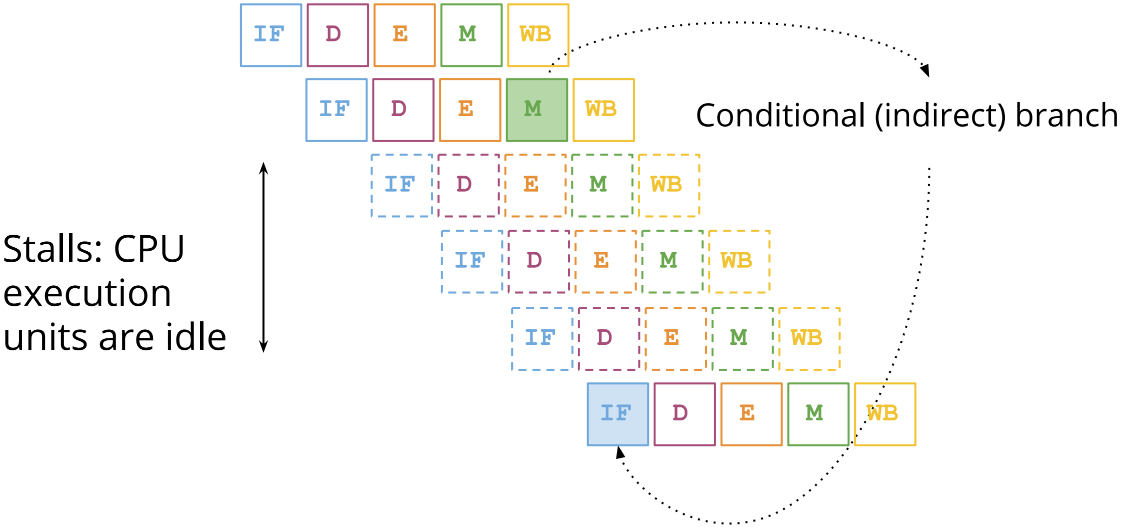
IF D E M WB Conditional (indirect) branch

IF D E M WB

Out-of-Order + Speculative Execution

Yeah, whatever!! Reorder buffer is there, let’s use it...

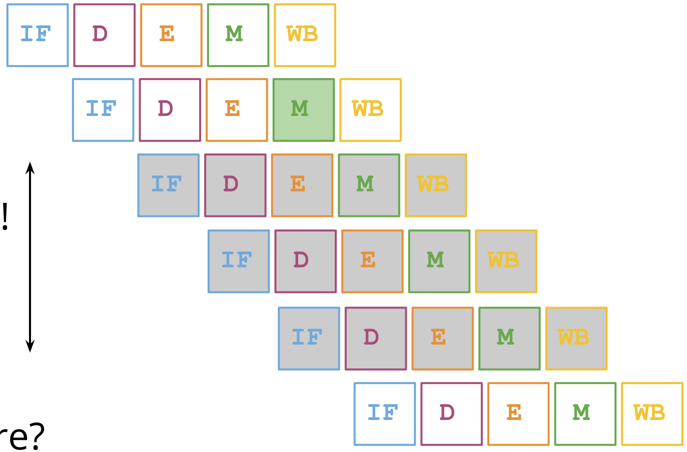
- Ignore control-flow dependencies: execute instructions anyway
- We “occupy” **stalls**, so we’re no any slower!


IF
D
E
M
WB
Conditional (indirect) branch
Stalls: CPU execution units are idle

Out-of-Order + Speculative Execution

Yeah, whatever!! Reorder buffer is there, let’s use it…

- Ignore control-flow dependencies: execute instructions anyway
- We “occupy” **stalls**, so we’re no any slower!

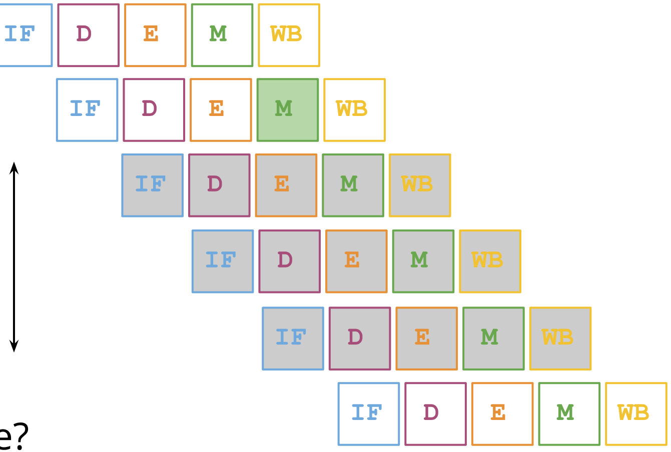


Actual instructions!
…
…
…
But which ones?
Fetched from where?
etc.

Out-of-Order + Speculative Execution

Yeah, whatever!! Reorder buffer is there, let’s use it…

- Ignore control-flow dependencies: **execute** instructions anyway
- We “occupy” **stalls**, so we’re no any slower!



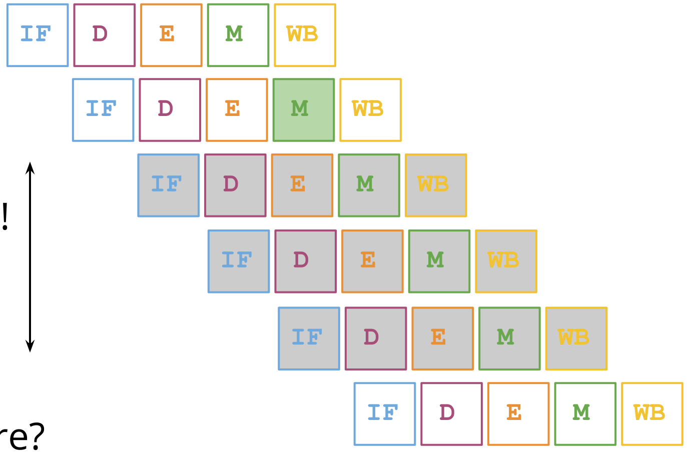
Actual instructions!
…
…
…
But which ones?
Fetched from where?
etc.

Whatever instructions they are, I’m not any slower

Out-of-Order + Speculative Execution

Yeah, whatever!! Reorder buffer is there, let's use it...

- Ignore control-flow dependencies: **execute** instructions anyway
- We "occupy" **stalls**, so we're no any slower!



Actual instructions!
…
…
…
But which ones?
Fetched from where?
etc.

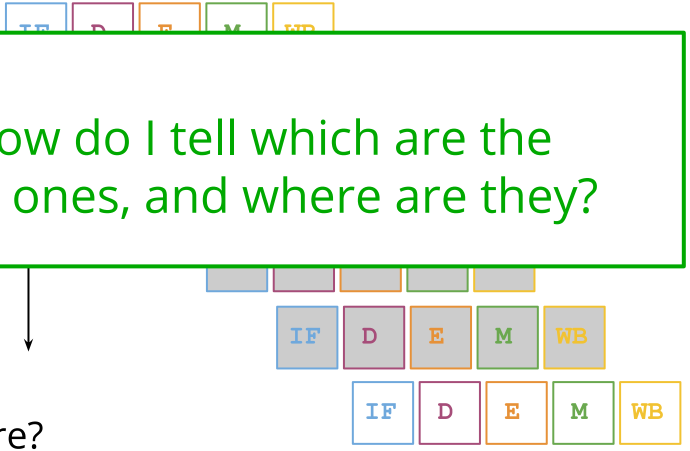
Whatever instructions they are, I'm not any slower

If they could be the **right** ones, I'll be faster!!!

Out-of-Order + Speculative Execution

Yeah, whatever!! Reorder buffer is there, let’s use it…

- Ignore **control-flow dependencies**: execute instructions anyway
- We “occupy” **stalls**, so we’re no any slower!



But:
Q: How do I tell which are the right ones, and where are they?

Actual instr
…
…
…
But which ones?
Fetched from where?
etc.

Whatever instructions they are, I’m not any slower

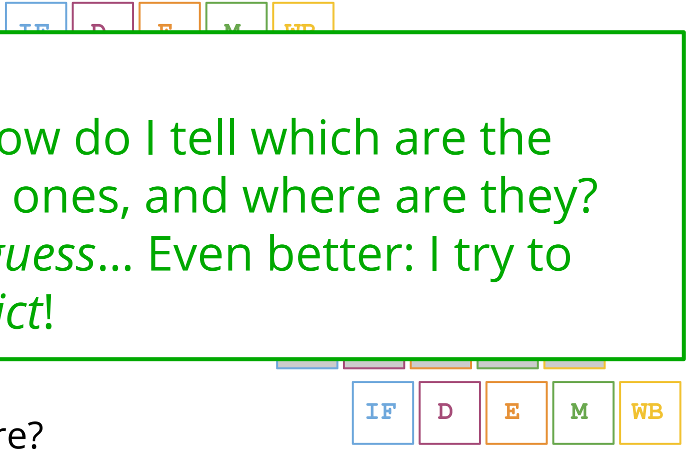
If they could be the **right** ones, I’ll be faster!!!

Out-of-Order + Speculative Execution

Yeah, whatever!! Reorder buffer is there, let's use it…

- ~~Ignore~~ **Guess** control-flow dependencies: execute instructions anyway
- We "occupy" **stalls**, so we're no any slower!

Actual instr
…
…
…
But which ones?
Fetched from where?
etc.



But:
Q: How do I tell which are the right ones, and where are they?
A: I *guess*… Even better: I try to *predict*!

Whatever instructions they are, I'm not any slower

If they could be the **right** ones, I'll be faster!!!

Out-of-Order + Speculative Execution

Yeah, whatever!! Reorder buffer is there, let’s use it…

- ~~Ignore~~ **Guess** control-flow dependencies: execute instructions anyway
- We “occupy” **stalls**, so we’re no any slower!

Actual instr
…
…
…
But which o
Fetched fro
etc.

But:
Q: How do I tell which are the right ones, and where are they?
A: I *guess*… Even better: I try to *predict*!
Q: Ok, cool! But wait, what if you guess wrong (mispredict)? :-O

Whatever instructions they are, I’m not any slower

If they could be the **right** ones, I’ll be faster!!!

Out-of-Order + Speculative Execution

Q: Ok, cool! But wait, what if you guess wrong (mispredict)? :-O

Yeah, whatever!! Reorder buffer is there, l

- ~~Ignore~~ **Guess** control-flow dependencies: **execute** instructions anyway
- We “occupy” stalls, so we’re no any slower!

Execute them *speculatively*:

- Execute them but defer (some of their) effects
- Until we know whether they’ll run “for real”
  - If yes, apply the effects (memory/register writes, exceptions, ...)
  - If no, throw everything away

Out-of-Order + Speculative Execution

Q: Ok, cool! But wait, what if you guess wrong (mispredict)? :-O

Yeah, whatever!! Reorder buffer is there, l

- ~~Ignore~~ **Guess** control-flow dependencies: **execute** instructions anyway
- We “occupy” stalls, so we’re no any slower!

Execute them *speculatively*:

- Execute them but defer (some of their) effects
- Until we know whether they’ll run “for real”
  – If yes, apply the effects (memory/register writes, **exceptions**, …)
  – If no, throw everything away

Branch Prediction

How do we guess:

1. Whether or not a *conditional branch* (if, loops) will be taken or not taken
2. Where or not an *unconditional direct* branch (function call, function return) or an *unconditional indirect* branch (function pointer) branch will be taken or not taken

We can’t. We can predict (e.g., basing on previous history):

1. Branch predictors
2. Branch Target Buffer, Return Stack Buffer

Branch Prediction

How do we guess **direct branches** (if, loops)

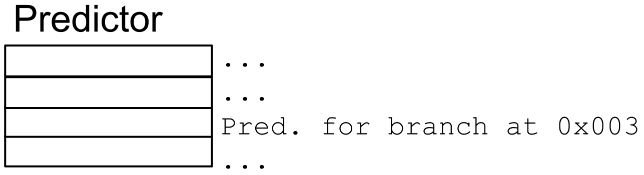
- Static prediction: no runtime knowledge
  - Always taken (loops! + loops execute multiple times… by definition!): 70% correct
  - Backward taken, forward not taken (BTFNT), (loops again! + compiler help), PPC 601 (1993): 80% correct
- Dynamic prediction:
  look at history, at runtime
  - 1 bit history, predict basing on last occurrence, DEC/MIPS (1992/1994): 85% correct
  - 2 bit history, “often taken” == always taken, Pentium (1993): 90% correct
  - Store history, ‘100100’ == taken once every 3 times, Pentium II (1997): 93% correct
  - Multilevel “agree” predictor, PA-RISC (2001): 95% correct
  - Neural networks, AMD Zen/Bulldozer (2001)
  - Geometric predictor, predictor chaining, Intel (2006)

Kernel Recipes 2018 - Meltdown and Spectre: seeing through the magician’s tricks
http://danluu.com/branch-prediction/

Branch Prediction

How do we guess **direct branches** (if, loops)

```
0x002 if (A)
0x003   do_A()
0x004 do_notA()
      ...
      ...
      ...
```



Predictor

...

...

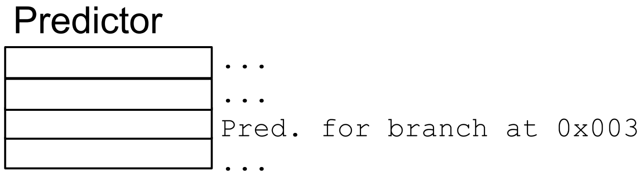
Pred. for branch at 0x003

...

Branch Prediction

How do we guess **direct branches** (if, loops)

```
0x002 if (A)
0x003   do_A()
0x004 do_notA()
      ...
      ...
      ...
```



Predictor

...

...

Pred. for branch at 0x003

...

CPU: branch taken, no prediction

```
begin[if (A)]
...
...
end[if (A)] == true, branch!
begin[do_A()]
...
end[do_A()]
```

Branch Prediction

How do we guess **direct branches** (if, loops)

```
0x002 if (A)       //true
0x003   do_A()
0x004 do_notA()
      ...
      ...
      ...
```

Predictor

	...
	...
1 = taken	Pred. for branch at 0x002
	...

CPU: branch taken, no prediction

```
begin[if (A)]
...
...
end[if (A)] == true, branch!
begin[do_A()]
...
end[do_A()]
```

CPU: branch taken, *ok prediction*

```
begin[if (A)]
check_pred[0x002] ⇐ taken
spec_begin[do_A()]
end[if (A)] == true, branch!
...
spec_commit[do_A()] == end[do_A()]
...
```

Finished do_A() **earlier!**

Branch Prediction

How do we guess **direct branches** (if, loops)

```
0x002 if (A)      //true
0x003   do_A()
0x004 do_notA()
      ...
      ...
      ...
```

Predictor

	...
	...
0 = not taken	Pred. for branch at 0x002
	...

CPU: branch taken, no prediction

```
begin[if (A)]
...
...
end[if (A)] == true, branch!
begin[do_A()]
...
end[do_A()]
```

CPU: branch taken, *misprediction*

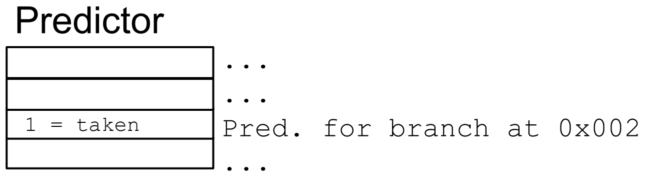
```
begin[if (A)]
check_pred[0x002] ⇐ not taken
spec_begin[do_not(A)]
end[if (A)] == true, branch!
begin[do_A()]
spec_undo[do_notA()]
end[do_A()]
```

Finished do_A() **no later**!

Branch Prediction

How do we guess **direct branches** (if, loops)

```
0x002 if (A)
0x003   do_A()
0x004 do_notA()
  ... ...
  ... ...
  ... ...
```



Predictor

...

...

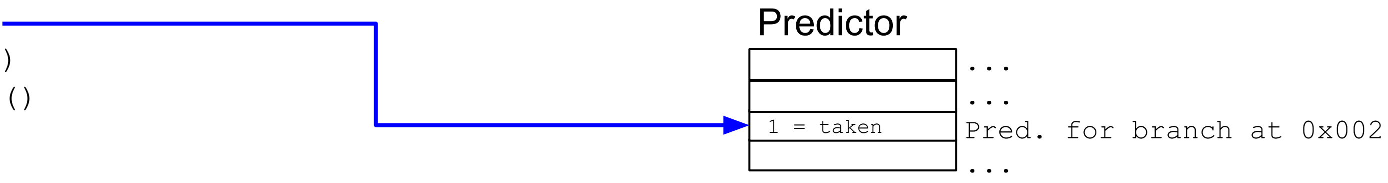
1 = taken Pred. for branch at 0x002

...

Branch Prediction

How do we guess **direct branches** (if, loops)

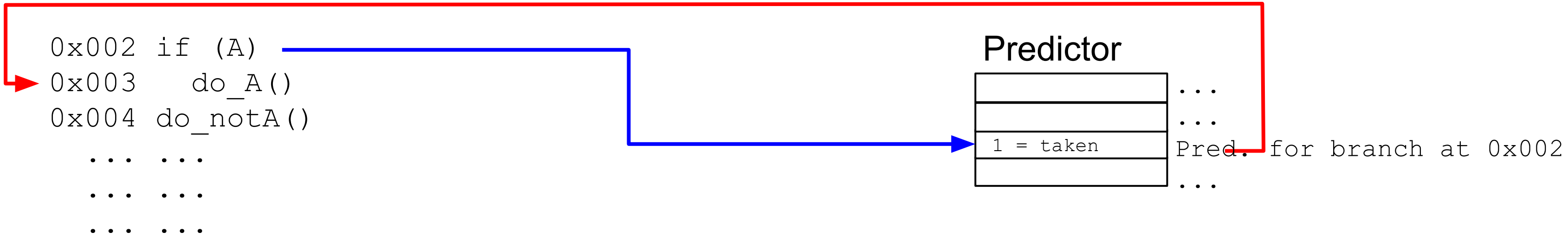
```
0x002 if (A)
0x003   do_A()
0x004 do_notA()
  ... ...
  ... ...
  ... ...
```



(1) Check predictor

Branch Prediction

How do we guess **direct branches** (if, loops)

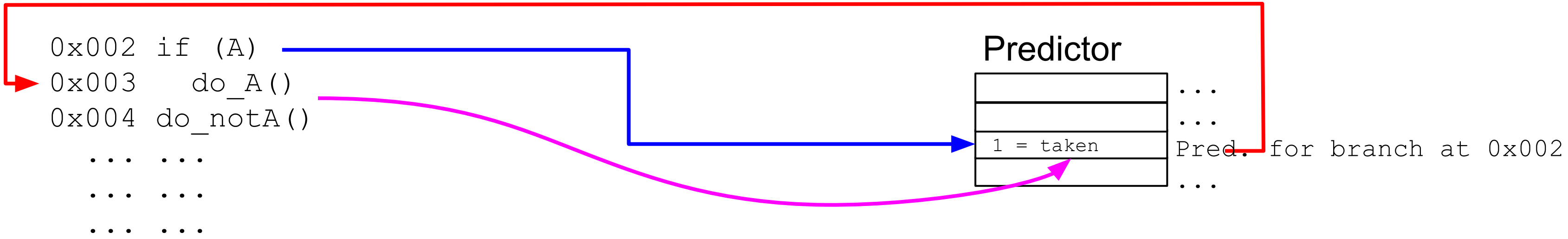


```
0x002 if (A)
0x003   do_A()
0x004 do_notA()
   ... ...
   ... ...
   ... ...
```

(1) Check predictor
(2) taken ⇒ speculatively execute do_A()

Branch Prediction

How do we guess **direct branches** (if, loops)


```
0x002 if (A)
0x003   do_A()
0x004 do_notA()
   ... ...
   ... ...
   ... ...
```

(1) Check predictor

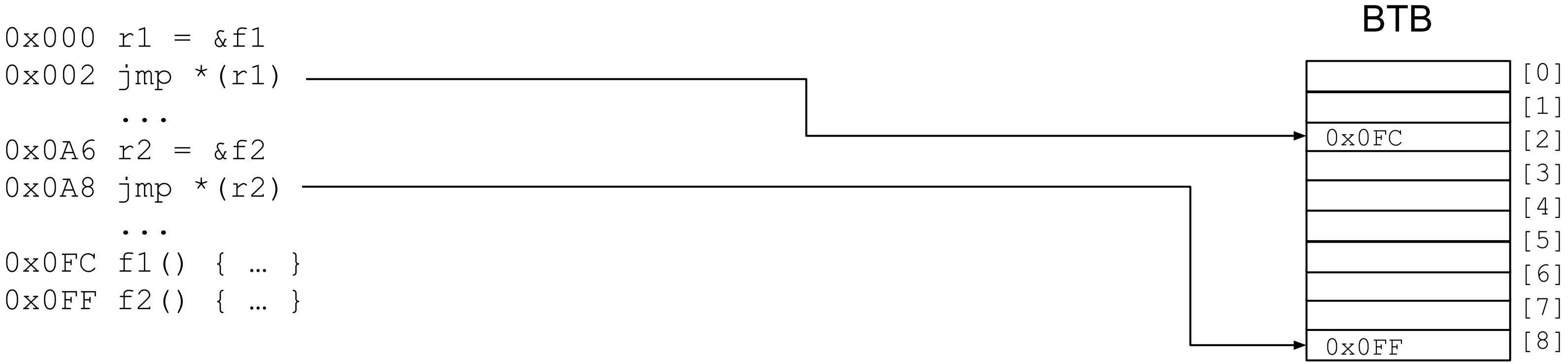
(2) taken ⇒ speculatively execute do_A()

(3) Update predictor (with what really happened)

Branch Prediction

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a branch cache

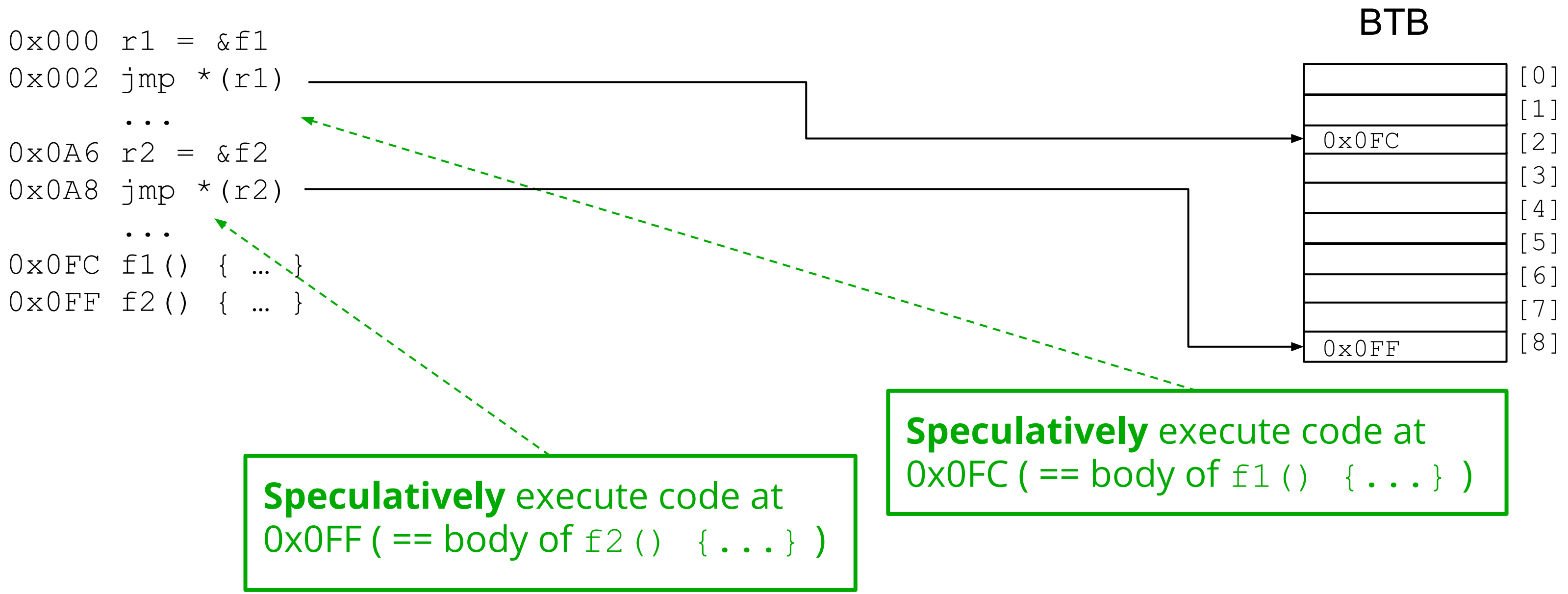


```
0x000 r1 = &f1
0x002 jmp *(r1)
      ...
0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
0x0FC f1() { … }
0x0FF f2() { … }
```

Branch Prediction

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a branch cache



```
0x000 r1 = &f1
0x002 jmp *(r1)
      ...
0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
0x0FC f1() { … }
0x0FF f2() { … }
```

BTB

	[0]
	[1]
0x0FC	[2]
	[3]
	[4]
	[5]
	[6]
	[7]
0x0FF	[8]

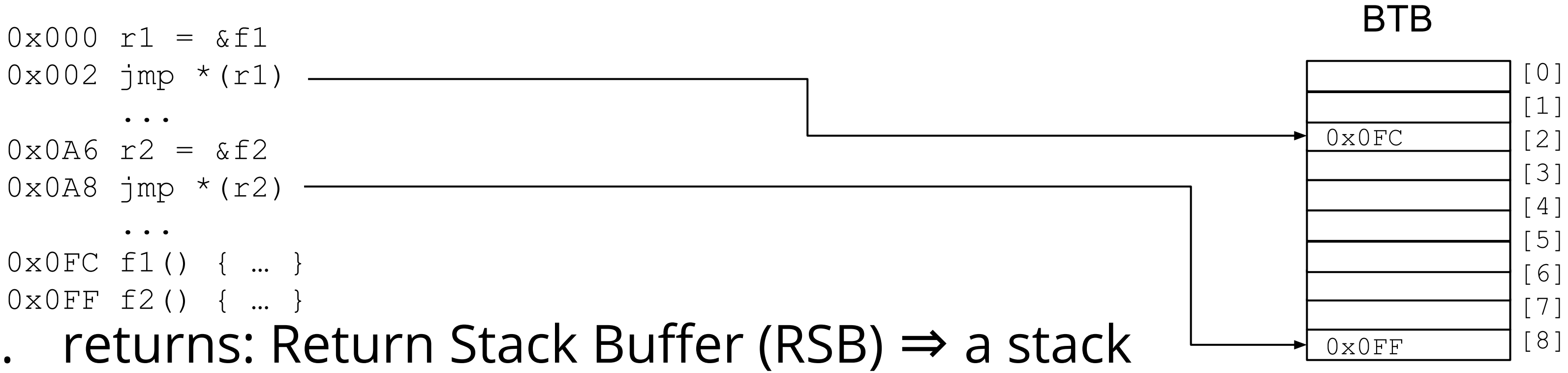
Speculatively execute code at 0x0FC (== body of `f1() {...}`)

Speculatively execute code at 0x0FF (== body of `f2() {...}`)

Branch Prediction

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a branch cache

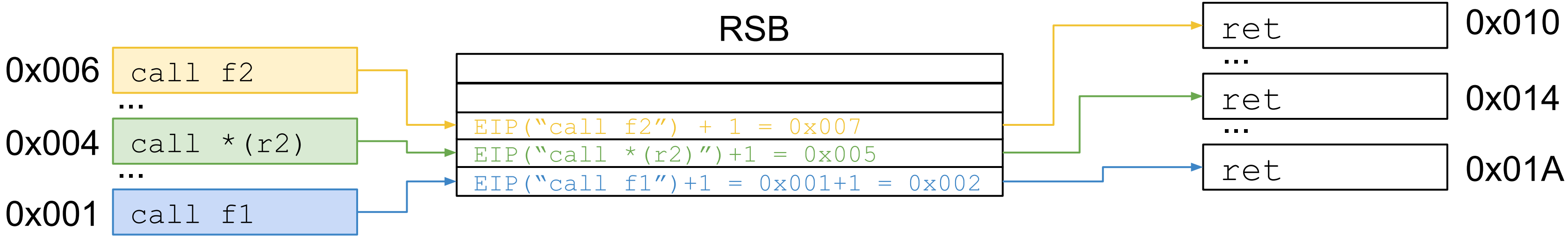


```
0x000 r1 = &f1
0x002 jmp *(r1)
      ...
0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
0x0FC f1() { … }
0x0FF f2() { … }
```

BTB

	[0]
	[1]
0x0FC	[2]
	[3]
	[4]
	[5]
	[6]
	[7]
0x0FF	[8]

2. returns: Return Stack Buffer (RSB) ⇒ a stack



0x006 call f2
…
0x004 call *(r2)
…
0x001 call f1

RSB

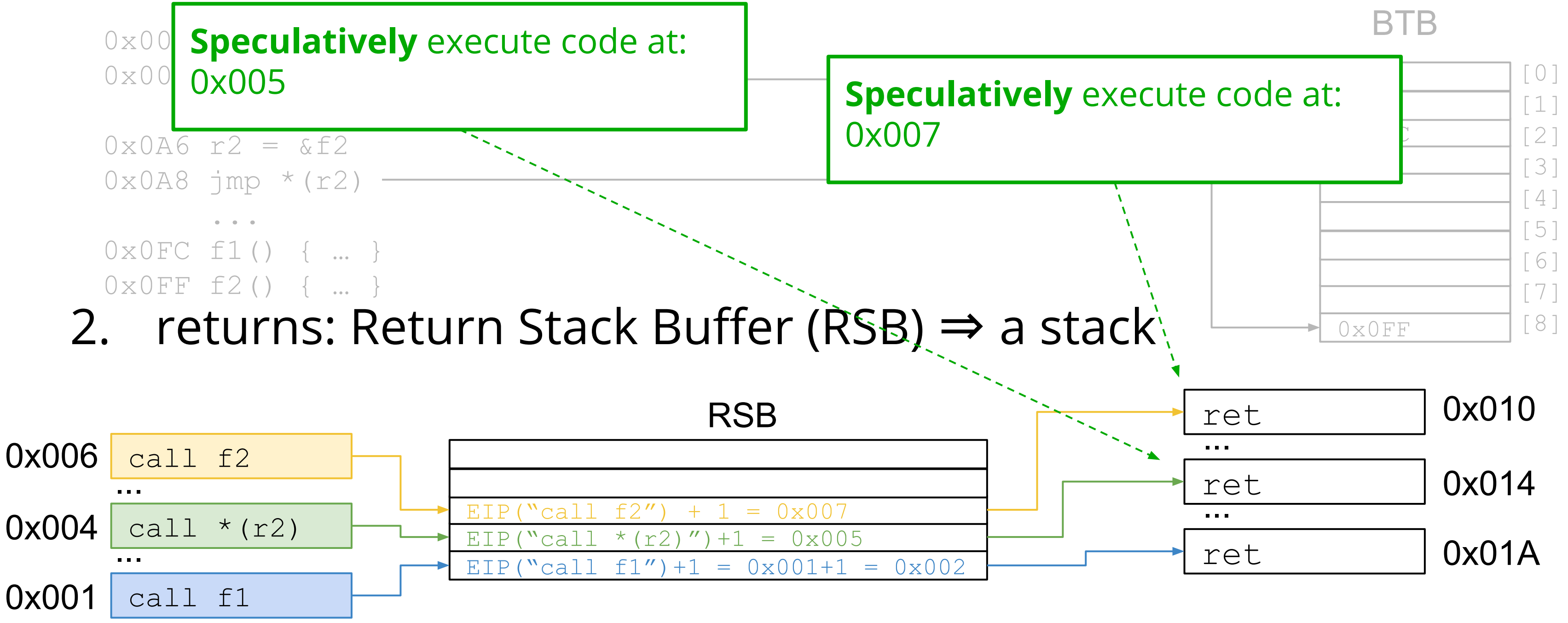
EIP("call f2") + 1 = 0x007
EIP("call *(r2)")+1 = 0x005
EIP("call f1")+1 = 0x001+1 = 0x002

ret 0x010
…
ret 0x014
…
ret 0x01A

Branch Prediction

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a branch cache
2. returns: Return Stack Buffer (RSB) ⇒ a stack



```
0x00
0x00

0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
0x0FC f1() { … }
0x0FF f2() { … }
```

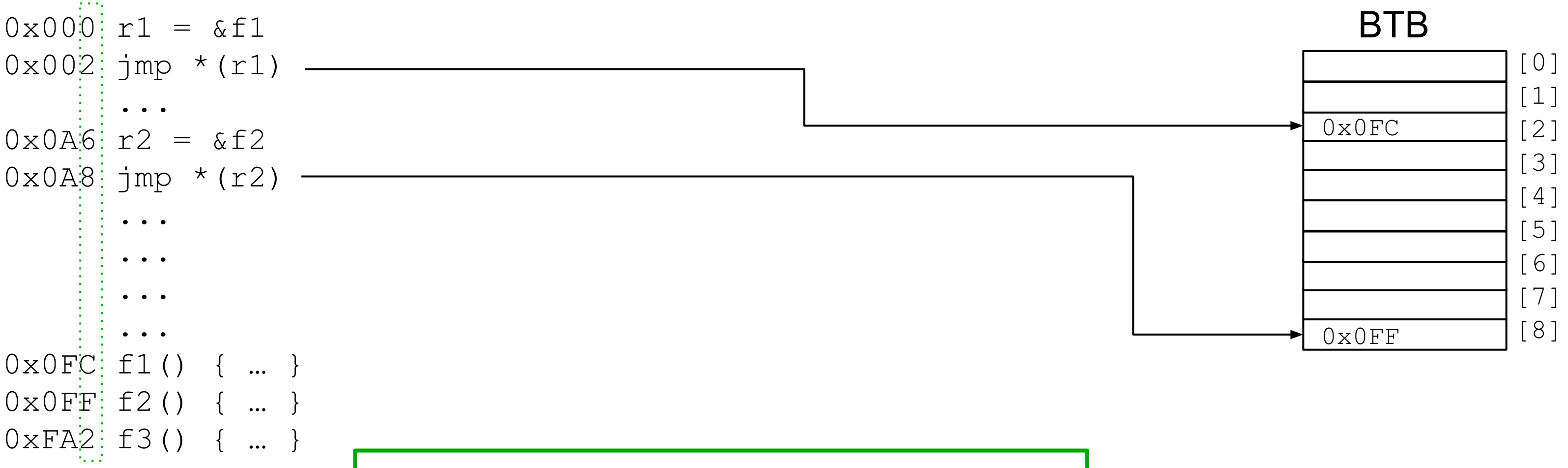
Speculatively execute code at:
0x005

Speculatively execute code at:
0x007

Branch Prediction: Aliasing in the BTB

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a cache/TLB



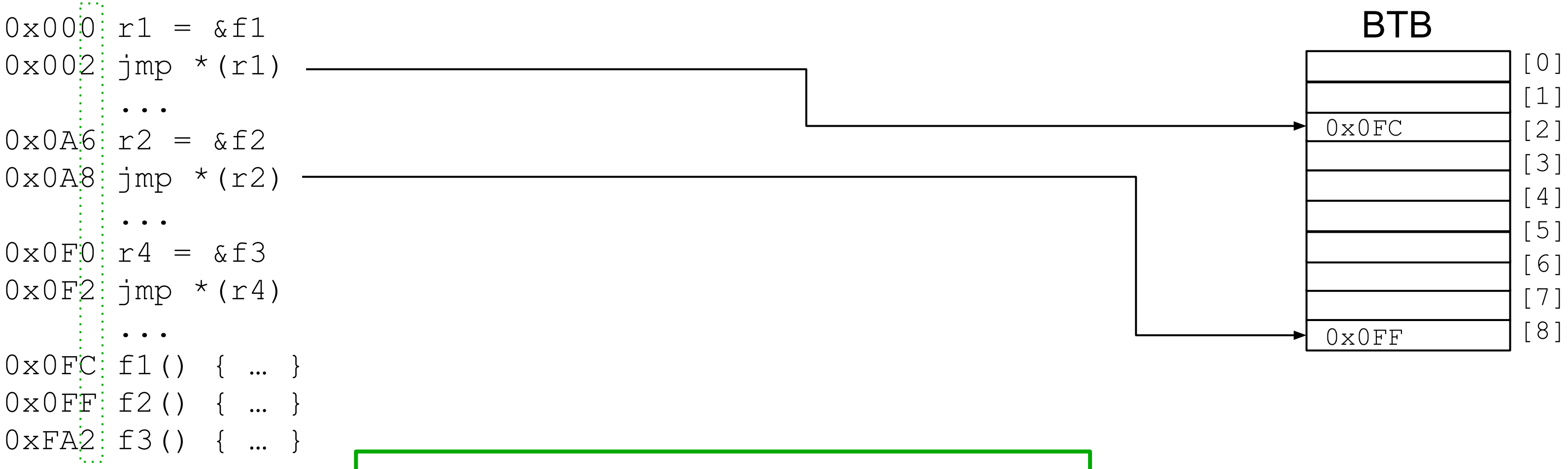
```
0x000 r1 = &f1
0x002 jmp *(r1)
      ...
0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
      ...
      ...
      ...
0x0FC f1() { … }
0x0FF f2() { … }
0xFA2 f3() { … }
```

BTB == an array
Indexing == a hash of jmp instr. address
(in this case, index = last hex digit)

Branch Prediction: Aliasing in the BTB

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a cache/TLB



```
0x000 r1 = &f1
0x002 jmp *(r1)
      ...
0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
0x0F0 r4 = &f3
0x0F2 jmp *(r4)
      ...
0x0FC f1() { … }
0x0FF f2() { … }
0xFA2 f3() { … }
```

BTB

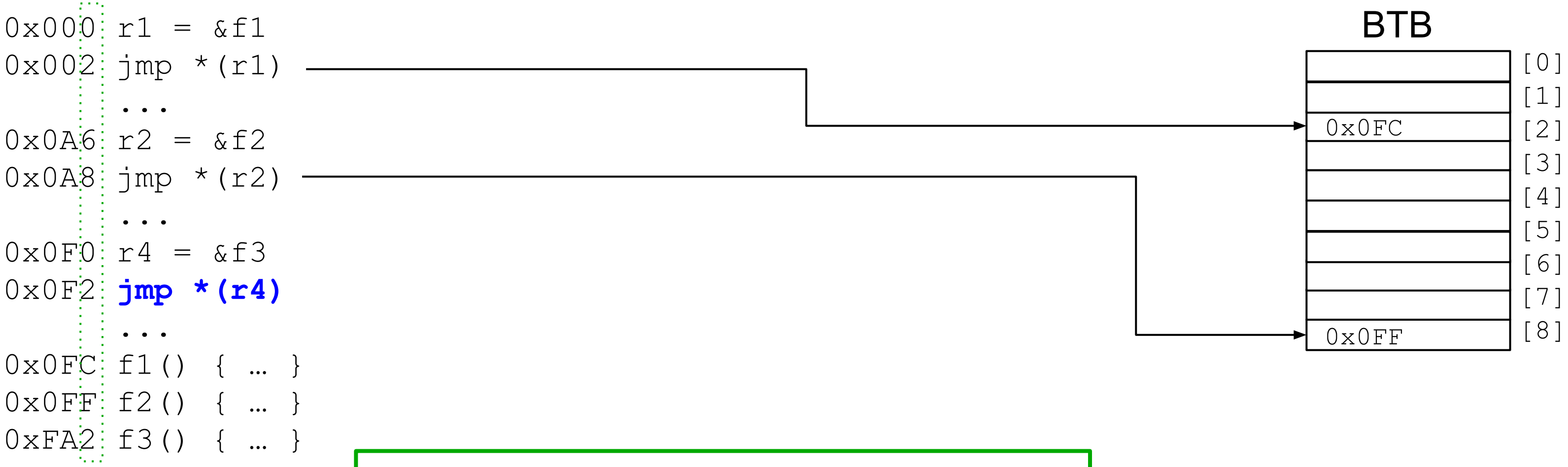
	[0]
	[1]
0x0FC	[2]
	[3]
	[4]
	[5]
	[6]
	[7]
0x0FF	[8]

BTB == an array
Indexing == a hash of jmp instr. address
(in this case, index = last hex digit)

Branch Prediction: Aliasing in the BTB

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a cache/TLB



```
0x000 r1 = &f1
0x002 jmp *(r1)
      ...
0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
0x0F0 r4 = &f3
0x0F2 jmp *(r4)
      ...
0x0FC f1() { … }
0x0FF f2() { … }
0xFA2 f3() { … }
```

BTB

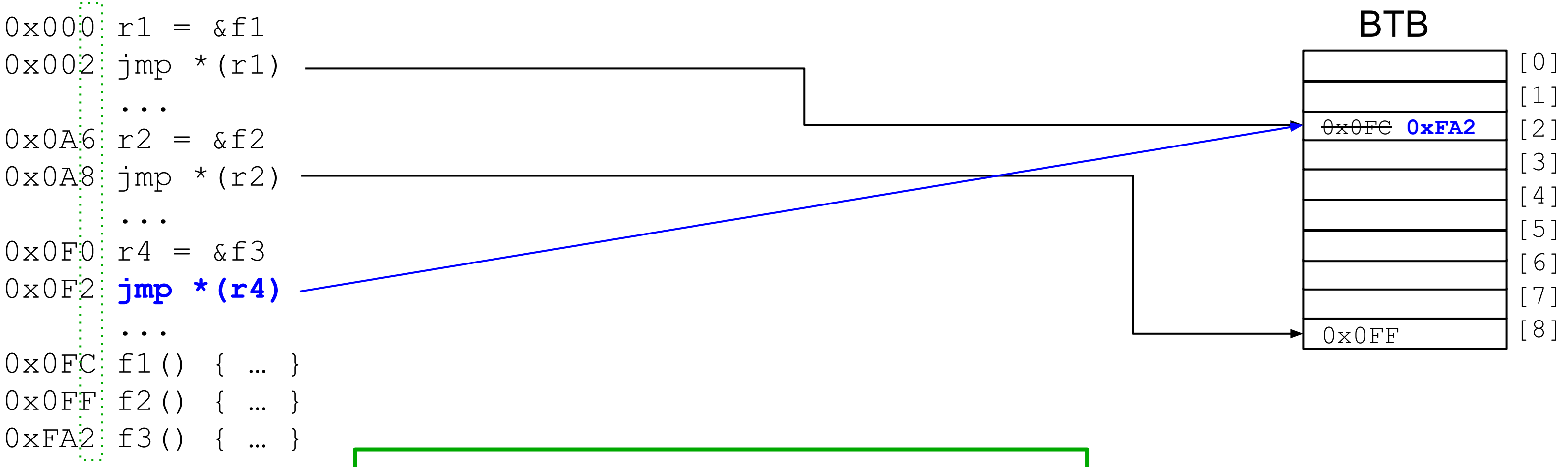
	[0]
	[1]
0x0FC	[2]
	[3]
	[4]
	[5]
	[6]
	[7]
0x0FF	[8]

BTB == an array
Indexing == a hash of jmp instr. address
(in this case, index = last hex digit)

Branch Prediction: Aliasing in the BTB

How do we guess **indirect branches** (func. pointers/returns)

1. direct / indirect calls: Branch Target Buffer (BTB) ⇒ a cache/TLB



```
0x000 r1 = &f1
0x002 jmp *(r1)
      ...
0x0A6 r2 = &f2
0x0A8 jmp *(r2)
      ...
0x0F0 r4 = &f3
0x0F2 jmp *(r4)
      ...
0x0FC f1() { … }
0x0FF f2() { … }
0xFA2 f3() { … }
```

BTB

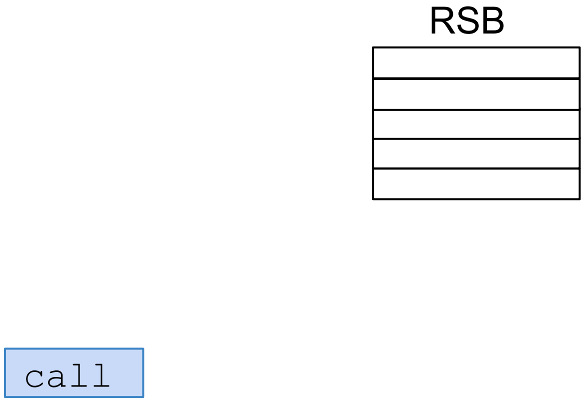
BTB	
	[0]
	[1]
~~0x0FC~~ **0xFA2**	[2]
	[3]
	[4]
	[5]
	[6]
	[7]
0x0FF	[8]

BTB == an array
Indexing == a hash of jmp instr. address
(in this case, index = last hex digit)

Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

2. returns: Return Stack Buffer (RSB) ⇒ a stack



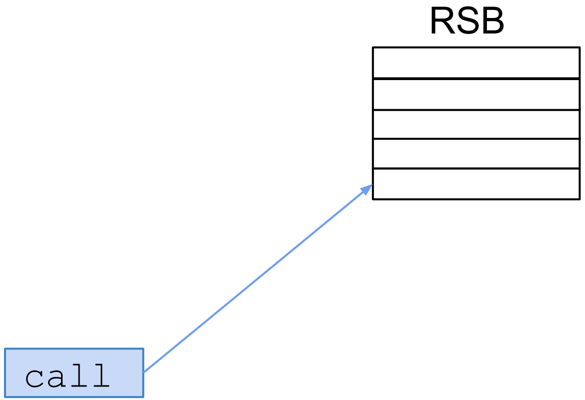
RSB

`call`

Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

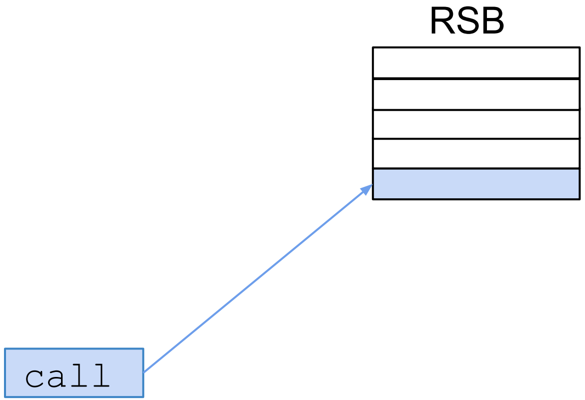
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

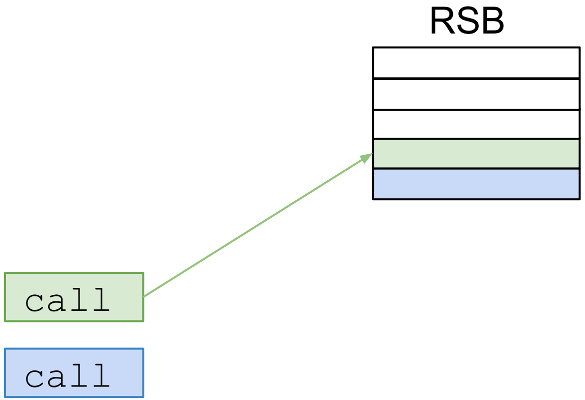
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

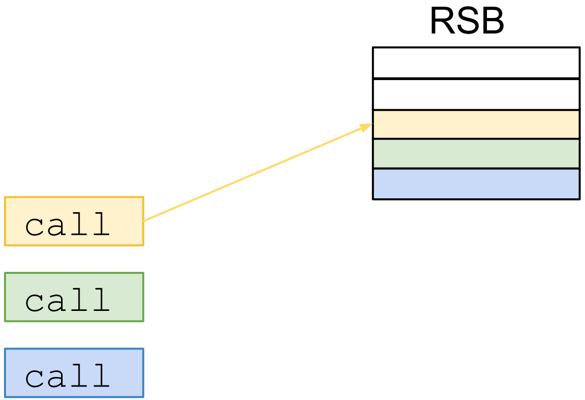
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

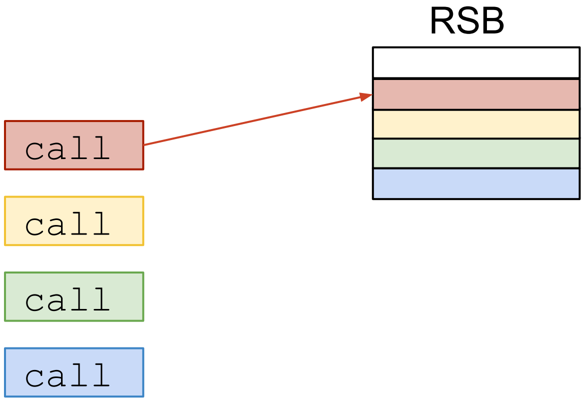
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

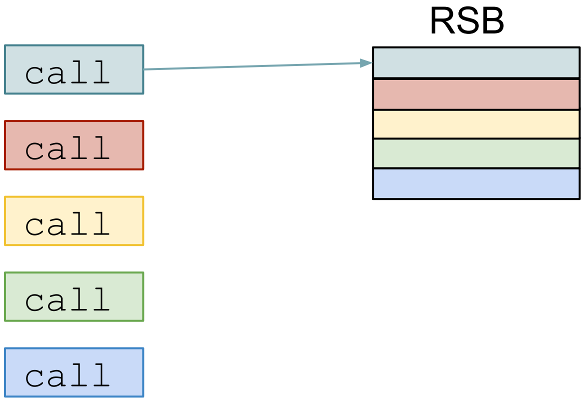
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

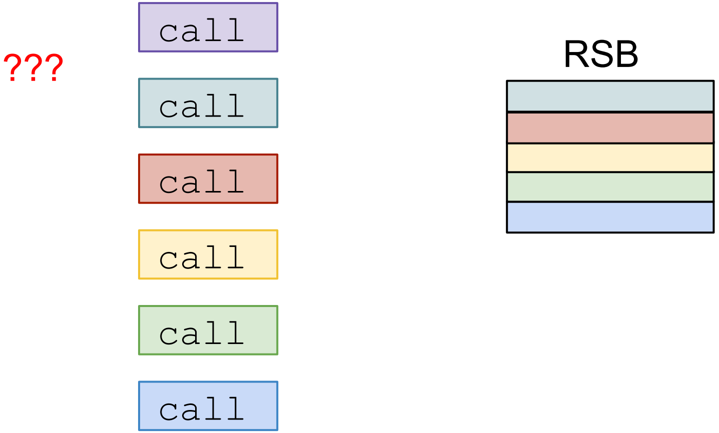
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

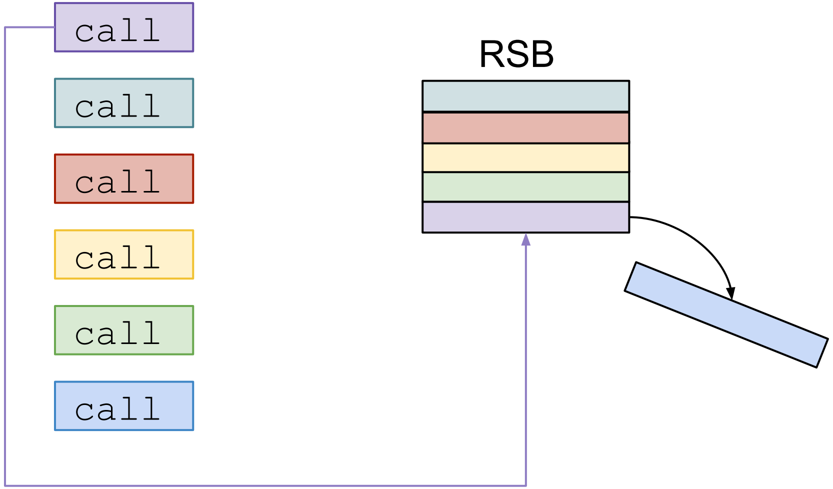
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

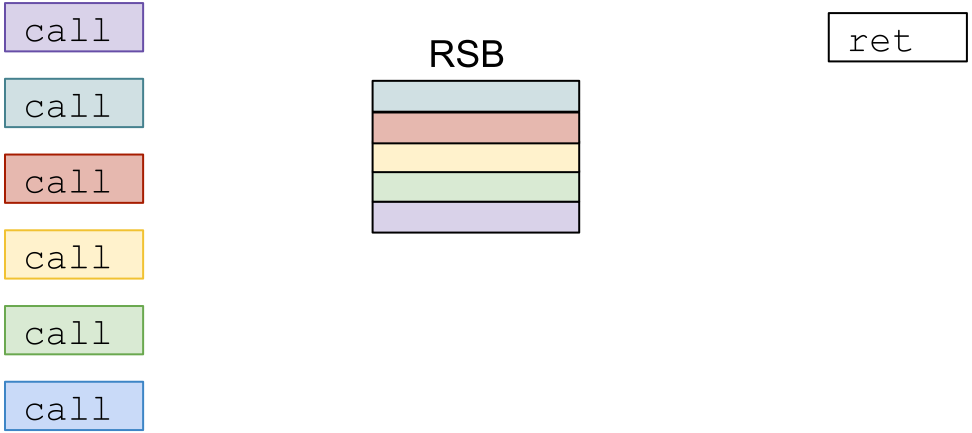
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

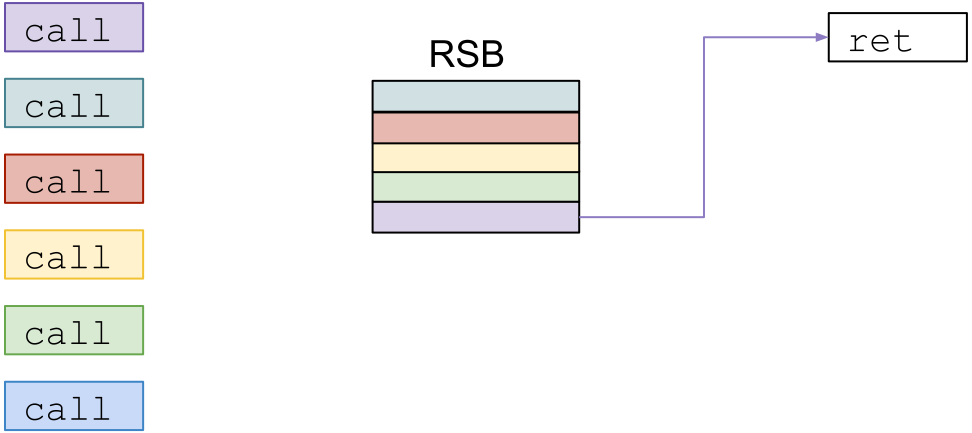
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

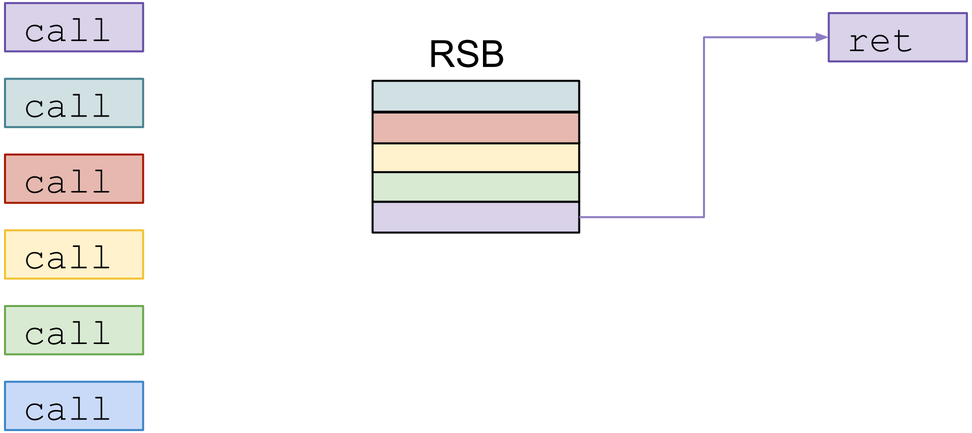
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

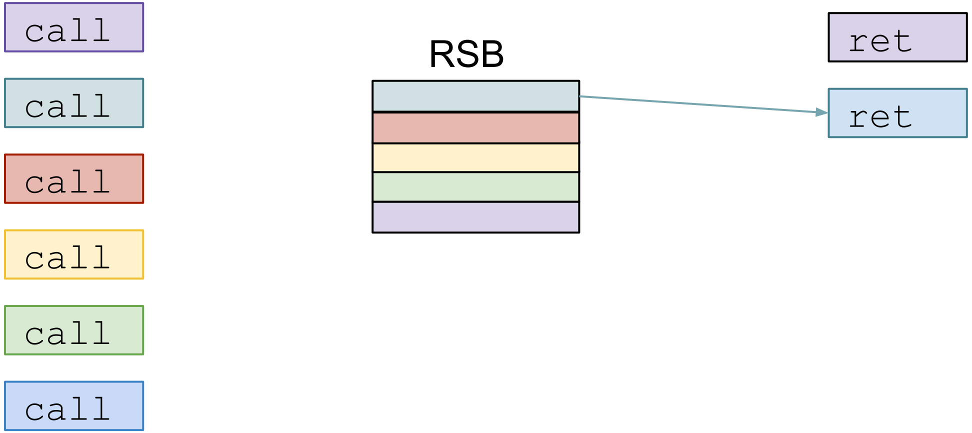
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

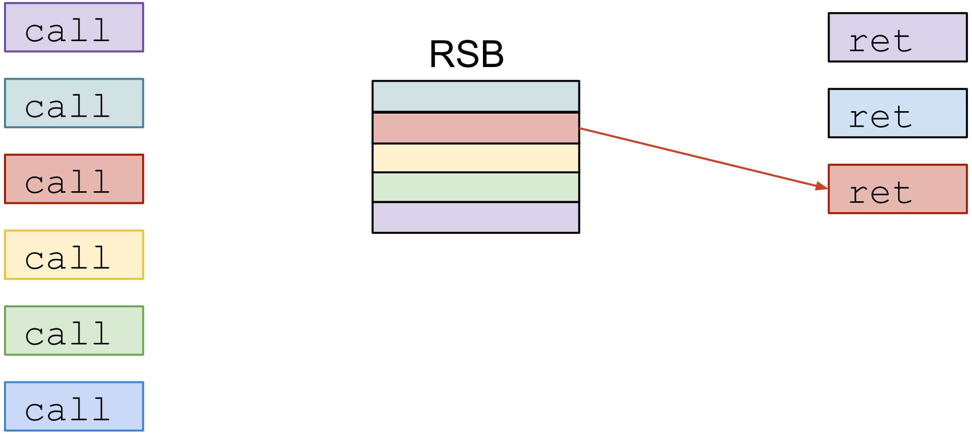
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

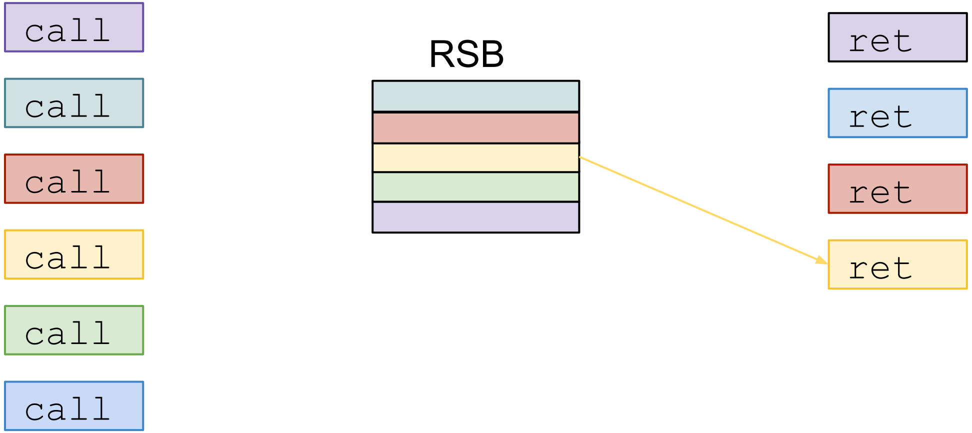
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

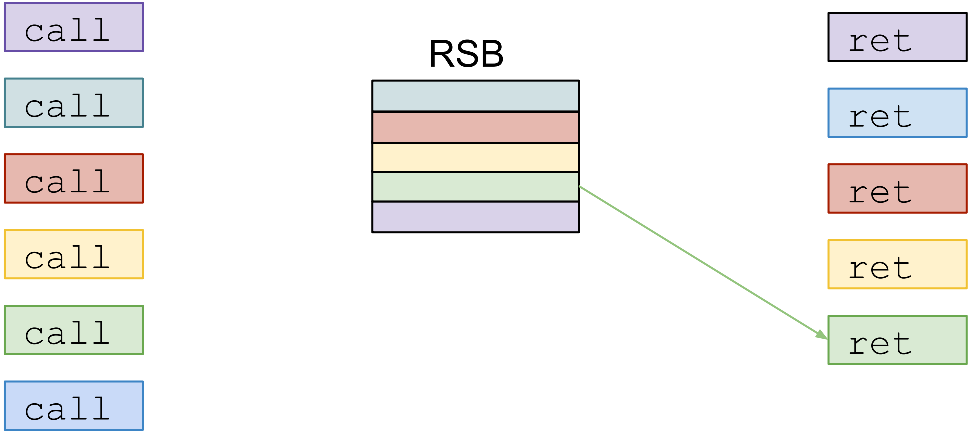
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

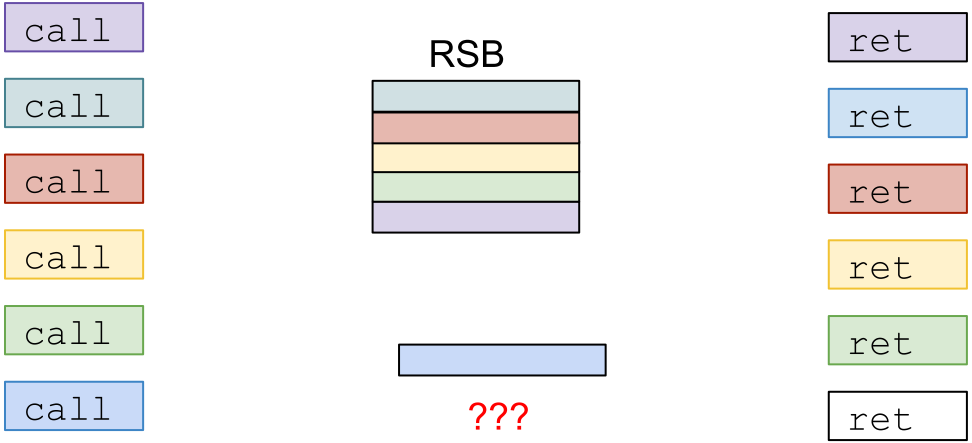
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

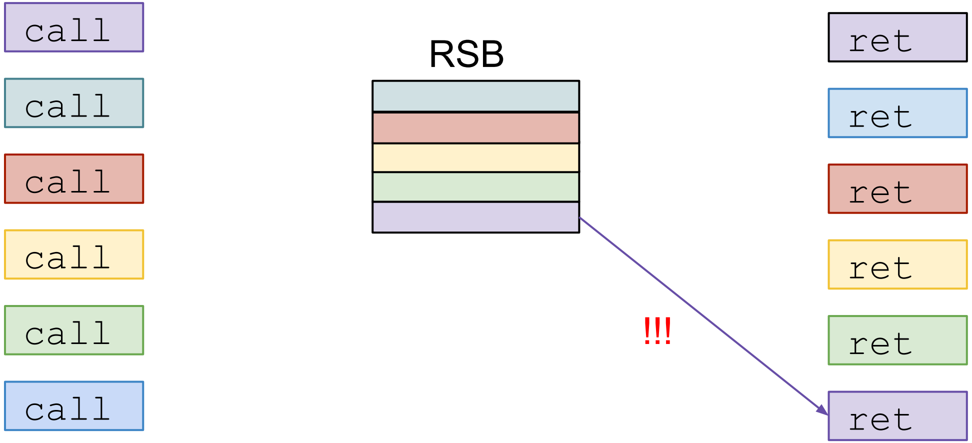
2. returns: Return Stack Buffer (RSB) ⇒ a stack



Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

2. returns: Return Stack Buffer (RSB) ⇒ a stack

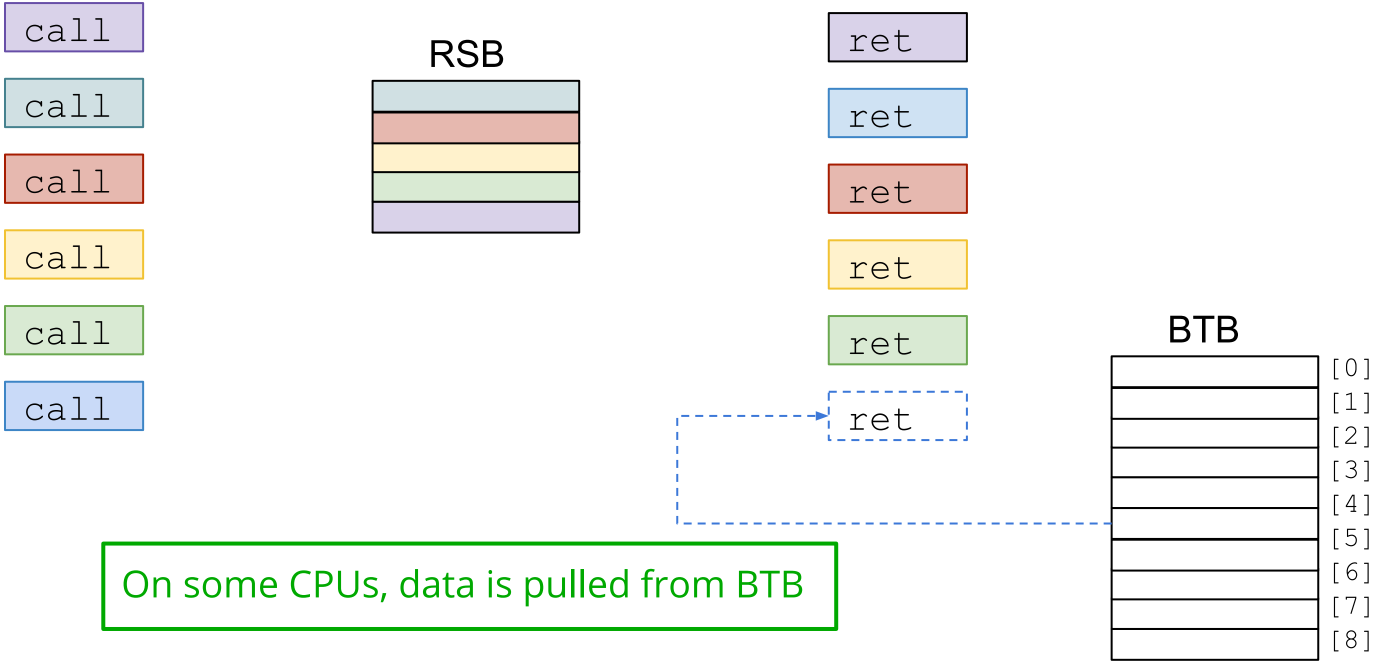


On some CPUs, we just use what we find there...

Branch Prediction: RSB Underflow

How do we guess **indirect branches** (func. pointers/returns)

2. returns: Return Stack Buffer (RSB) ⇒ a stack



On some CPUs, data is pulled from BTB

Alternative Universes

Speculative execution

- **speculate** = to guess, **execution** = to do something
  speculative execution = do something based on a guess

- IRL:
  - You to a friend: <<hey, do you want a cup coffee?>>
  - While talking/waiting for answer: turn on machine, prep. cups, ...

- In CPUs:
  - Memory is slow. While waiting for data, do something
  - instruction reordering, superscalar pipelines, branch prediction, ...

```
if <A> is true          ⇒      do <x> | check
  do <x>                       <A>
```

  - Modern CPUs speculate **a lot!** (~= 200 entries reorder buffers)

Speculative Execution:
`do <x>` , while waiting to be able to `check <A>`

[Kernel Recipes ‘18: Paolo Bonzini - “Meltdown and Spectre: seeing through the magician’s tricks”]

[NYLUG: Andrea Arcangeli, Jon Masters - “Speculation out of control, taming CPU bugs”]

Speculative Execution: Alternate Universes (*)

- I can create an alternate universe
- everything the same, **I have superpowers**:
  - I can do whatever I want, I always succeeds (it's *my* alternate universe! :-D)
- After, say, 30 seconds:
  - alternate universe disappears
  - in the original universe, I remember nothing :-(
  - **good** things I've done ⇒ "copied" back to original universe
  - **bad** things I've done ⇒ *never happened* in original universe

(*) Analogy stolen from George's talk

Speculative Execution: Alternate Universes (*)

What if, alteration of the **heat** of objects, happening in the alternate universe, **leaks** to original universe?

- I can c
- everyt
  - I c
    (it
- After, say, 30 seconds:
  - alternate universe disappears
  - in the original universe, I remember nothing
  - **good** things I've done ⇒ "copied" back to original universe
  - **bad** things I've done ⇒ *never happened* in original universe

(*) Analogy stolen from George's talk

Speculative Execution: Alternate Universes (*)

- I can create an alternate universe
- everything the same, **I have superpowers**:
  - I ca
    (it's
- After, s
  - alte
  - in t
  - **goo** niverse
  - **bac** niverse

(*) Analogy stolen fro

Speculative Execution: Alternate Universes (*)

- I can create an alternate universe
- everything the same, **I have superpowers:**
  - I ca
    (it's
- After, s
  - alte
  -

niverse

niverse

Stop looking at Facebook, BTW!

(*) Analogy stolen fro

Side Channels

Side Channels (Covert Channels)

Gaining information on a system by **observing** its behavior

- ~~Read otherwise unaccessible memory via a buffer overflow~~
- Measuring microarchitectural properties
- ⇒ not interact with nor influence execution of a program
- ⇒ not let one modify/delete/... any data

Caches as side channels:

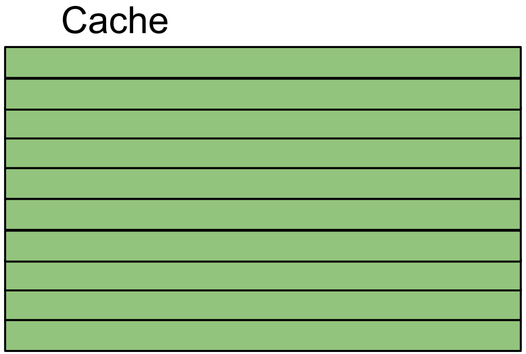
- Accessing memory is fast, if data is in cache
- Accessing memory is slow, if data is in cache
- ⇒ measuring data access time == *cache side-channel*

Cache as a Side Channel

Execution time of **instruction**: depending on **data** being in caches

Example:

- I fill the cache (big array)
- Call `target_func(int idx)`
  - I control value of `idx`
- `target_func()` bring its data in cache



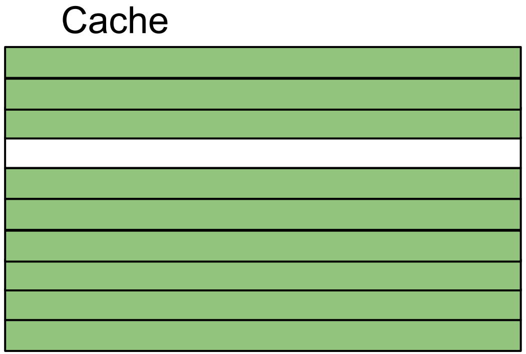
Prime and Probe

Cache as a Side Channel

Execution time of **instruction**: depending on **data** being in caches

Example:

- I fill the cache (big array)
- Call `target_func(int idx)`
  - I control value of `idx`
- `target_func()` bring its data in cache
- I measure access time to all array elements
- The slowest one tells me something about what `target_func()` has done
  - (remember, I control, idx)



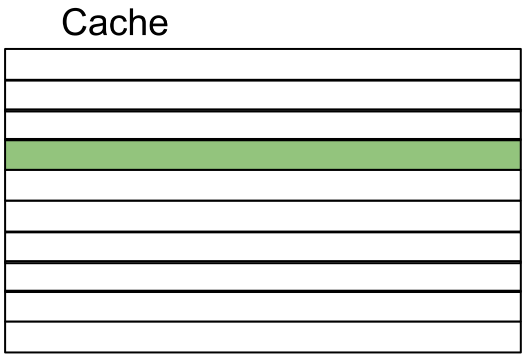
Prime and Probe

Cache as a Side Channel (The Other Way Round)

Execution time of **instruction**: depending on **data** being in caches

Example:

- I ~~fill~~ empty the cache (big array)
- Call `target_func(int idx)`
  - I control value of `idx`
- `target_func()` bring its data in cache
- I measure access time to all array elements
- The ~~slowest~~ fastest one tells me something About what `target_func()` has done
  - (remember, I control, idx)



Flush and Reload

Attacking Speculative Execution

Speculative Execution Attack

```
result_bit = 0;           //goal: read the 5th bit of what's at an address
bit = 4;                  //that I normally wouldn't be able to read!
flush_cacheline(L);
if ( fork_alt_univ() ) {  //returns 1 in alternate, 0 in original universe :-)
  if ( *target_address & (1 << bit) )
    //in the alternate universe now
    load_cacheline(L);
}
if ( is_cacheline_loaded(L) )
  //”Back” in in original universe
  result_bit = 1;
```

Remember alternate universes…

do it in a loop, use a bitmask and shift (<<)

Speculative Execution Attack

This is how we “trick” the CPU to execute code “in speculation” (e.g., “poison” branch prediction)

The CPU is executing this “in speculation” ==> **no fault!**

```
result_bit = 0;             //goal: read the 5th bit of what's at an address
bit = 4;                    //that I normally wouldn't be able to read!
flush_cacheline(L);
if ( fork_alt_univ() ) {   //returns 1 in alternate, 0 in original universe
:-)
  if ( *target_address & (1 << bit) )
    //in the alternate universe now
    load_cacheline(L);
}
if ( is_cacheline_loaded(L) )
  //"Back" in in original universe
  result_bit = 1;
```

Cache used as a **side-channel:**

Extract information from behavior

E.g., our looking-at-Facebook “heated” spoon, a stethoscope for hearing locks’ clicks, …

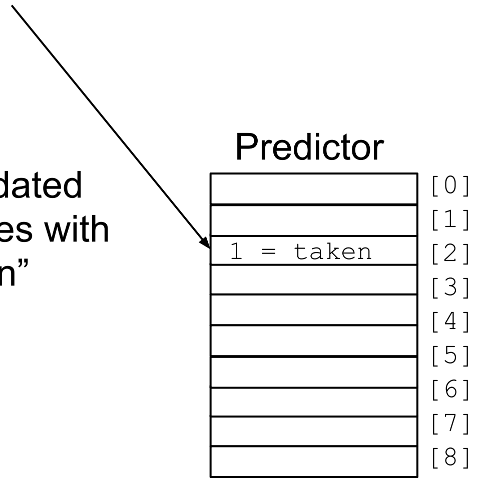
do it in a loop, use a bitmask and shift (<<)

BTB Poisoning Attack

Conditional branch predictor:

Attacker:

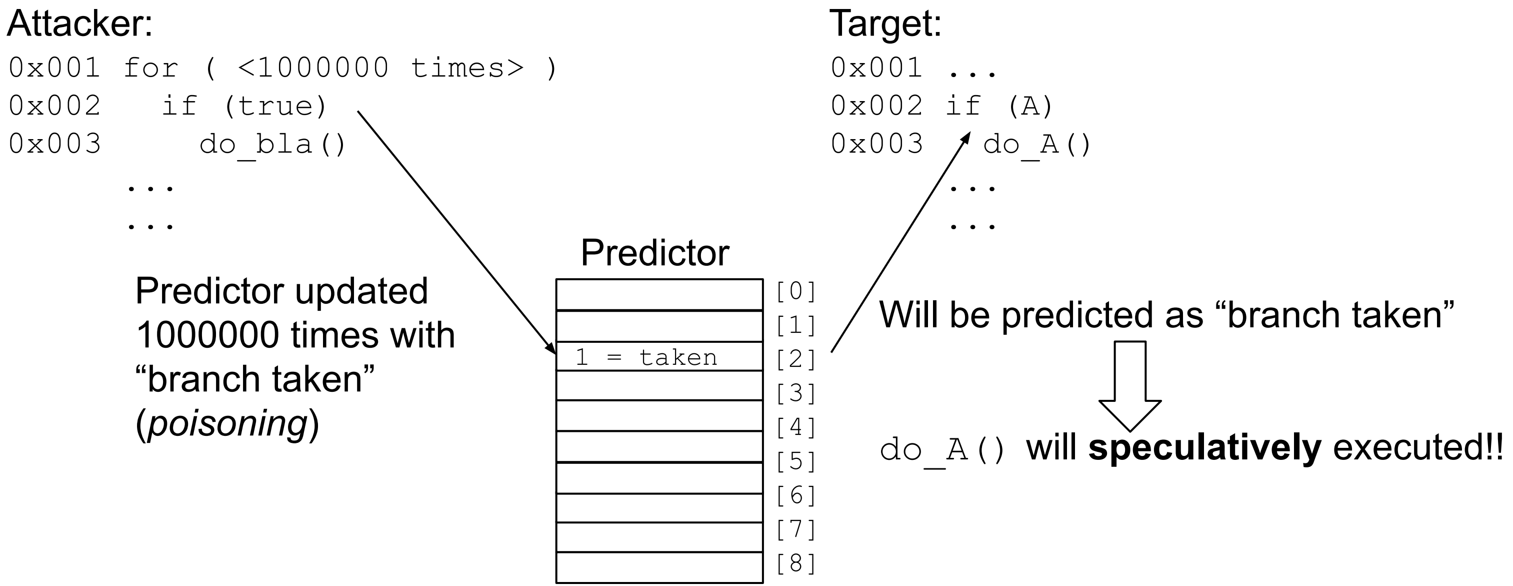
```
0x001 for ( <1000000 times> )
0x002   if (true)
0x003     do_bla()
      ...
      ...
```



Predictor updated 1000000 times with "branch taken"

BTB Poisoning Attack

Conditional branch predictor:



Attacker:

```
0x001 for ( <1000000 times> )
0x002   if (true)
0x003     do_bla()
      ...
      ...
```

Predictor updated 1000000 times with "branch taken" (*poisoning*)

Predictor

Predictor	
	[0]
	[1]
1 = taken	[2]
	[3]
	[4]
	[5]
	[6]
	[7]
	[8]

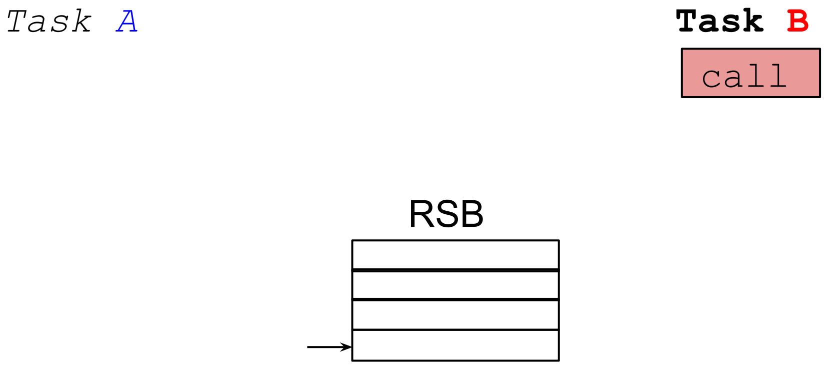
Target:

```
0x001 ...
0x002 if (A)
0x003   do_A()
        ...
        ...
```

Will be predicted as "branch taken"

`do_A()` will **speculatively** executed!!

RSB Underflow “Attack”



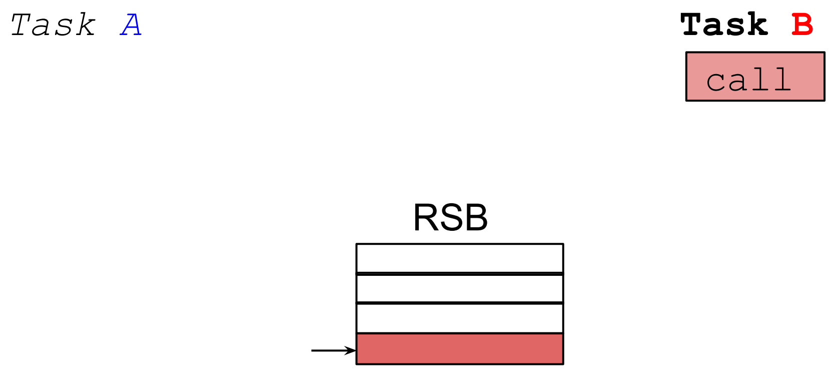
Task A

Task B

call

RSB

RSB Underflow “Attack”



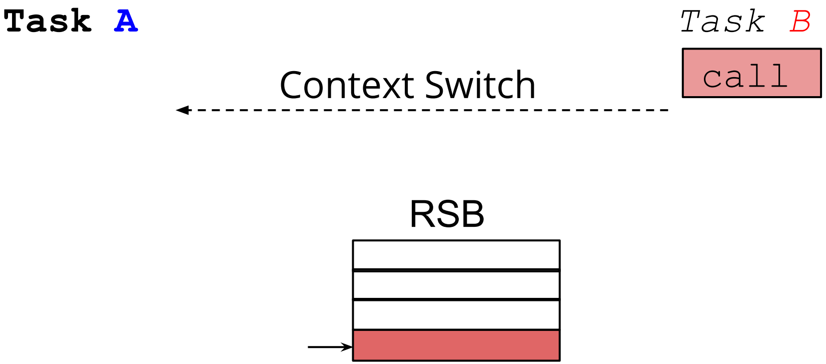
Task A

Task B

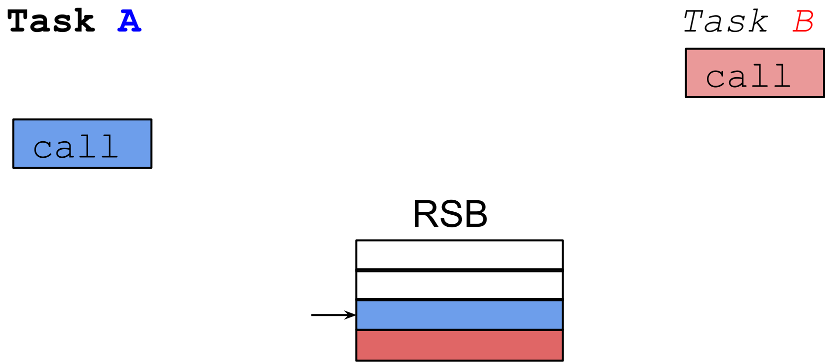
call

RSB

RSB Underflow “Attack”



RSB “Overflow Attack”



Task **A**

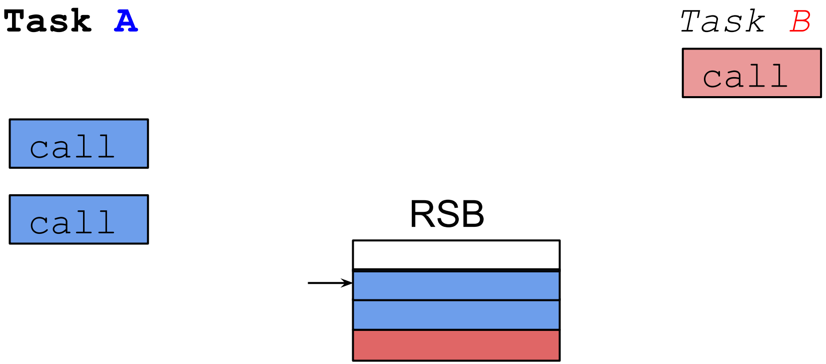
Task B

call

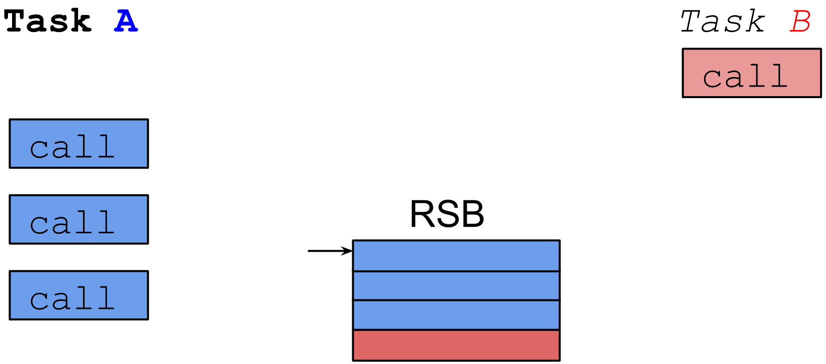
call

RSB

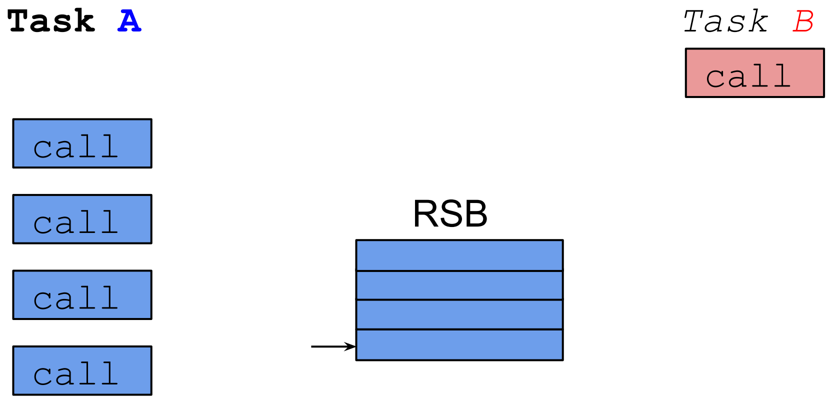
RSB “Overflow Attack”



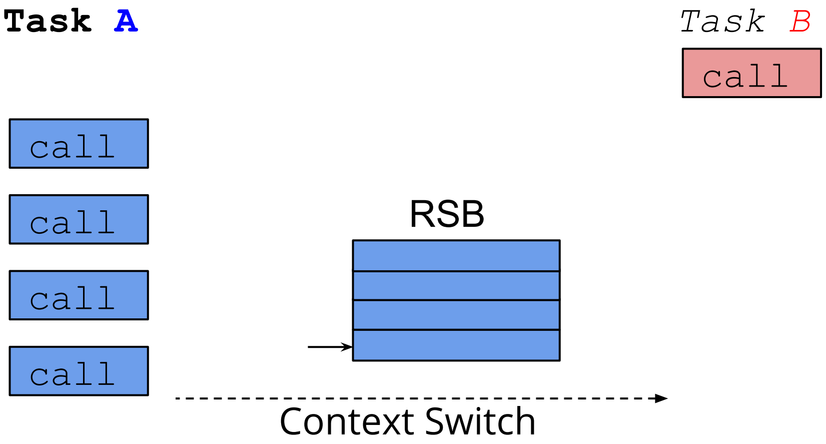
RSB “Overflow Attack”



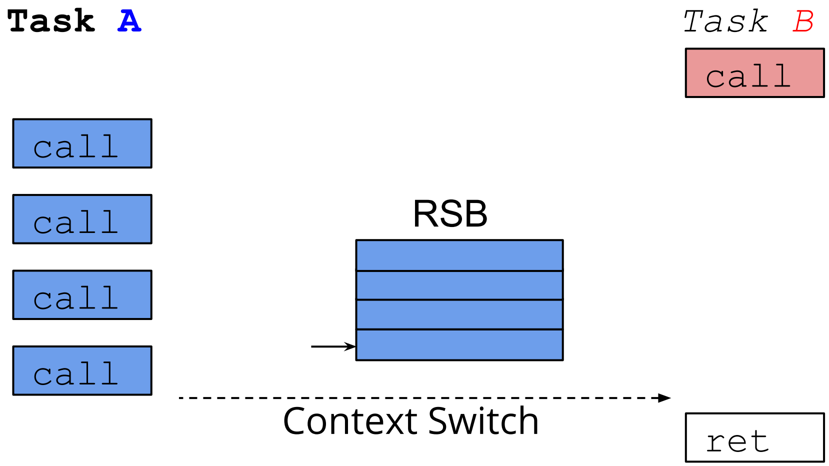
RSB “Overflow Attack”



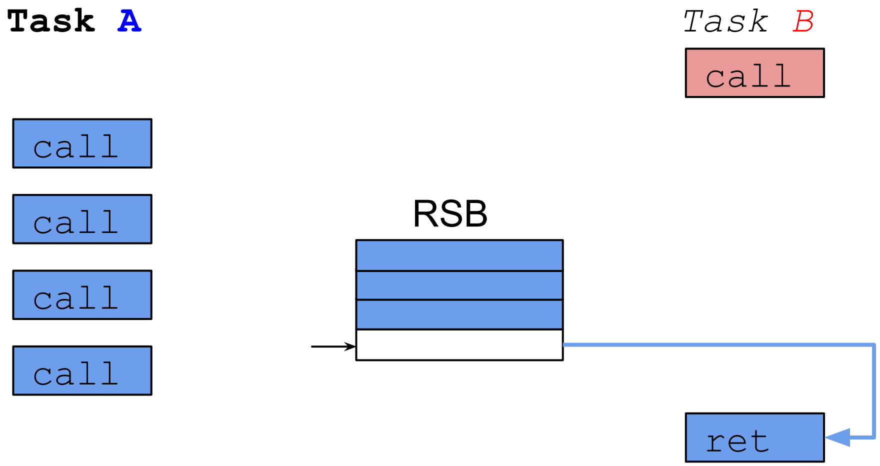
RSB “Overflow Attack”



RSB “Overflow Attack”

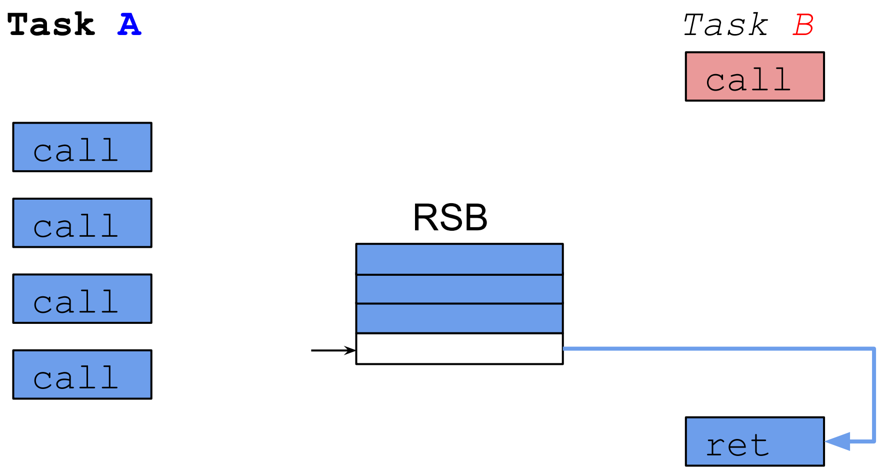


RSB “Overflow Attack”



It’s blue... Shouldn’t it be red?!? :-O

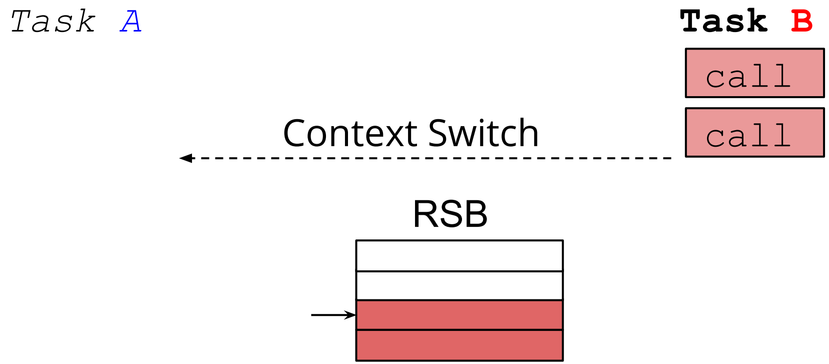
RSB "Overflow Attack"



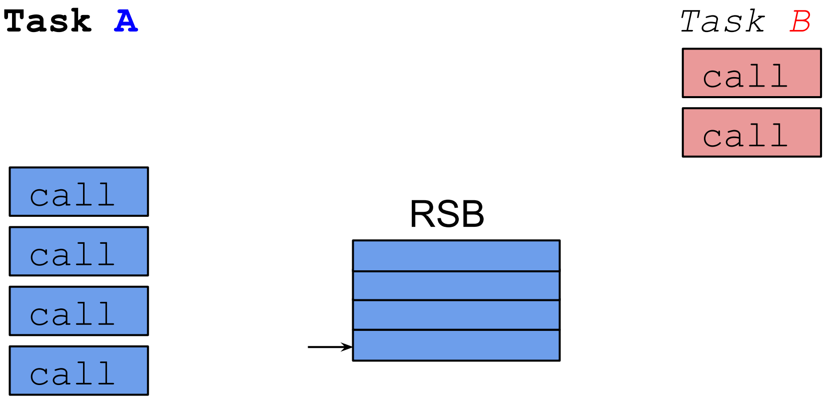
It's blue... Shouldn't it be red?!? :-O

While **B**'s `ret` is being done, CPU speculatively executes **A**'s code (or, potentially, **A**'s controlled code)!

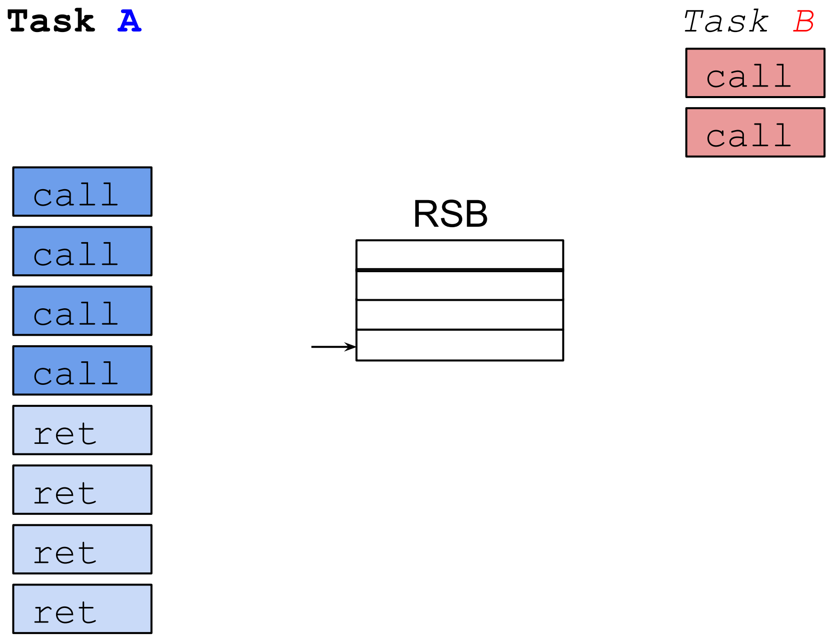
RSB "Underflow Attack"



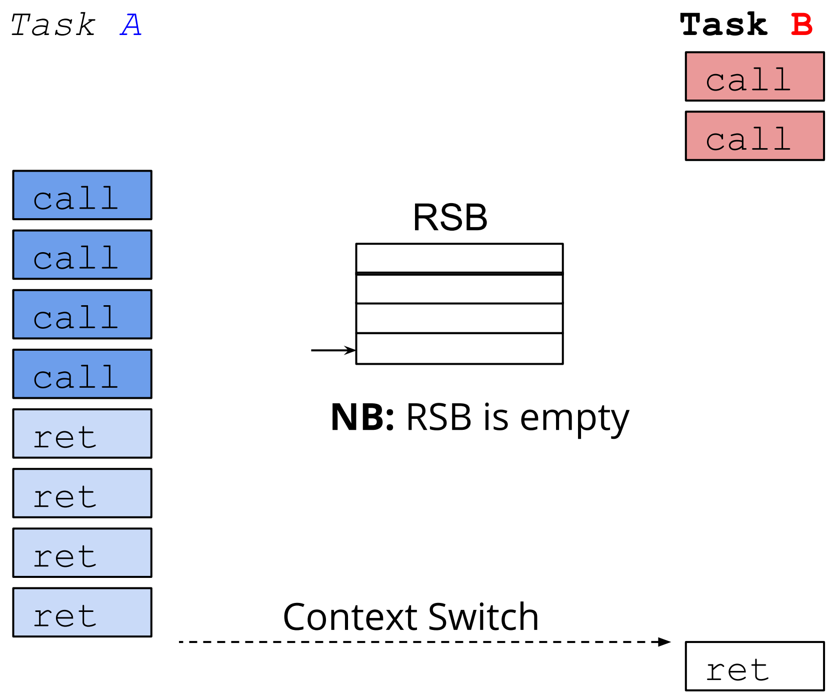
RSB “Underflow Attack”



RSB “Underflow Attack”

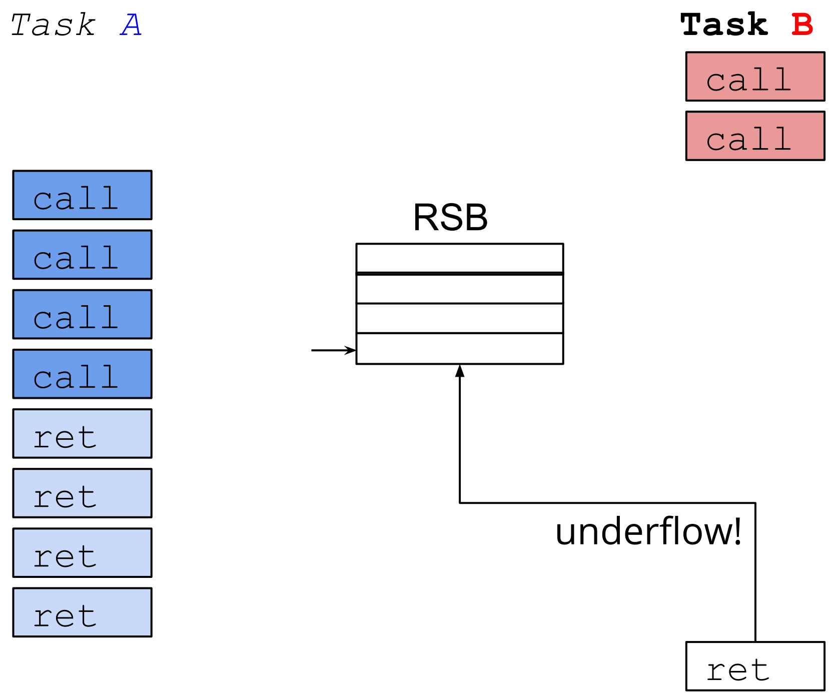


RSB “Underflow Attack”

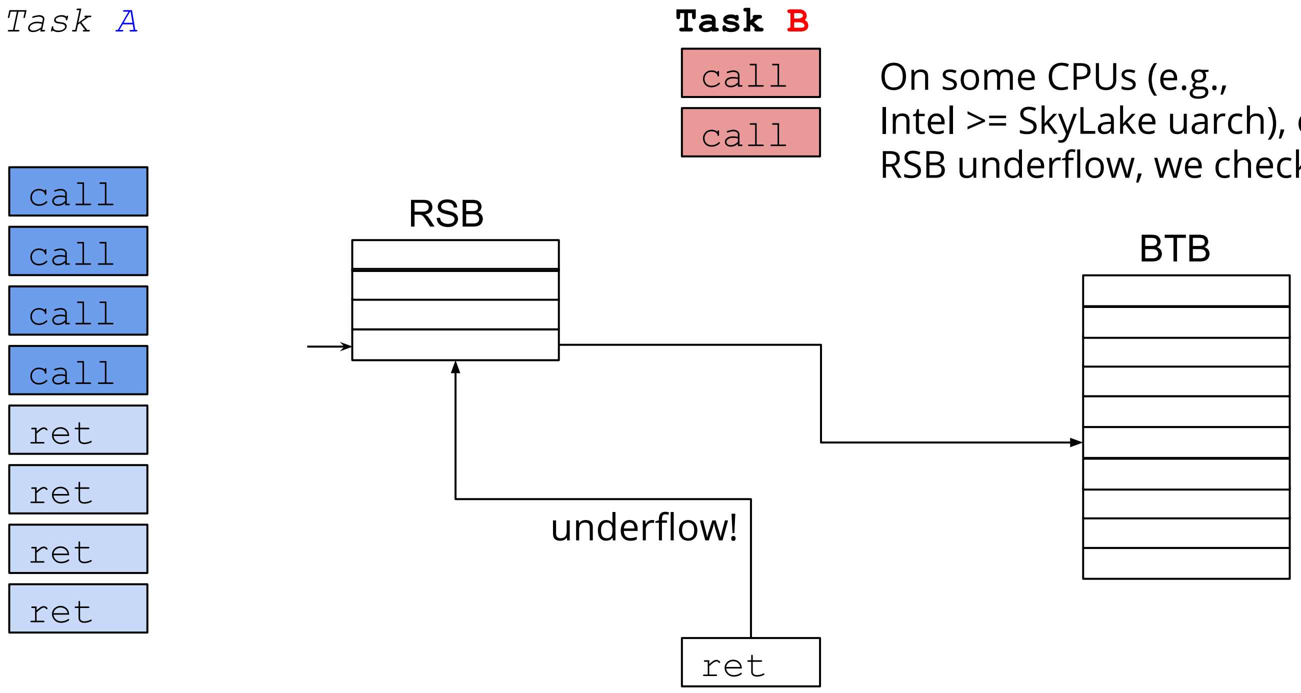


NB: RSB is empty

RSB “Underflow Attack”

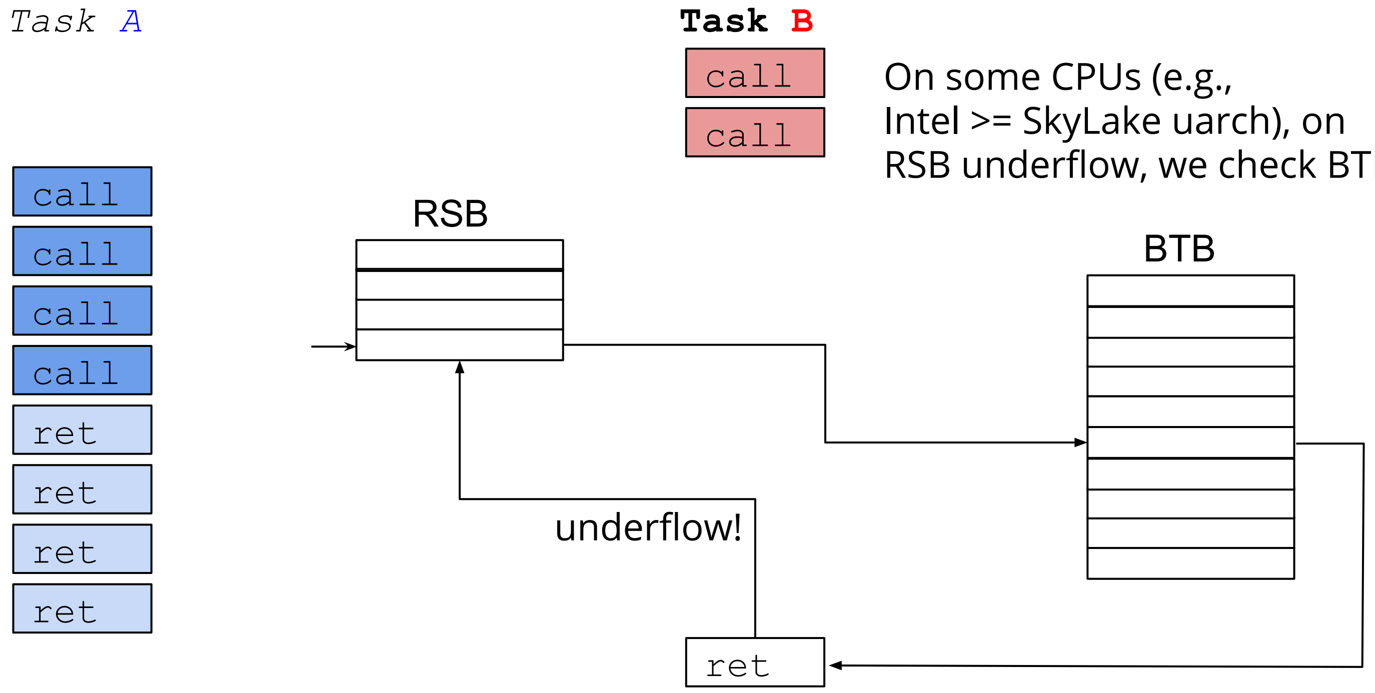


RSB “Underflow Attack”



On some CPUs (e.g., Intel >= SkyLake uarch), on RSB underflow, we check BTB

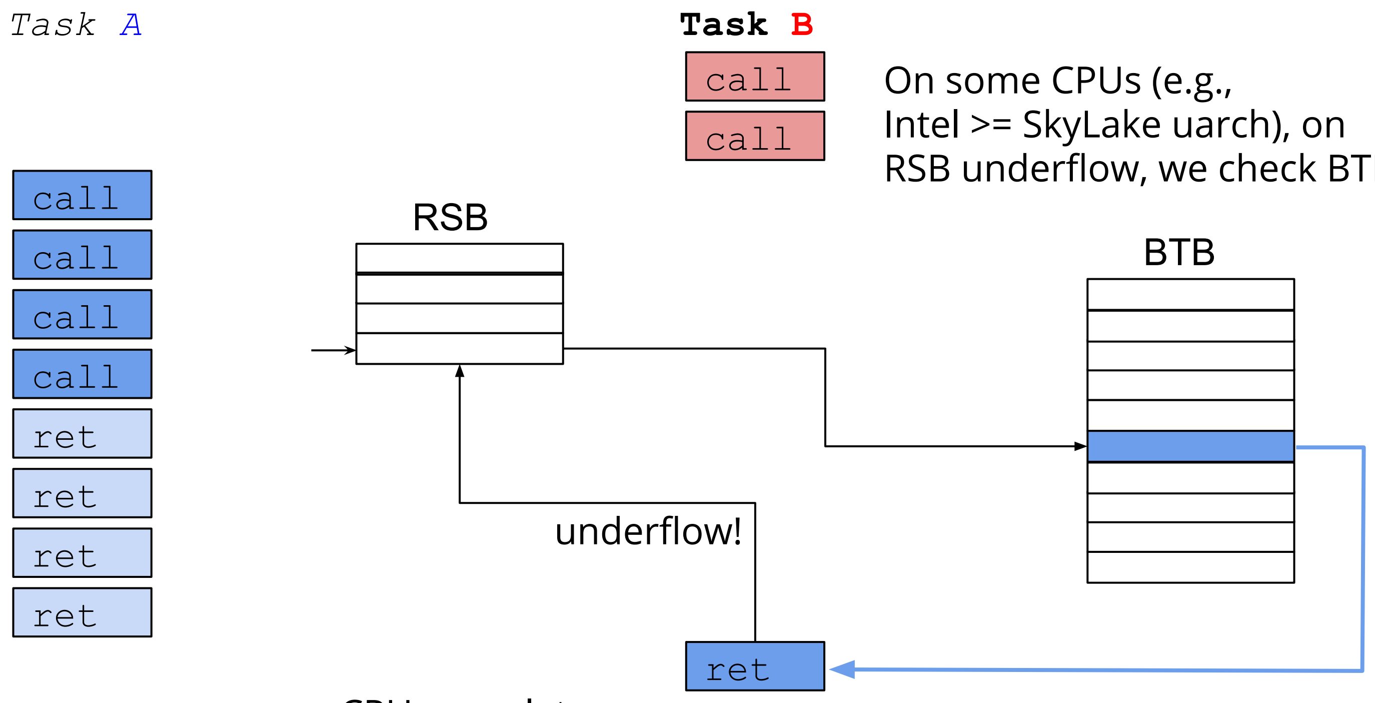
RSB “Underflow Attack”



On some CPUs (e.g., Intel >= SkyLake uarch), on RSB underflow, we check BTB

CPU speculates on What’s in the BTB

RSB “Underflow Attack”



On some CPUs (e.g., Intel >= SkyLake uarch), on RSB underflow, we check BTB

CPU speculates on What’s in the BTB

What if A poisoned the BTB?!?

Speculative Execution: Fundamental Assumptions

- **Spec. Execution** ~= out-of-order execution + branch prediction
- Safe *iff*:
    a. **Rollback works**: not retired (== executed speculatively, but rolled back) instructions have *no side effects* and leave no trace

    b. **No messing with guesses**: it is impossible to *reliably* tell whether or not a particular block of code will be executed speculatively

Speculative Execution: Fundamental Assumptions

- **Spec. Execution** ~= out-of-order execution + branch prediction
- Safe *iff*:
  a. **Rollback works**: ~~not retired (== executed speculatively, but rolled back) instructions have *no side effects* and leave no trace~~
     Architectural registers, flags, ..., ok, no side effects.
     Caches, TLBs, ..., not ok, *side effects*!

  b. **No messing with guesses**: it is impossible to *reliably* tell whether or not a particular block of code will be executed speculatively

Speculative Execution: Fundamental Assumptions

- **Spec. Execution** ~= out-of-order execution + branch prediction
- Safe *iff*:
    a. **Rollback works**: ~~not retired (== executed speculatively, but rolled back) instructions have *no side effects* and leave no trace~~
    Architectural registers, flags, …, ok, no side effects.
    Caches, TLBs, …, not ok, *side effects*!

    b. **No messing with guesses**: ~~it is impossible to *reliably* tell whether or not a particular block of code will be executed speculatively~~
    Predictions based on history (branch having previously been taken/!taken can be "poisoned", and hence *controlled*

Threat Model / Attack Scenarios

Meltdown / Spectre / others: TL;DR

Data Exfiltration “only”:

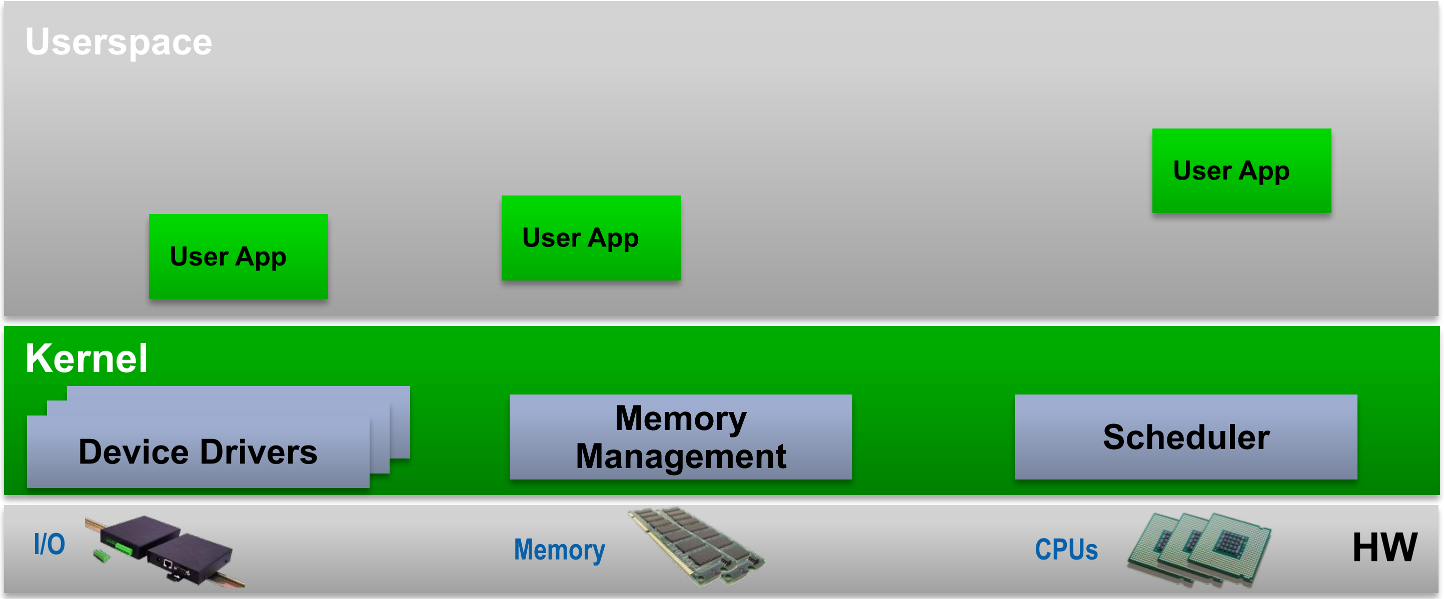
- An unprivileged application can read (but not write) other’s memory, irrelevant of the isolation technique (virtualization, container, namespace…) or the OS (Linux, Windows, MacOS…)

- Does not provide privilege escalation per-se, although it can help

Is my credit card data at risk:

- Don’t know / don’t care
- We’ll talk about technical aspects

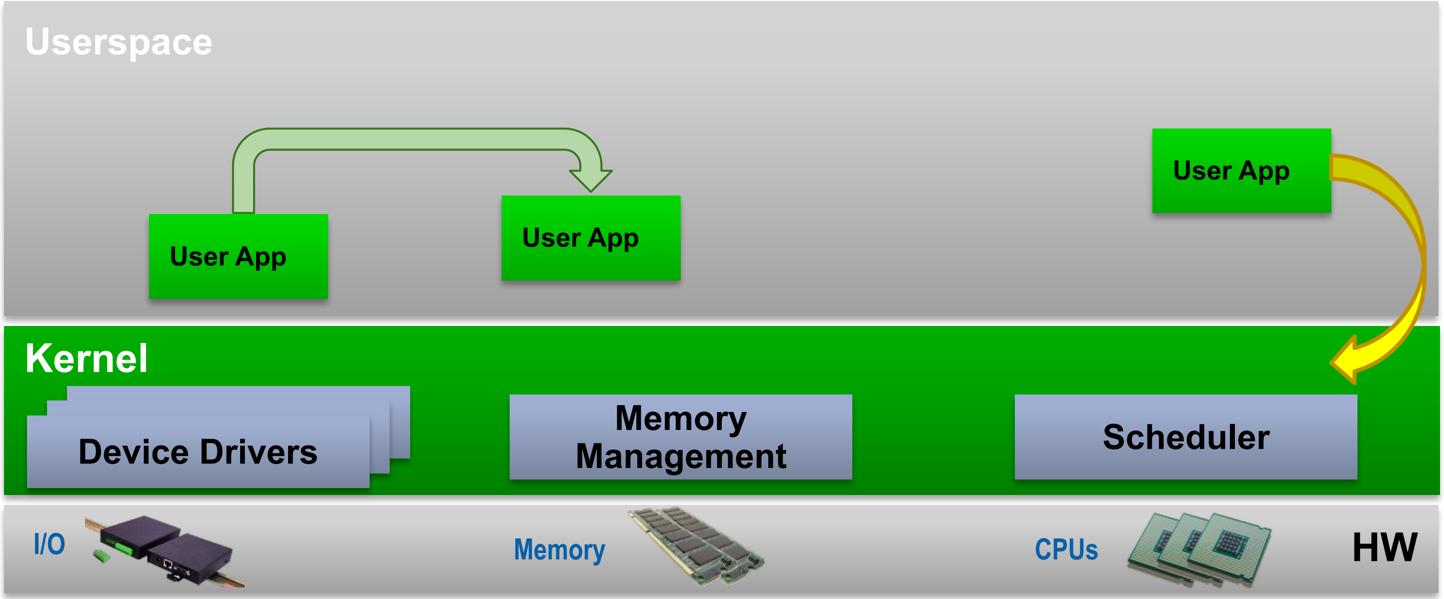
Security, isolation, ...



Security, isolation, ...

Attack Scenarios:

↷ == **successfully** attacked!
(e.g., read data/steal secrets)



- User App to Other User App(s)
- User App to Kernel

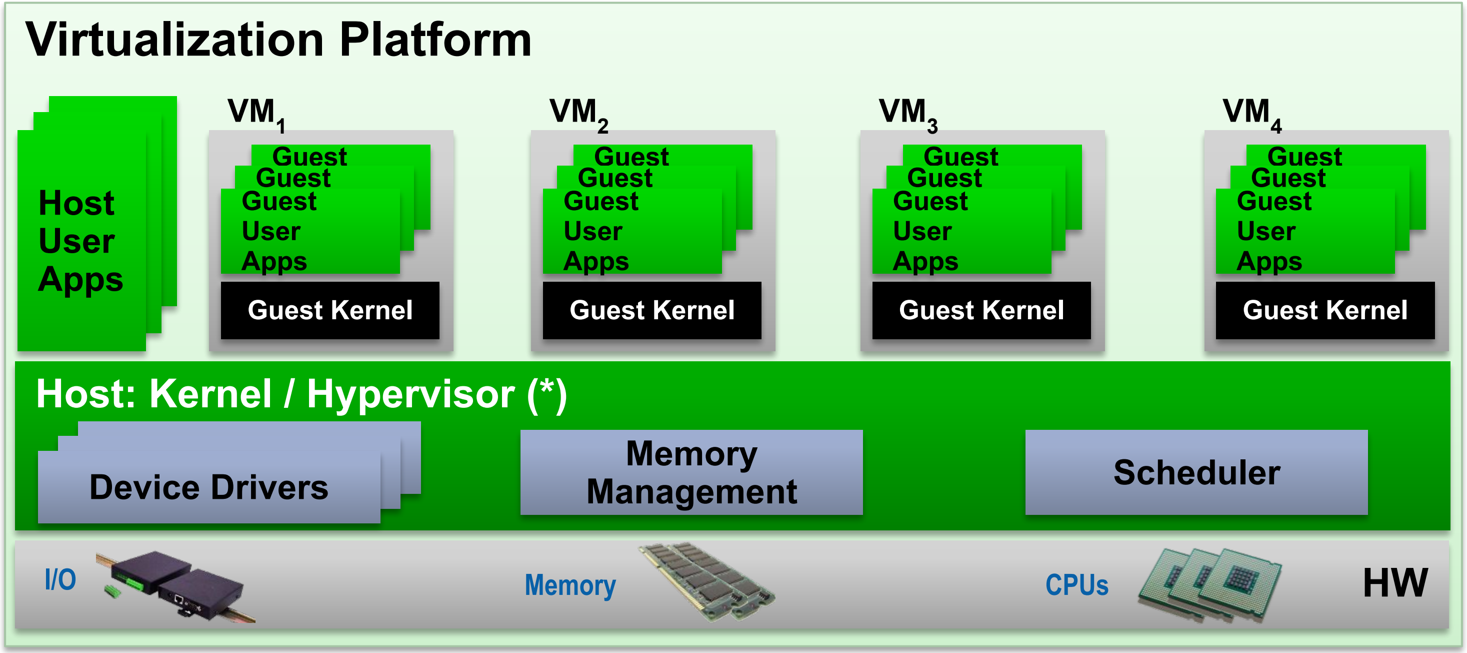
Security, isolation …

Attack Scenarios:

1. **User App to Other User Apps(s)**
   - Damage contained within App(s) data
   - Might be different apps of same user / different apps of different users
   - User Apps must protect themselves

2. **User App to Kernel**
   - Implies nr. 1
   - Kernel must protect itself

Virtualization, security, isolation ...

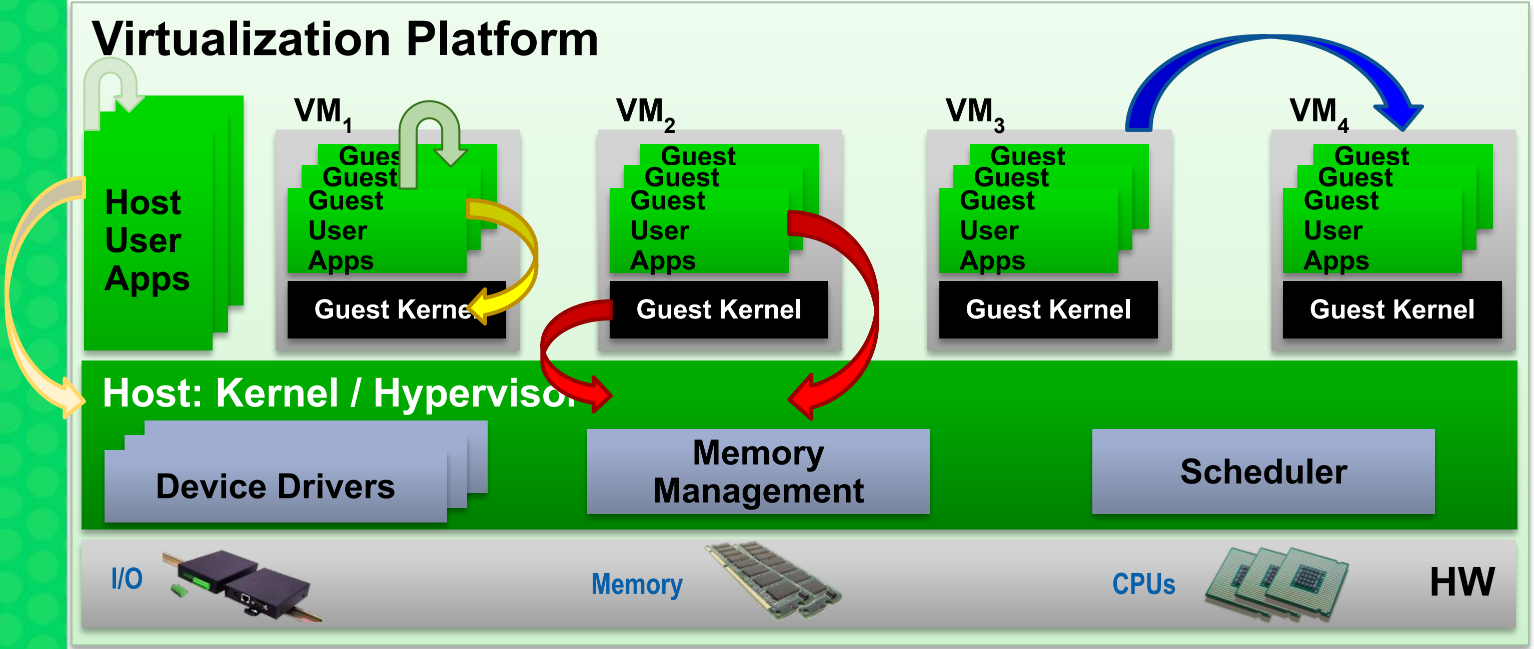


(*) slightly different between Xen and KVM

Virtualization, security, isolation …

Attack Scenarios:

== **successfully** attacked!
(e.g., read data/steal secrets)



- Host User <u>to</u> Other Host User(s)
- Guest User <u>to</u> Other Guest User(s)
- Host User <u>to</u> Host Kernel
- Guest User <u>to</u> Guest Kernel
- Guest <u>to</u> Other Guest(s)
- Guest User <u>to</u> Hypervisor
- Guest Kernel <u>to</u> Hypervisor

Virtualization, security, isolation …

Attack Scenarios:

1. **Host User App to Other Host User Apps(s)**
2. **Guest User App to Other Guest User Apps(s)**
   - Damage contained within App(s) data inside a VM
   - VM user must protect his/her apps
3. **Host User to Host Kernel**
4. **Guest User to Guest Kernel**
   - Implies nr. 2
   - Damage contained within VM/customer
   - Guest kernel must protect itself (mitigations ~= Host User to Host Kernel case)
5. **Guest to Other Guest(s) (*)**
   - VM 3 can steal secrets from VM 4
   - Hypervisor must isolate VMs
6. **Guest to Hypervisor (Bad! Bad! Bad! Bad!) (*)**
   - Damage: implies nr. 5 “on steroids”!
   - Hypervisor must protect itself

(*) we don’t really care if “Guest User to …” or Guest Kernel to …” as one should never trust anything running in a VMs --whatever it is the VM kernel or userspace. Either one (or both!) may have been compromised, and become malicious!

Virtualization, security, isolation ...

Attack Scenarios:

1. **Host User App to Other Host User Apps(s)**
2. **Guest User App to Other Guest User Apps(s)**
   - Damage contained within App(s) data insid
   - VM user must protect his/her apps
3. **Host User to Host Kernel**
4. **Guest User to Guest Kernel**
   - Implies nr. 2
   - Damage contained within VM/customer
   - Guest kernel must protect itself (mitigations ~= Host User to Host Kernel case)
5. **Guest to Other Guest(s) (*)**
   - VM 3 can steal secrets from VM 4
   - Hypervisor must isolate VMs
6. **Guest to Hypervisor (Bad! Bad! Bad! Bad!) (*)**
   - Damage: implies nr. 5 “on steroids”!
   - Hypervisor must protect itself

- Most critical
- Most important, for someone working on OSes & hypervisors (that would be me ;-P)
- Most interesting (personal opinion)

... We’ll focus on these

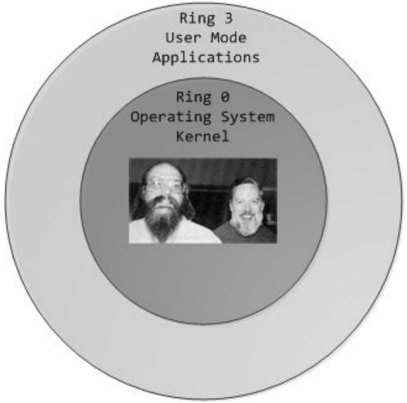
(*) we don’t really care if “Guest User to ...” or Guest Kernel to ...” as one should never trust anything running in a VMs --whatever it is the VM kernel or userspace. Either one (or both!) may have been compromised, and become malicious!

Meltdown

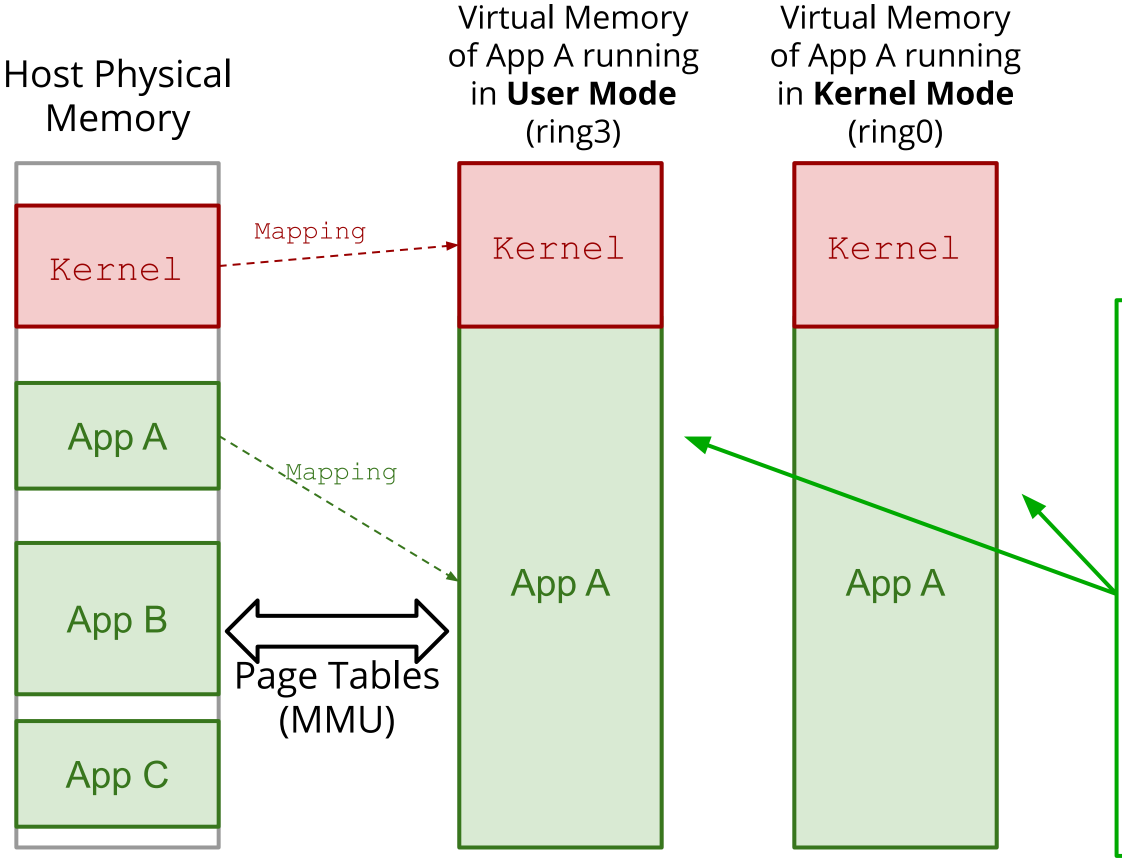
Meltdown (“Spectre v3”)

Rouge Data Cache Load (CVE-2017-5754)

- Virtual Memory, paging, system/user (s/u) bit:
  - Kernel: ring0, can access all memory pages
  - User Apps: ring3, can’t access kernel’s (ring0) pages
- While in speculation:
  - Everyone can access everything!
    - Kernel can read kernel addresses
    - Kernel can read user addresses
    - User can read user addresses
    - **User** can read **kernel** addresses…
- **No** leaky gadget needed in kernel/hypervisor. Attacker can use her own **in user code** (much,much worse than Spectre!)
- Affected **CPUs**: *Intel*, *one ARM CPU*, *PPC* (to some extent… only data in L1, …)



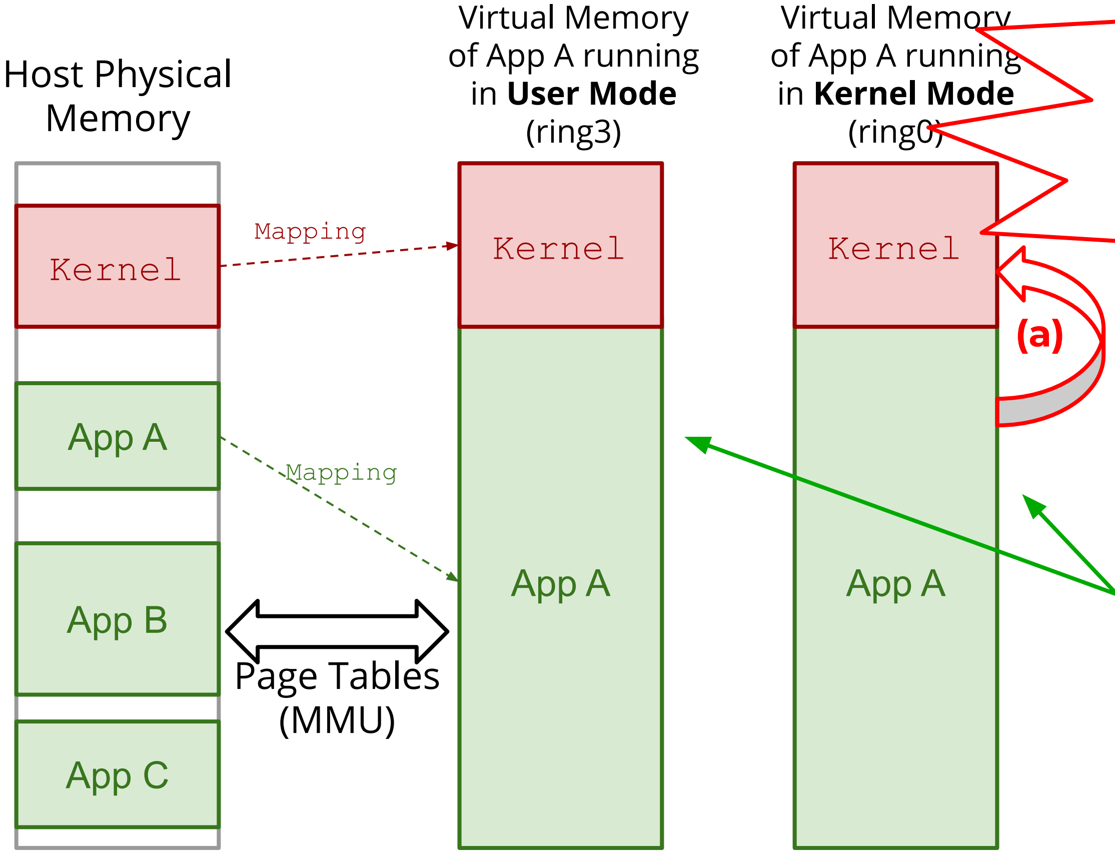
Meltdown


Host Physical Memory
Virtual Memory of App A running in **User Mode** (ring3)
Virtual Memory of App A running in **Kernel Mode** (ring0)
Kernel
Mapping
Kernel
Kernel
App A
Mapping
App A
App A
App B
Page Tables (MMU)
App C

Yes, virtual memory map **is identical** for User App A, when running in both **user and kernel mode**! Why?

- User apps switch from user to kernel mode: e.g., syscall, interrupts, ...
- Changing virtual memory map come at high price: TLB flush
- Kernel is the same for everyone, so, why bother?

Meltdown

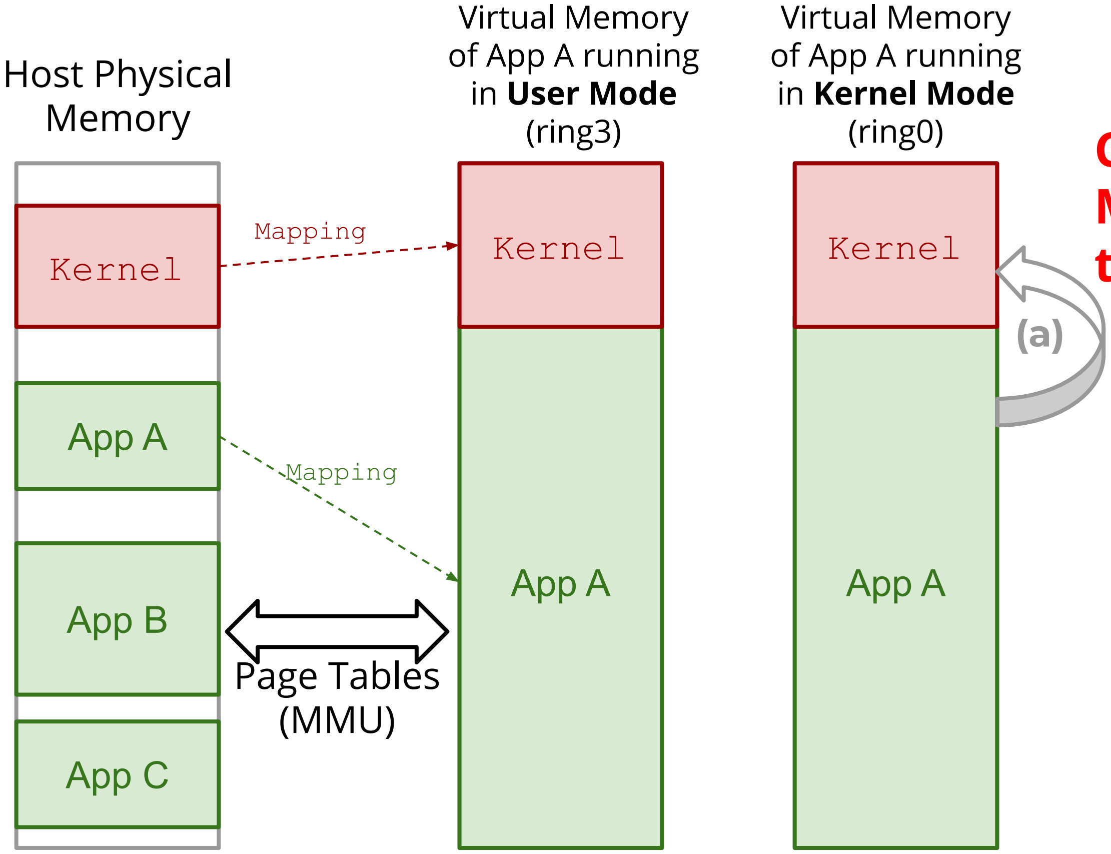

Host Physical Memory
Kernel
App A
App B
App C
Mapping
Mapping
Page Tables (MMU)
Virtual Memory of App A running in **User Mode** (ring3)
Kernel
App A
Virtual Memory of App A running in **Kernel Mode** (ring0)
Kernel
App A
(a)

But… Can't App A, running in User Mode, access Kernel memory then?

Yes, virtual memory map **is identical** for User App A, when running in both **user and kernel mode**! Why?

- User apps switch from user to kernel mode: e.g., syscall, interrupts, …
- Changing virtual memory map come at high price: TLB flush
- Kernel is the same for everyone, so, why bother?

Meltdown


Host Physical Memory
Virtual Memory of App A running in **User Mode** (ring3)
Virtual Memory of App A running in **Kernel Mode** (ring0)
Kernel
Mapping
Kernel
Kernel
App A
Mapping
App A
App A
App B
App C
Page Tables (MMU)
(a)

Can't App A, running in User Mode, access Kernel memory then?

- **Normally:** `s/u` bit in page tables:
  - No, it can't, when in user mode
  - Yes it can, when in kernel mode
- **Speculatively**: `s/u` bit in page tables *ignored*
  - Yes it can, *all the time!*

Meltdown

User space code:

```
int w, x, xx, array[];
if ( <false_but_predicted_as_true> ) {
  w = *((int*)kernel_memory_address);
  x = array[(w & 0x001)];
}
t0 = rdtsc(); xx = array[0]; t0 = rdtsc() - t0
t1 = rdtsc(); xx = array[1]; t1 = rdtsc() - t1
if ( t0 < t1)
  //access to array[0] faster → (* kernel_memory_address)&1 = 0
else
  //access to array[1] faster → (* kernel_memory_address)&1 = 1
```

Meltdown

User space code:

```
int w, x, xx, array[];
if ( <false_but_predicted_as_true> ) {
  w = *((int*) kernel_memory_address );
  x = array[(w & 0x001)];
}
t0 = rdtsc(); xx = array[0]; t0 = rdtsc() - t0
t1 = rdtsc(); xx = array[1]; t1 = rdtsc() - t1
if ( t0 < t1)
  //access to array[0] faster → (* kernel_memory_address )&1 = 0
else
  //access to array[1] faster → (* kernel_memory_address )&1 = 1
```

Trigger speculation:
Make sure the branch is predicted "taken"
(e.g., poison BHB)

Accessed in speculation:

- Privilege check bypassed
- No fault (instruction doesn't retire)

Leaky gadget:

- Load a secret
- Load something else, offseted by that secret

Entirely **under attacker's control**!

Meltdown: Impact

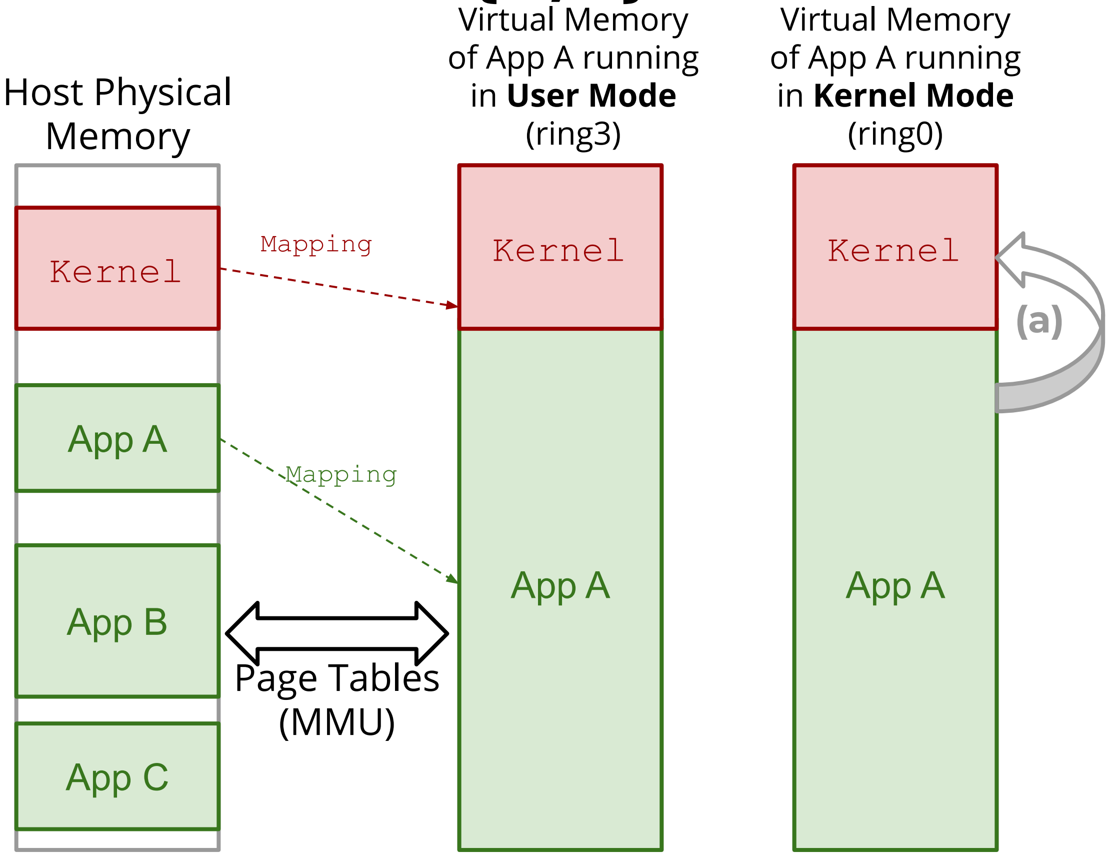
- **Guest User to Guest Kernel (Guest User App to Guest User App(s)):**
  - **KVM:** yes (User to User goes via kernel mappings in User Apps)
  - **Xen HVM, PVH, PV-32bit[1]:** yes (User to User goes via kernel mappings in User Apps)
  - **Xen PV-64bit:** no [2]
- **Guest to Hypervisor (Guest to Other Guest(s)):**
  - **KVM:** no
  - **Xen HVM, PVH, PV-32bit:** no
  - **Xen PV-64bit:** yes :-(([2]
- **Containers:** affected :-((

- **Rather easy** to exploit !

[1] Address space is too small

[2] Looong story… ask offline ;-P

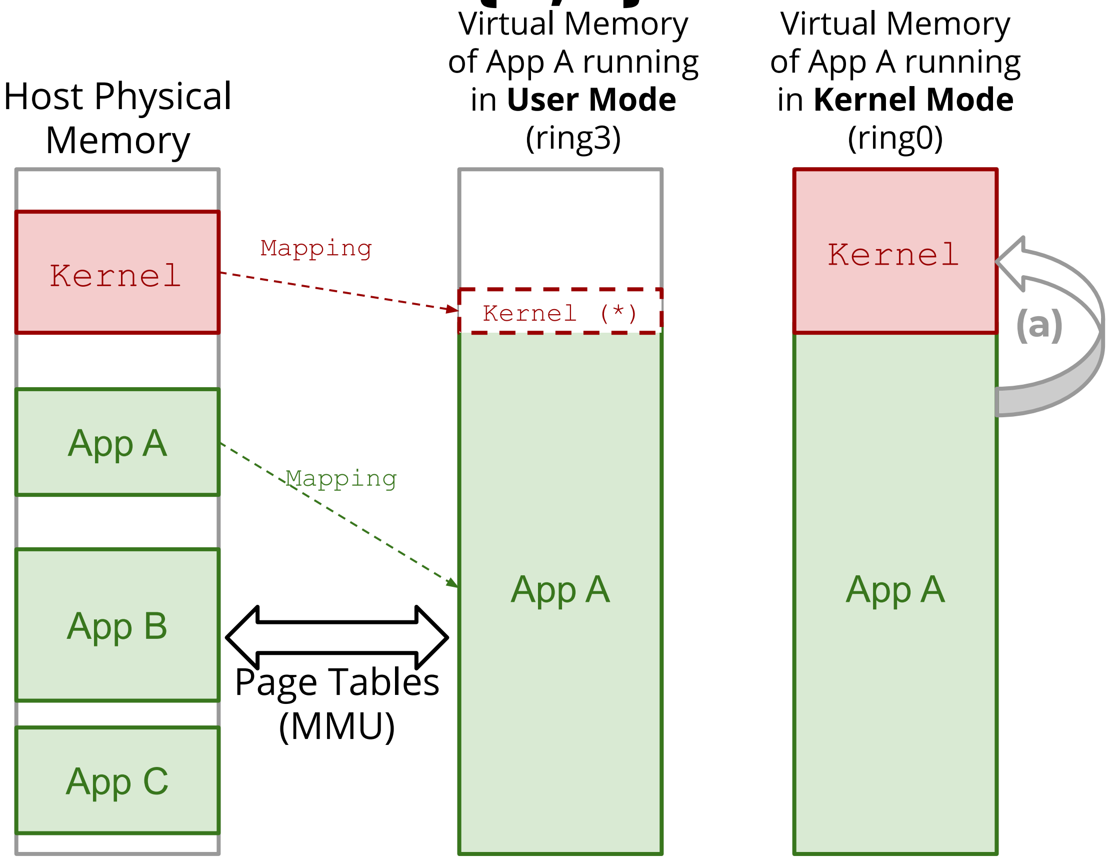
Meltdown: {K,X}PTI


Host Physical Memory
Virtual Memory of App A running in **User Mode** (ring3)
Virtual Memory of App A running in **Kernel Mode** (ring0)
Kernel
App A
App B
App C
Mapping
Mapping
Page Tables (MMU)
Kernel
App A
Kernel
App A
(a)

KPTI / XPTI:
Kernel Page Table Isolation,
Xen Page Table Isolation:

- In speculation CPU can access everything that is mapped

Meltdown: {K,X}PTI

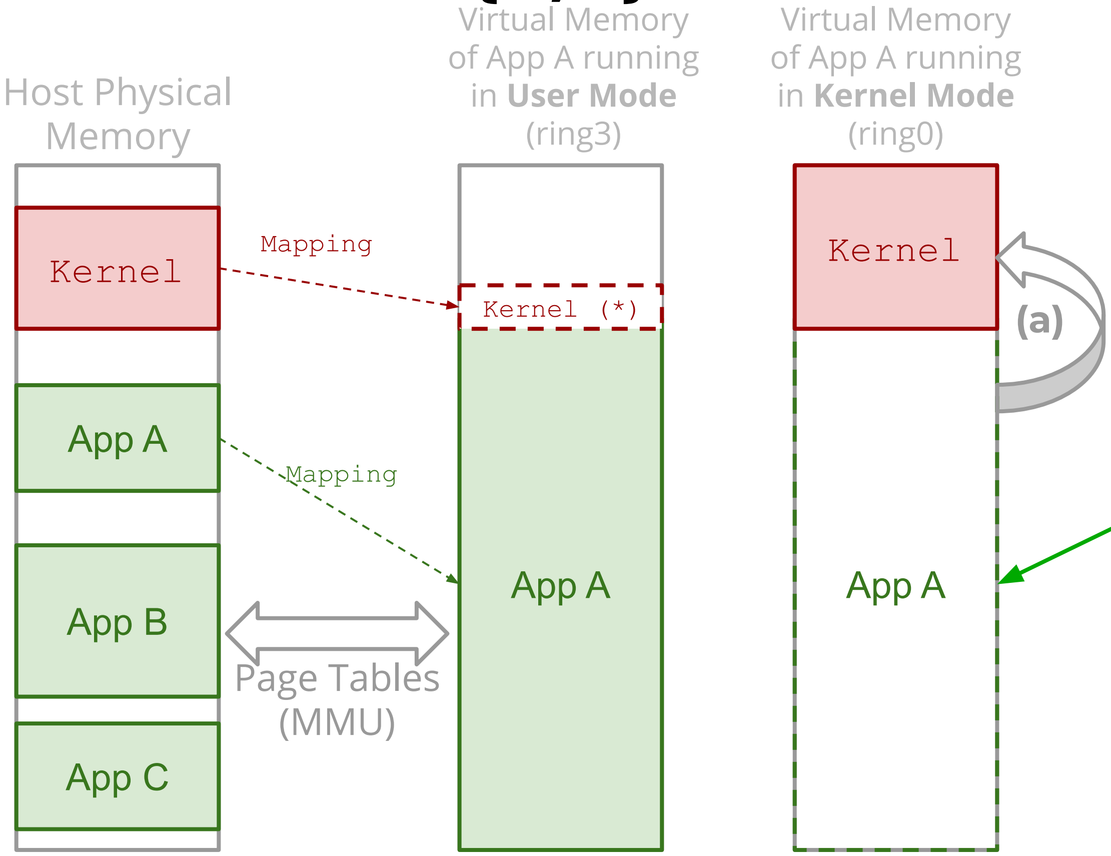

Host Physical Memory
Kernel
App A
App B
App C
Mapping
Mapping
Page Tables (MMU)
Virtual Memory of App A running in **User Mode** (ring3)
Kernel (*)
App A
Virtual Memory of App A running in **Kernel Mode** (ring0)
Kernel
App A
(a)

(*) Only “trampolines” for syscalls, IRQs, ...

KPTI / XPTI:
Kernel Page Table Isolation,
Xen Page Table Isolation:

- In speculation CPU can access everything that is mapped
- **Let’s *not* map everything!** ... ... and pay the price for that! :-(

Meltdown: {K,X}PTI


Host Physical Memory
Kernel
App A
App B
App C
Mapping
Mapping
Page Tables (MMU)
Virtual Memory of App A running in **User Mode** (ring3)
Kernel (*)
App A
Virtual Memory of App A running in **Kernel Mode** (ring0)
Kernel
App A
(a)

(*) Only “trampolines” for syscalls, IRQs, ...

KPTI / XPTI:
Kernel Page Table Isolation,
Xen Page Table Isolation:

- In s… acc… is n…
- **Let’s *not* map everything!** … … … … … … and pay the price for that! :-(

BTW, this is also mapped with NX=1 (if available)

- Not Executable ⇒ similar to SMEP
- Good! (e.g., for Spectre)

Meltdown: PCID

- User Mode ⇒ Kernel Mode (and vice-versa)
  - syscalls, IRQs, ...
  - Change virtual memory layout (CR3 register)
  - Flush **all** TLB (~ page tables cache). *It really hurts performance*
- PCID (Process-Context IDentifier):
  - Tag TLB entries ⇒ ~~flush all **TLB**~~ flush selectively
  - In Intel CPUs since 2010 !!! (PCID in Westmere, INVPCID in Haswell)
- Until now … … …
  - complicate to use, and we map everything anyway, why bother?
- Now (i.e., after Meltdown):
  - *Let's bother!*
  - Used in both Xen and Linux

Meltdown: Mitigation

- **KVM:**
  - Enable KPTI on *host* (protects host kernel from Host User Apps!)
  - Enable KPTI inside *guests*
- **Xen:**
  - Enable XPTI, to protect *PV-64bit* guests (including Dom0!)
  - Enable KPTI inside *HVM*, *PVH* and *PV-32bit* guests
- **Containers:**
  - Enable KPTI on *host*

Meltdown:
Performance Impact

Expected: from **-5%** to **-30%** performance impact

- Workload dependant: worse if I/O and syscall intensive
- Slowdowns of more than **-20%** reached only on synthetic benchmarks (e.g., doing lots of tiny I/O)
- For “typical” workloads, we’re usually **well within -10%** ...
- ... with PCID support!
  - LKML posts: postgres -5%, haproxy -17%
  - Brendan Gregg - KPTI/KAISER Meltdown Initial Performance Regressions
  - Gil Tene - PCID is now a critical performance/security feature on x86

Spectre

Spectre v1

Bounds-Check Bypass (CVE-2017-5753)

- Attacks conditional branch prediction
- Vulnerable code (**leaky gadget**) must be present in target, or JIT (*)
- Affected **CPUs**: *everyone* (Intel, AMD, ARM)

```
uint8_t arr_size, arr[];      //array_size not in cache
Uint8_t arr_size, arr2[];     //elements 1 and 2 not in cache
//untrusted_index_from_attacker = <out of array[] boundaries>
if ( untrusted_index_from_attacker < arr_size ) {
  val = arr[untrusted_index_from_attacker];
  idx2 = (val&1) + 1;
  val2 = arr2[idx2]; //arr2[1] in cache ⇒ (arr[untrusted_index]&1) = 0
}                    //arr2[2] in cache ⇒ (arr[untrusted_index]&1) = 1
```

(*) Just in Time code generators

Spectre v1

Bounds-Check Bypas

- Attacks conditio

de (**leaky gadget**) must be present in target, or JIT (*)

: *everyone* (Intel, AMD, ARM)

Target == kernel/hypervisor (Linux if KVM, Xen if Xen). **Not really common!**

Xen: no JIT,
KVM: eBPF

Trigger speculation:
Make sure the branch is predicted “taken”
(e.g., poison BHB)

```
uint8_t arr_size, arr[];        //array_size not in cache
Uint8_t arr_size, arr2[];       //elements 1 and 2 not in cache
//untrusted_index_from_attacker = <out of array[] boundaries>
if ( untrusted_index_from_attacker < arr_size )
  val = arr[untrusted_index_from_attacker];
  idx2 = (val&1) + 1;
  val2 = arr2[idx2]; //arr2[1] in cache ⇒ (arr[untrusted_index]&1) = 0
}                    //arr2[2] in cache ⇒ (arr[untrusted_index]&1) = 1
```

Leaky gadget:

- Load a secret
- Leak it, by loading something else, offseted by that secret

(*) Just in Time code generators

Spectre v1: Impact, mitigations, performance

- Impact:
  - **Guest User App to Guest User App(s):** yes (JIT, e.g., Javascript in browsers)
  - **Guest User to Guest Kernel, Guest to Hypervisor, Containers:** well, theoretically (leaky gadgets or JIT in kernel/hypervisor)
- **Extremely hard** to exploit
- Mitigation:
  - none... **wait, what?**
  - Manual code sanitization (a.k.a. playing the whack-a-mole game!)
  - array_index_mask_nospec(), in Xen & Linux, to stop speculation
- Performance Implications: **none** (clever Tricks to avoid “fencing” ...)

Spectre v2

Branch Target Injection(CVE-2017-5715)

- Attacks indirect branch prediction: *function pointers* / `jmp *(%r11)`
- Attacker *might* be able to provide his own **leaky gadget**
- Affected **CPUs**: everyone (Intel, AMD, ARM)

Predictors of indirect branch targets:

- Are based on previous history (BTB); can be “poisoned”
- Branches done in userspace influence predictions in kernel space
- Branches done in SMT thread influence predictions on sibling

Attack:

- Same leaky gadget based strategy (PoC for KVM via eBPF)
- *Attacker provided* leaky gadget if !SMEP on the CPU (on x86)

Marc Zyngier - KVM/arm Meets the Villain: Mitigating Spectre

Very good talk about ARM specifics challenges

Spectre v2

Indirect jump:

```
     Address   Instruction
(1)  0x001123  jmp *(%r11)          //r11 = 0xddeeff
     ...       ...
     ...       ...
     0xaabbcc  <my leaky gadget>    //either target's or
     ...       ...                  //attacker's code
     ...       ...
(2)  0xddeeff  <xxx>
               <yyy>
```

Regular Execution:

- We are at **(1)**
- We jump at **(2)**

Indirect branch:

- Function pointer
- Which function is pointed is predicted

Spectre v2

Indirect jump:

```
        Address   Instruction
(1) (1) 0x001123  jmp *(%r11)          //r11 = 0xddeeff
        ...       ...                          0xaabbcc
        ...       ...
(1s)    0xaabbcc  <my leaky gadget>      //either target's
        ...       ...                  //attacker's code
        ...       ...
    (2) 0xddeeff  <xxx>
                  <yyy>
```

Regular Execution:

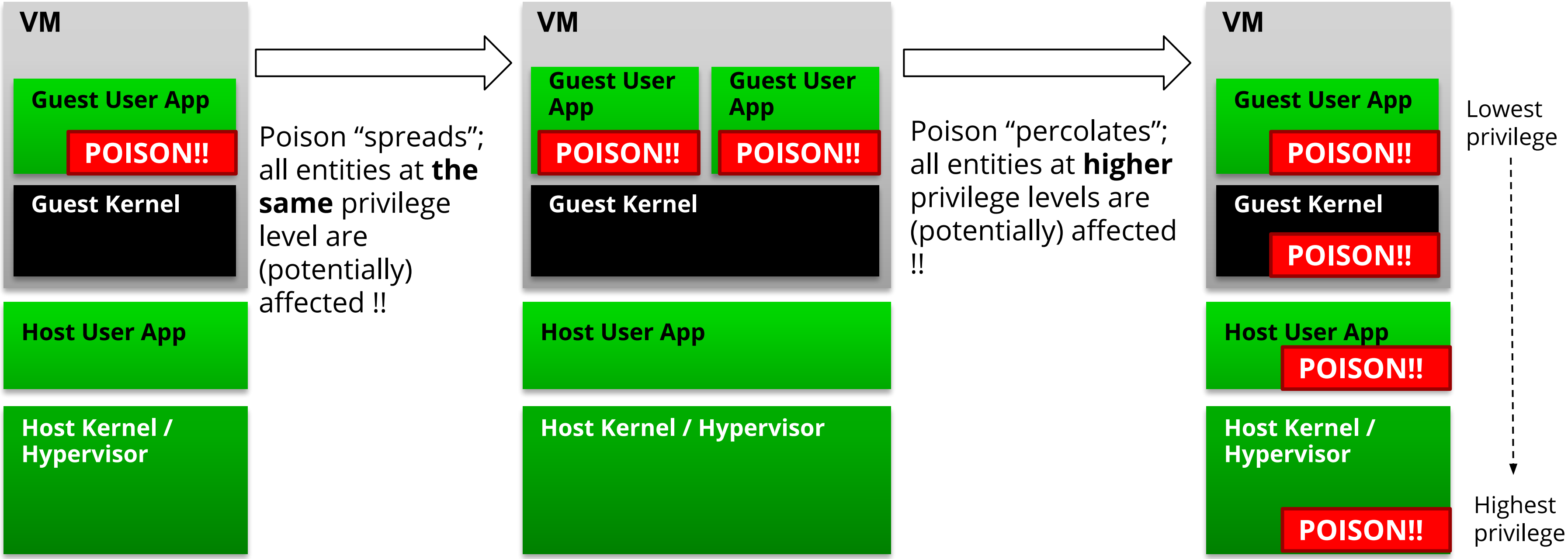
- We are at **(1)**
- We jump at **(2)**

Speculative Execution (Attack):

- We poison BTB to think that `r11 = aabbcc`
- We are at **(1)**
- We enter speculation at **(1s)**, where's the leaky gadget

Spectre v2 (& v1 !): Branch Predictor Poisoning

Guest User App “produce” Poison ⇒ BTB in the CPU



Poison “spreads”; all entities at **the same** privilege level are (potentially) affected !!

Poison “percolates”; all entities at **higher** privilege levels are (potentially) affected !!

Spectre v2: Impact

- **Guest User to Guest Kernel, (Guest User App to Guest User App(s)):** yes (JIT, e.g., Javascript in browsers)
- **Guest to Other Guest(s):** yes (via Guest to Hypervisor)
- **Guest to Hypervisor:** yes (*existing* leaky gadget if SMEP, or via JIT)
- **Containers:** affected

- **Reasonably hard** to exploit, exp. for vitrtualization

SMEP: Supervisor Mode Exec. Protection (Fischer, Stephen (2011-09-21))

- Kernel won't execute User App code
- We can't make kernel speculatively jump to a User App provided leaky gadget

Spectre v2: retpoline

… by Google

Let’s set up a trap for speculation:

```
jmp *%r11
```

```
  call set_up_target;
capture_spec:
  pause; lfence;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);
  ret;
```

Replacement can happen:

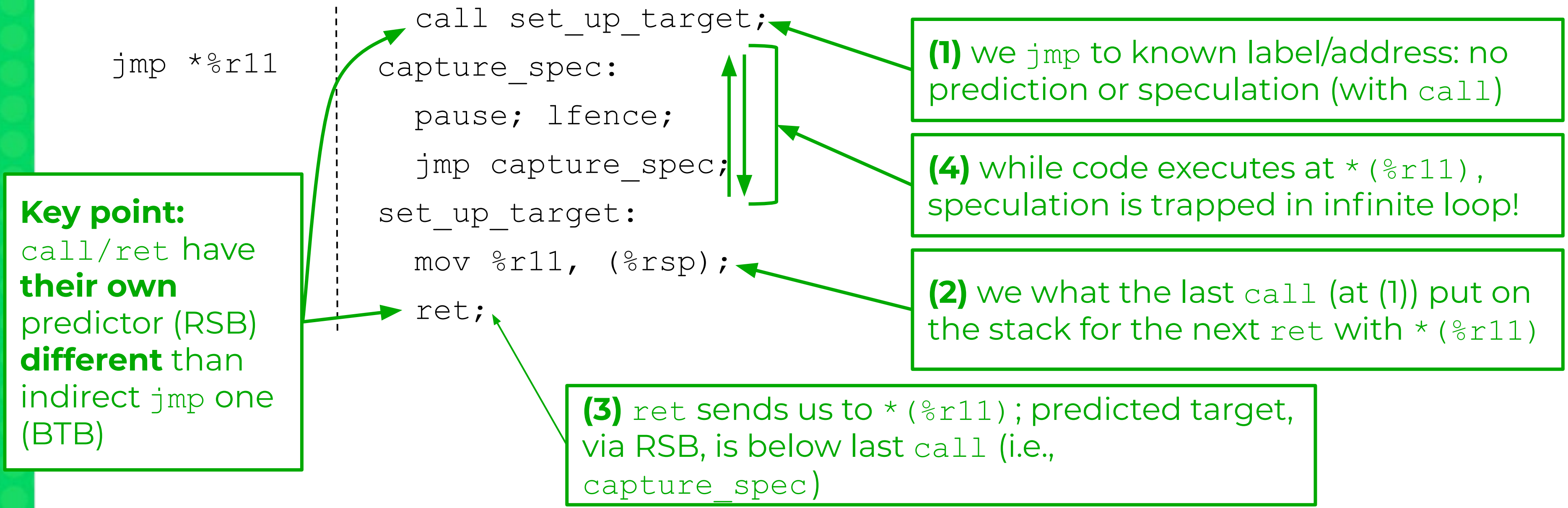
- kernel/hypervisor & userspace: compiler support (<<yay, let’s recompile everything!>> :-/)
- kernel/hypervisor: binary patching (e.g., Linux’s alternatives)

Spectre v2: retpoline

… by Google

Key point: `call/ret` have **their own** predictor (RSB) **different** than indirect `jmp` one (BTB)

Let’s set up a trap for speculation:



```
jmp *%r11
```

```
  call set_up_target;
capture_spec:
  pause; lfence;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);
  ret;
```

(1) we `jmp` to known label/address: no prediction or speculation (with `call`)

(4) while code executes at `*(%r11)`, speculation is trapped in infinite loop!

(2) we what the last `call` (at (1)) put on the stack for the next `ret` with `*(%r11)`

(3) `ret` sends us to `*(%r11)`; predicted target, via RSB, is below last `call` (i.e., `capture_spec`)

Key point: `call/ret` have **their own** predictor (RSB) **different** than indirect `jmp` one (BTB)

- Skylake+: ret target **might be** predicted with BTB lwn.net/Articles/745111/
- **“RSB Stuffing”** Retpoline: A Branch Target Injection Mitigation

Spectre v2:
IBPB, STIBP, IBRS

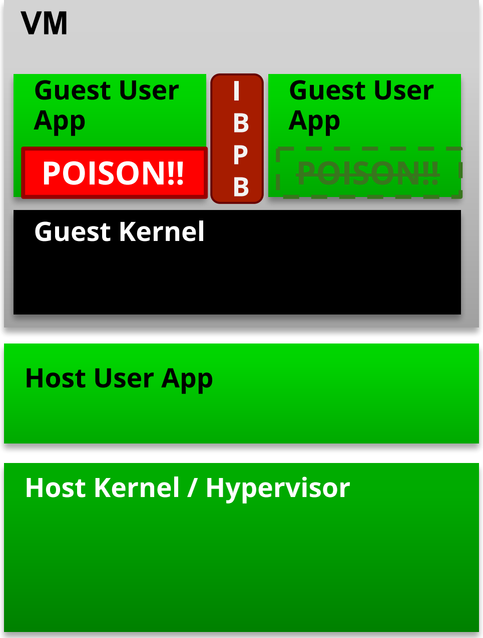
Firmware/Microcode update (e.g., from Intel).
~~Gross hacks~~... ahem.. New “instructions”:

- **IBPB:** flush all branch info learned so far
- **STIBP:** ignore info of branches done on sibling hyperthread
- **IBRS:** ignore info of branches done in a less-privileged mode (before it was most recently set)

Intended usage:

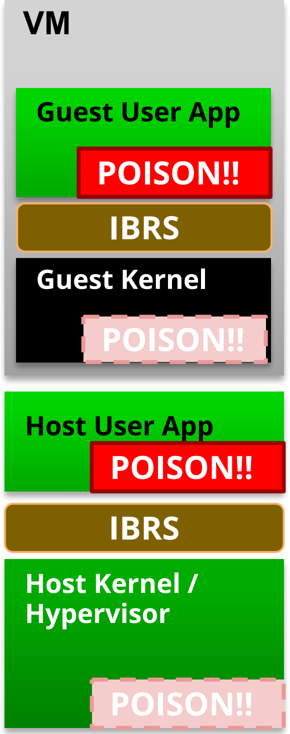
- **IBPB:** on context and/or vCPU switch. Prevents App/VM A influencing (poisoning?) branch predictions of App/VM B
- **STIBP:** when running with HT. Prevents App/VM running on thread influencing (poisoning?) branch predictions of App/VM on sibling
- **IBRS:** when entering kernel/hypervisor. Prevents Apps/VMs influencing (poisoning?) branch predictions in kernel/hypervisor

Spectre v2: IBPB, STIBP, IBRS



IBPB neutralizes BTB poison "horizontally"

(e.g., between processes)



IBRS neutralizes BTB poison "vertically"

(e.g., between priv. levels)

Spectre v2: Mitigation(s)

- **User Apps:**
  - retpoline
  - Make timer less precise ⇒ harder to measure side effects!
  - IBPB & STIBP (Spectre v2 app2app, in these days)
- **Xen:** tries to pick best combo at boot
  - retpoline, when safe. IBRS, when reptoline-unsafe
  - IBPB at VM switch
  - Clear RSB on VM switch
- **KVM:**
  - reptoline + some IBRS (e.g., when calling into firmware)
  - IBPB at VM switch (heuristics for IBPB at context switch)
  - Clear RSB on context/VM switch
- **Both Xen, KVM:** IBRS, IBPB, STIBP available/virtualized for VMs too

Spectre v2: Performance Impact

SPECTRE

It’s complicated!

- **retpoline:** good performance… is it enough ~~paranoia~~ protection?
- **IB*** barriers:
  - **IBPB:** *moderate* impact
  - **IBRS:** impact *varies a lot*, depending on hardware
  - **STIBP:** (these days) *huge* impact ⇒ making it per-app opt-in

    E.g. Intel:
    - pre-Skylake: super-bad
    - post-Skylake: not-too-bad
  - ⇒ it’s not only the flushing
    - x86 : these are, for now, MSR write (**sloooow**!)
    - ARM: on one CPU, disable/re-enable the MMU! :-O

Spectre (Again!)

Spectre v3a (Spectre-NG)

Rogue System Register Read (CVE-2018-3640)

- Speculative reads of system registers may leak info about system status (e.g., flags)

Spectre v4 (Spectre-NG)

Speculative Store Bypass (SSB) (CVE-2018-3639)

- Affected CPUs: *everyone* (Intel, AMD, ARM)
- in speculation, a load from an address can observe the result of a store which is **not the latest** store to that address:

  -
    ```
    STO 1 → R1
    STO 2 → R1
    (in speculation) LOAD R1 ⇒ sees 1 !!!
    ```
  - E.g.:
    ```
    user:    syscall() ← pass data
    Kernel:  copy data on stack
    … … … …
    Kernel:  store data on stack
    Kernel:  load data from stack ⇒ sees previous user provided data !!!
    ```
- Similar to Spectre v1: needs leaky gadget or JIT
- New instruction SSDB ⇒ no use Xen/KVM, useful for User Apps in guests

LazyFP (Spectre v5)

Lazy FPU State Leak (CVE-2018-3665)

- Affected CPUs: *Intel*
- FPU context is **large**
  - let’s *ignore* it at context switches
  - Mark it as invalid
  - **If** new context (process/VM) needs it: save it, switch it and mark as valid again
- Speculative execution:
  - New context needs it ⇒ uses it **right away**, in speculation, with old context’s values in it!
  - “old context’s values”: how about *keys* or *crypto stuff*?!?!
- XSAVEOPT …

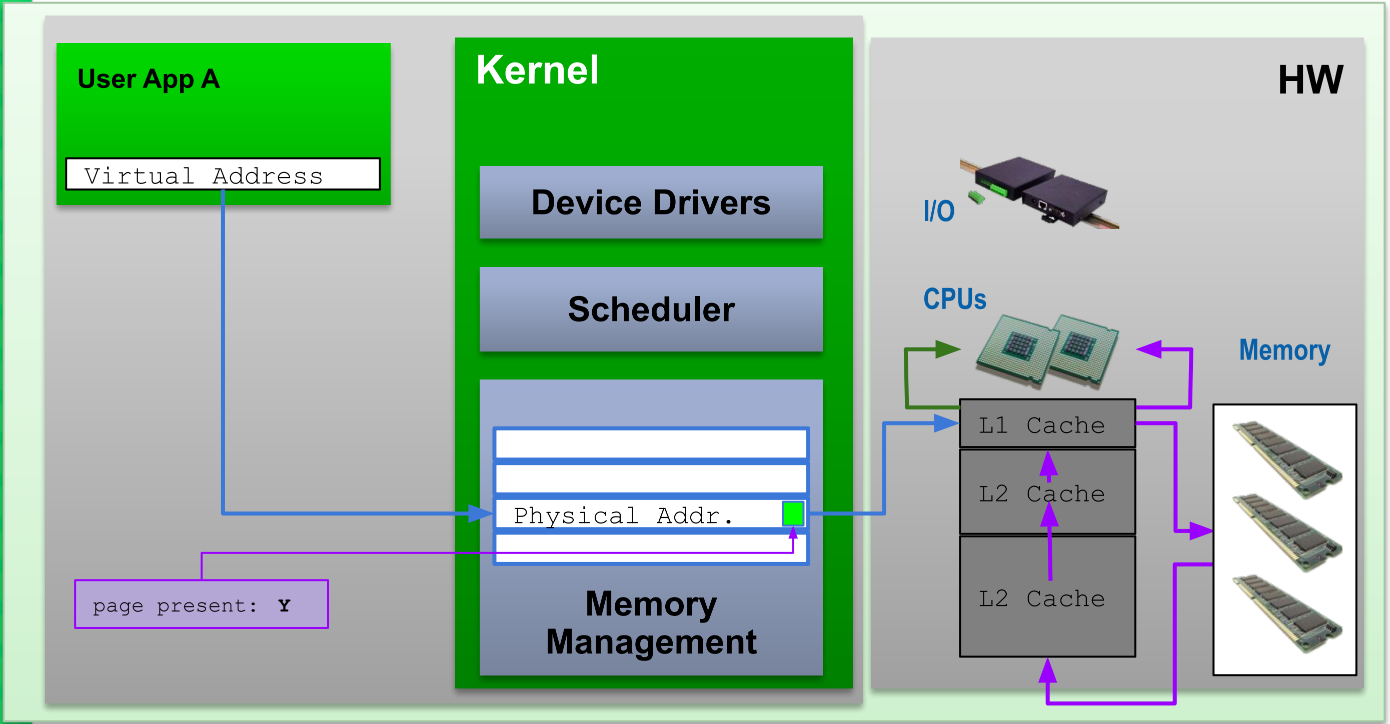
L1TF (Foreshadow / ForeshadowNG)

L1TF - Baremetal (Foreshadow)

L1TF / Foreshadow (CVE-2018-3620)

- Similar to Meltdown, potentially
- Meltdown: user space can read kernel pages,
  if they're mapped in its address space
  - `s/u` bit in page table entries, ignored, in speculation
  - User space manages to maliciously read (in speculation) all its *virtual addresses*
- L1TF: user space can kind of read *physical memory directly*!
  - `present` bit in page table entries ignored, in speculation
  - That means it can maliciously read (in speculation) *all RAM* ⇒ PTI is useless
- Affects **only** Intel (~= Meltdown)

L1TF - Baremetal



Regular execution

App accesses data in present page:

1. Page tables
2. Check L1 cache

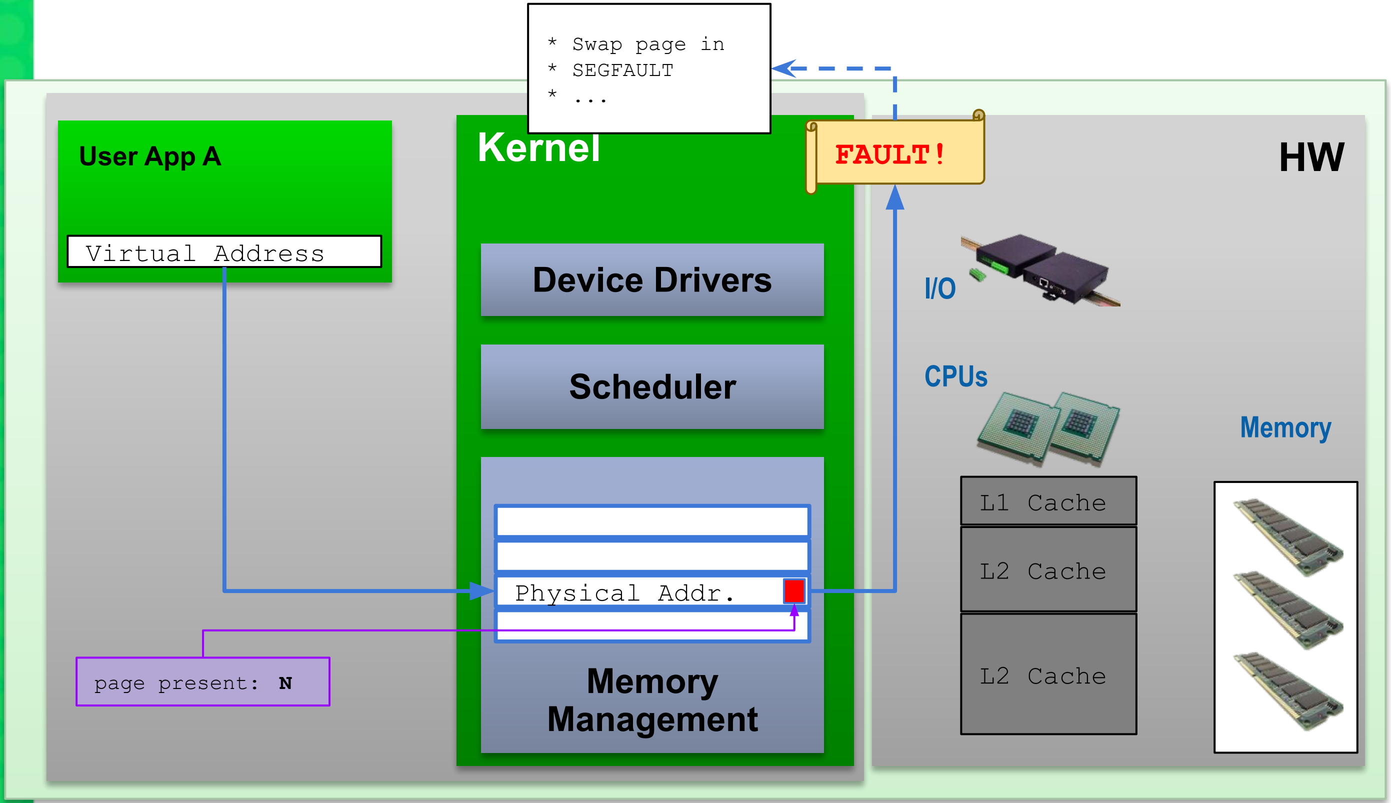
---

3. **Hit!** Load data in CPU

---

4. **Miss!** Fetch from L2/L3/RAM
5. Load in L3, L2, L1
6. Load in CPU

L1TF - Baremetal



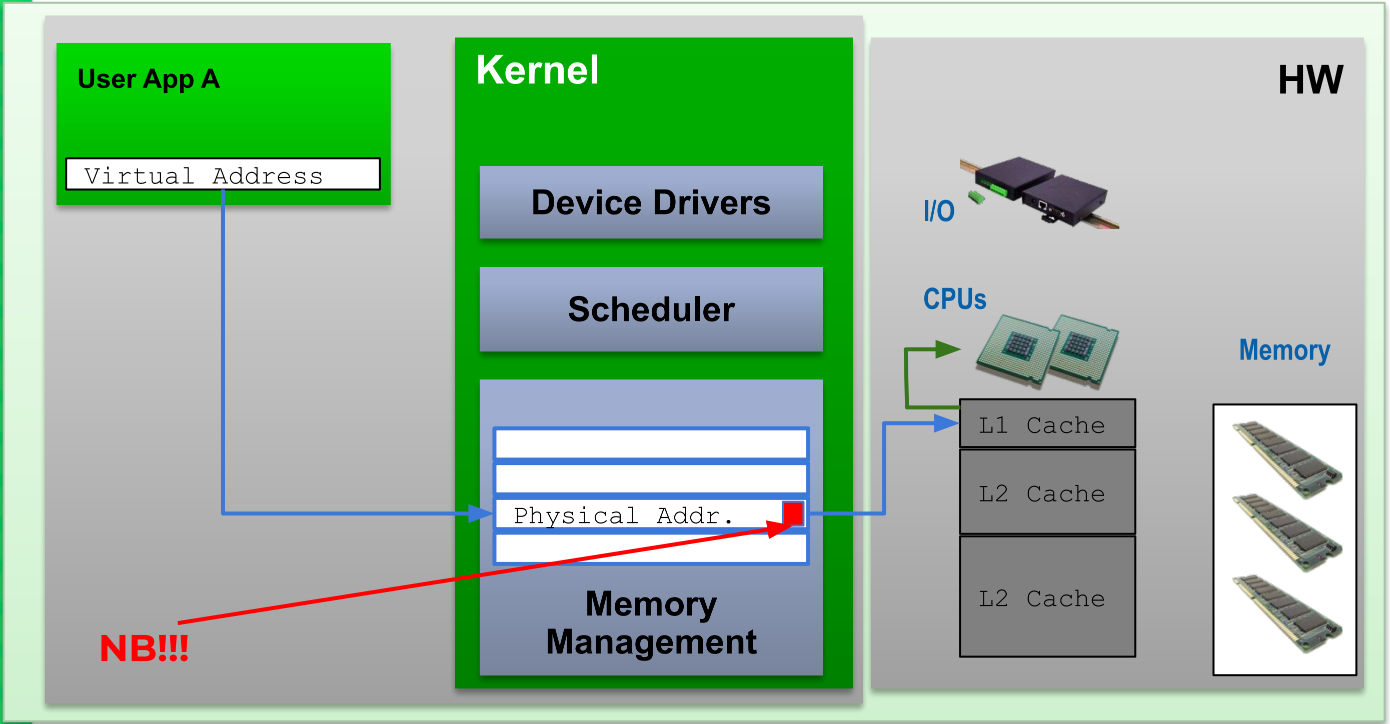
Regular execution

App accesses data in present page:

1. Page tables
   page !present
2. Page fault

Potentially Malicious App A: **stopped!**

L1TF - Baremetal



Speculative execution

App accesses data in present page:

1. Page tables

   ~~page !present~~

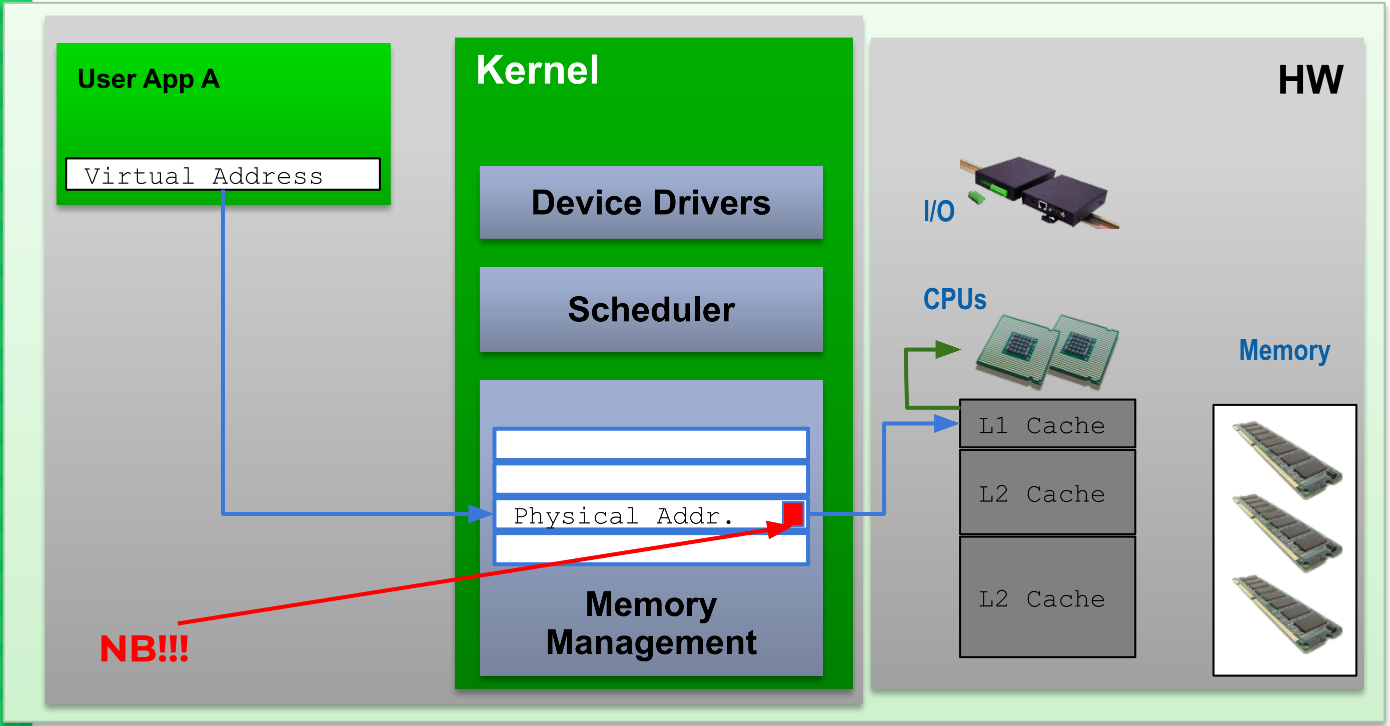
~~2. Page fault~~

2. Check L1 cache

   ---

3. **Hit!** Load data in CPU

L1TF - Baremetal



Speculative execution

App accesses data in present page:

1. Page tables

~~page !present~~

~~2. Page fault~~

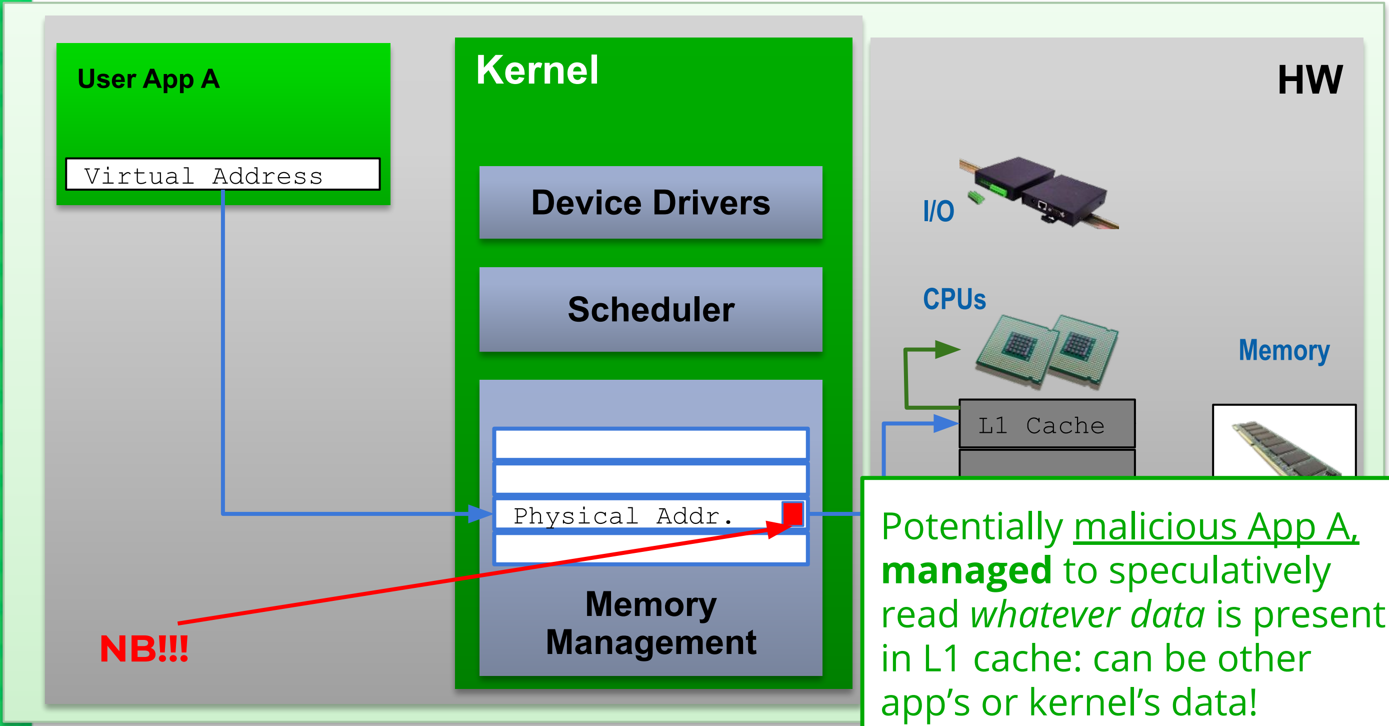
2. Check L1 cache

---

3. **Hit!** Load data in CPU

Wait… What?!?!

L1TF - Baremetal



Potentially <u>malicious App A,</u> **managed** to speculatively read *whatever data* is present in L1 cache: can be other app's or kernel's data!

Speculative execution

App accesses data in present page:

1. Page tables

   ~~page !present~~

~~2. Page fault~~

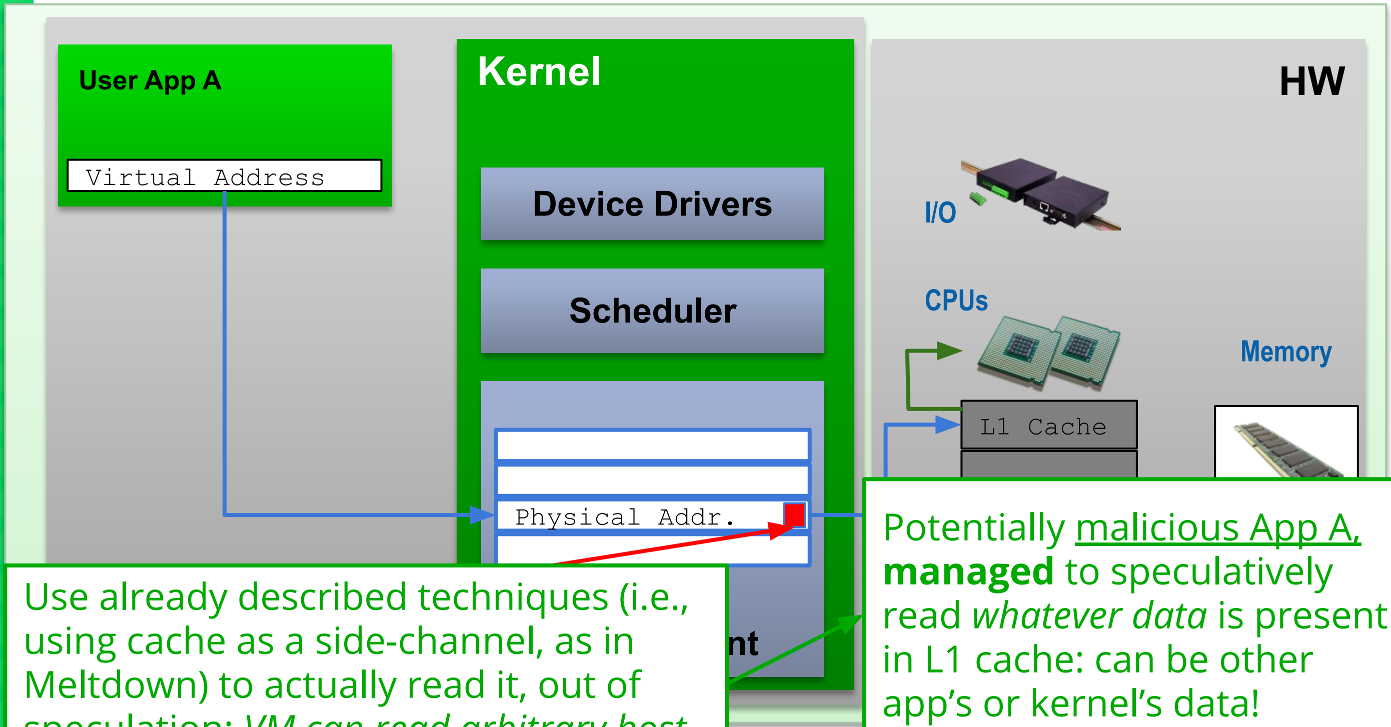
2. Check L1 cache

   ---

3. **Hit!** Load data in CPU

Wait... What?!?!

L1TF - Baremetal



Potentially malicious App A, **managed** to speculatively read *whatever data* is present in L1 cache: can be other app's or kernel's data!

Use already described techniques (i.e., using cache as a side-channel, as in Meltdown) to actually read it, out of speculation: *VM can read arbitrary host data!!*

Speculative execution

App accesses data in present page:

1. Page tables

   ~~page !present~~

~~2. Page fault~~

2. Check L1 cache

   ---

3. **Hit!** Load data in CPU

Wait... What?!?!

L1TF - Baremetal (Foreshadow)

Problems:

- What does the !present page table entry (PTE) contains?
  - Can be anything. Intel manual explicitly say the content will be ignored
  - OS is free to use it at will
  - Linux, Windows, etc.: offset of the page in swap space

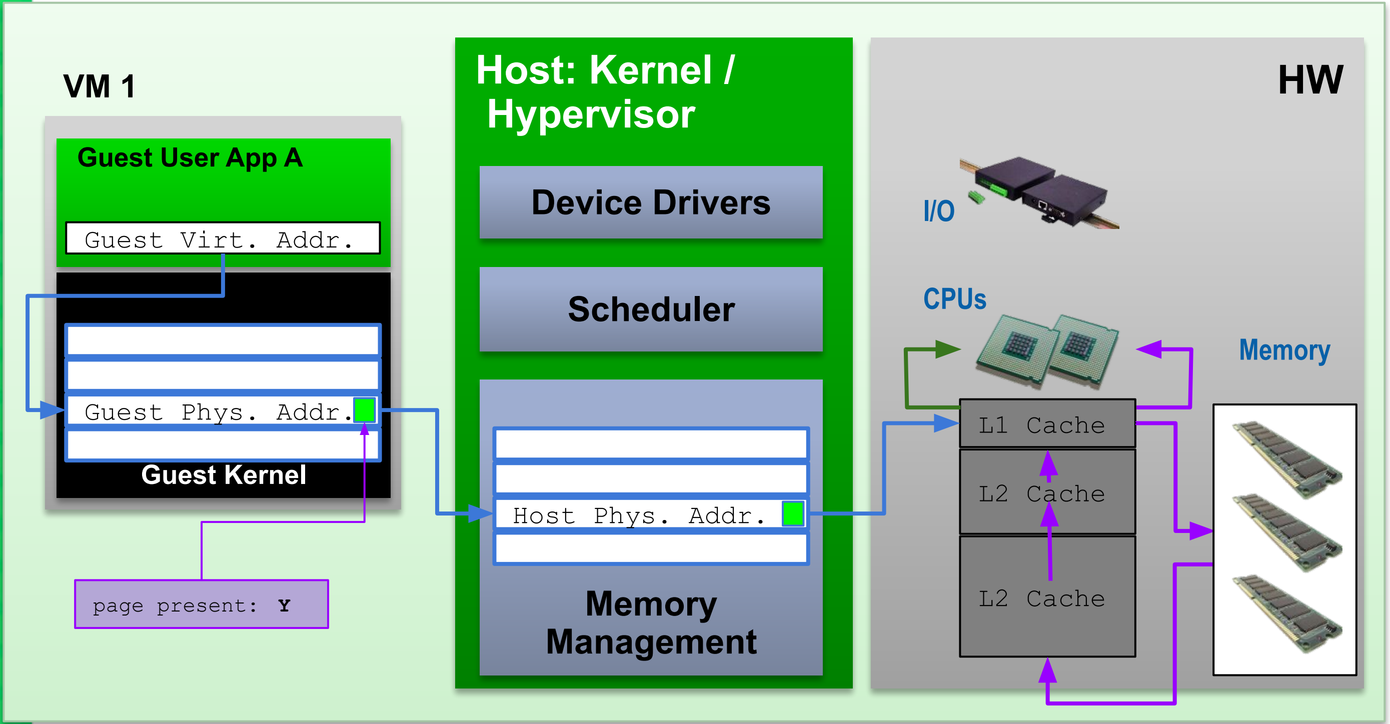
- Can an attacker process control its own (!present) PTEs?
  - The kernel is in charge of PTEs, … … …
  - … … … yeah, but, e.g., mprotect() (Linux syscall)
  - So, **yes**, it’s possible!

L1TF - Virtualization (Foreshadow-NG)

L1TF / Foreshadow-NG (CVE-2018-3646)

- Like Meltdown. But **scarier**. And almost **harder to fix** (for virt)!
- Meltdown: user space can read kernel pages,
  if they're mapped in its address space
  - `s/u` bit in page table entries, ignored, in speculation
  - User space manages to maliciously read (in speculation) all its *virtual addresses*
- L1TF: guests can kind of read *physical memory directly*!
  - `present` bit in page table entries ignored, in speculation
  - Guest manages to maliciously read (in speculation) *all RAM* ⇒ PTI is useless
  - … … … and, believe me, **it gets worse** !!!
- Affects **only** Intel (~= Meltdown)

L1TF - Virtualization



Regular execution

App accesses data in present page:

1. Guest page tables
2. Host page tables
3. Check L1 cache

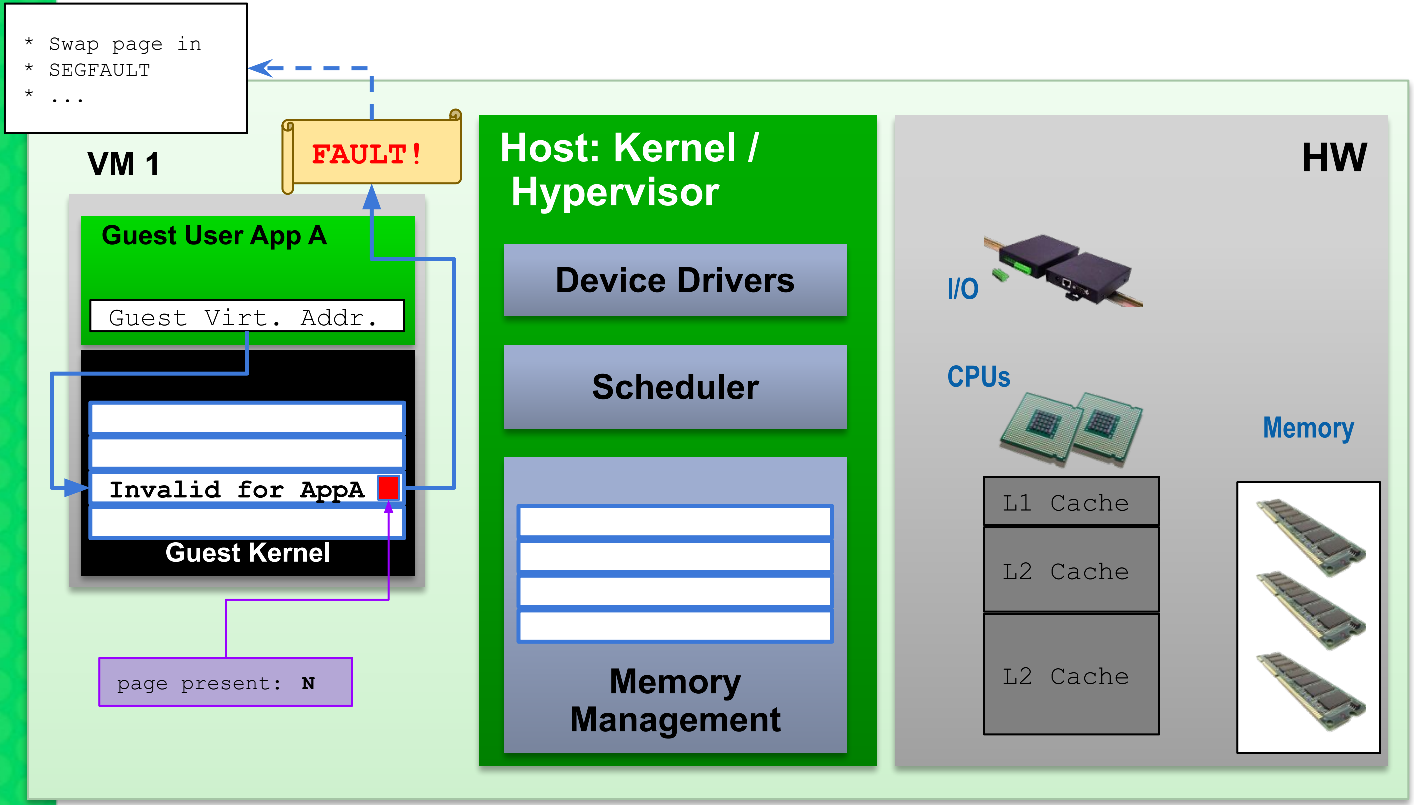
---

4. **Hit!** Load data in CPU

---

4. **Miss!** Fetch from L2/L3/RAM
5. Load in L3, L2, L1
6. Load in CPU

L1TF - Virtualization



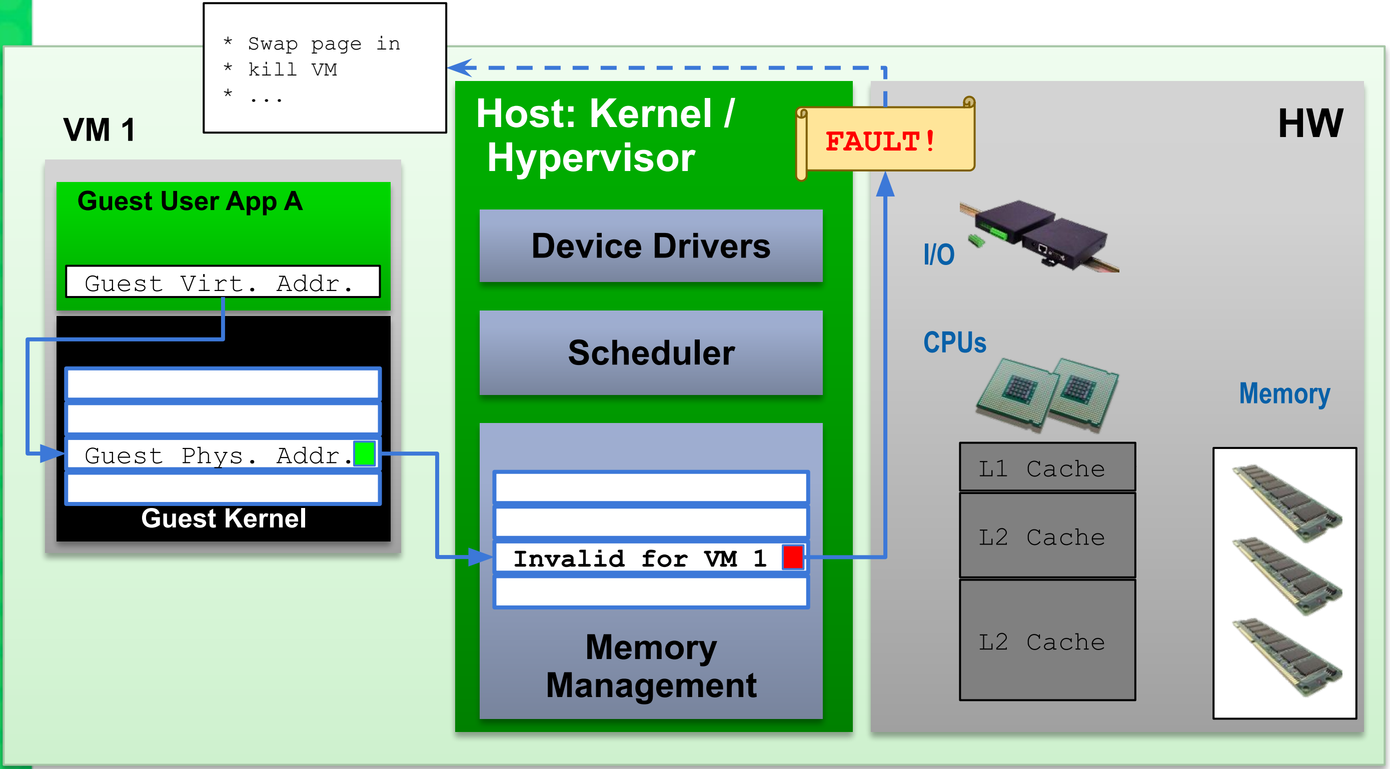
Regular execution

App accesses data in non present page:

1. Guest page tables page !present
2. Guest page fault

Potentially Malicious App A (e.g., trying to steal data within VM 1): **stopped!**

L1TF - Virtualization



Regular execution

Guest accesses data in non present page:

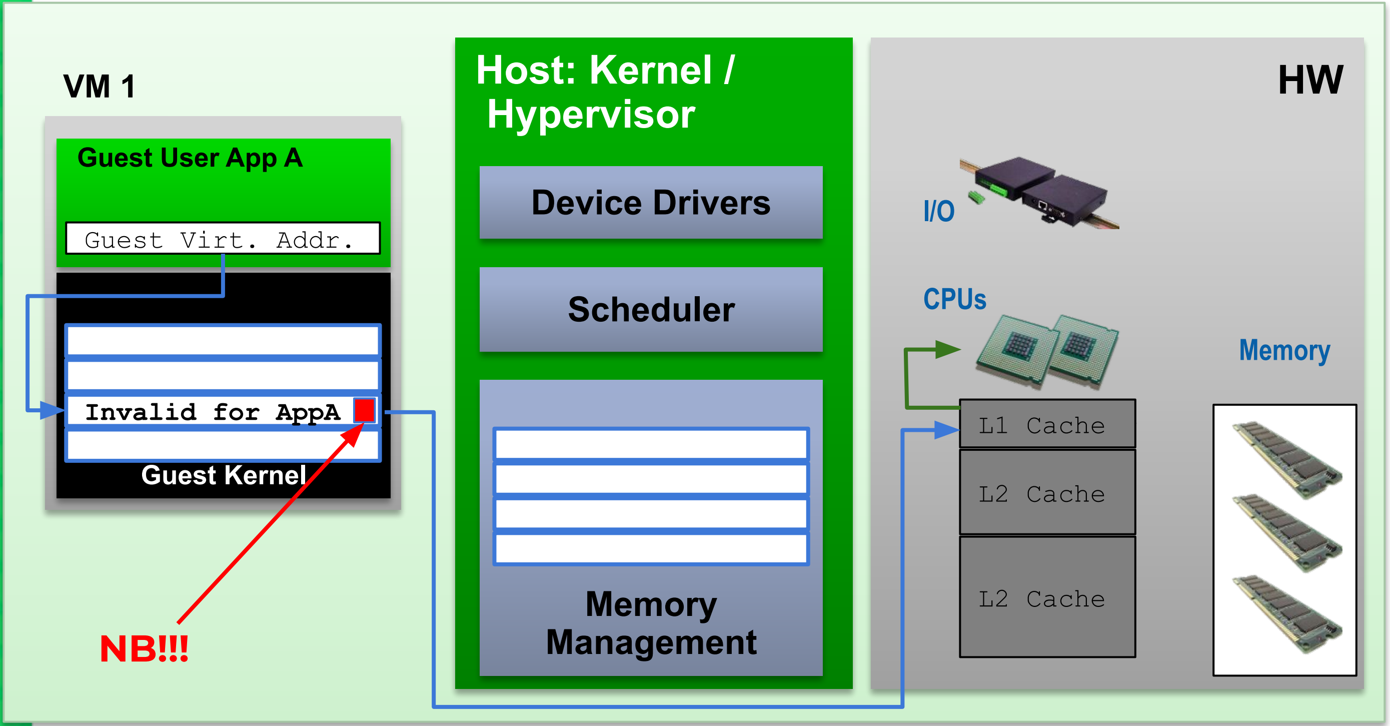
1. Guest page tables
2. Host page tables

   page !present

3. Host page fault

Potentially malicious App A, or VM 1 (or both), trying to steal from host or other VMs: **stopped!**

L1TF - Virtualization



Speculative execution

App (speculatively) accesses data in non present page:

1. Guest page tables

   ~~page !present~~

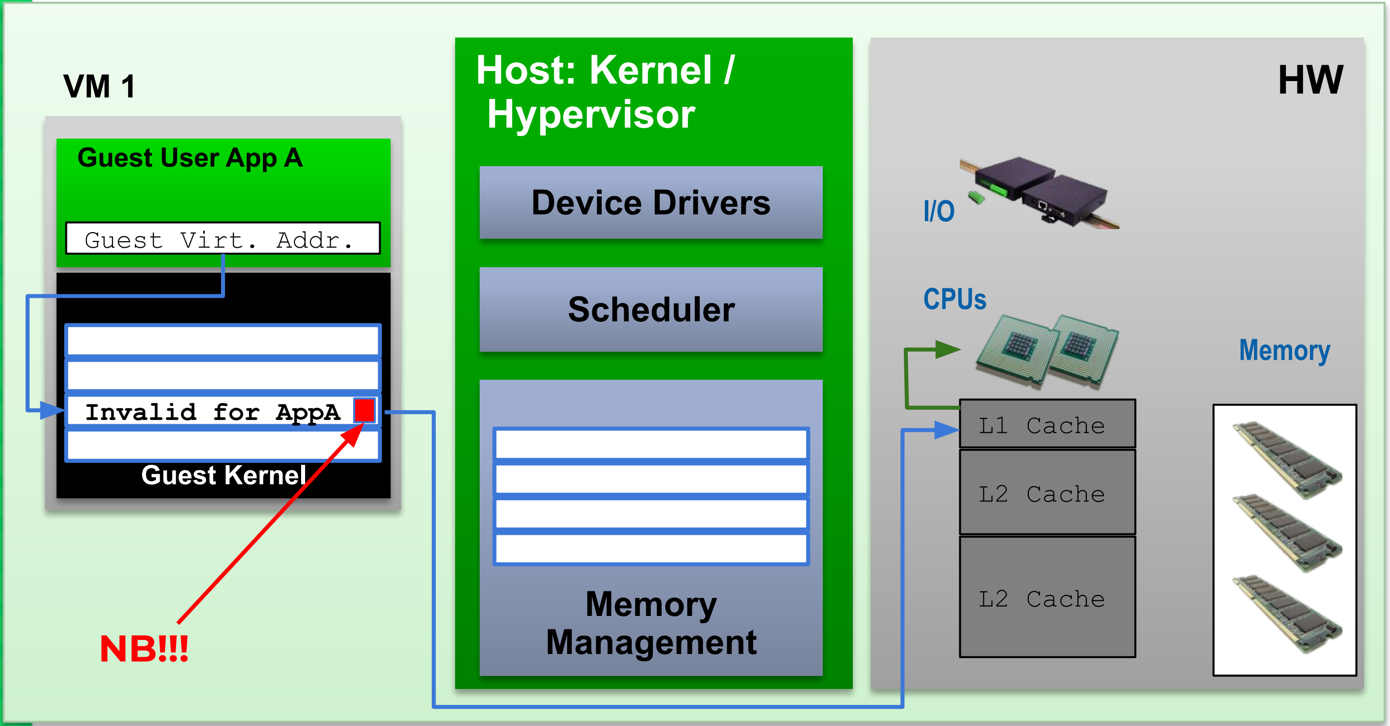
~~2. Host page tables~~

2. Check L1 cache

   ---

3. **Hit!** Load data in CPU

L1TF - Virtualization



Speculative execution

App (speculatively) accesses data in non present page:

1. Guest page tables
   ~~page !present~~

~~2. Host page tables~~

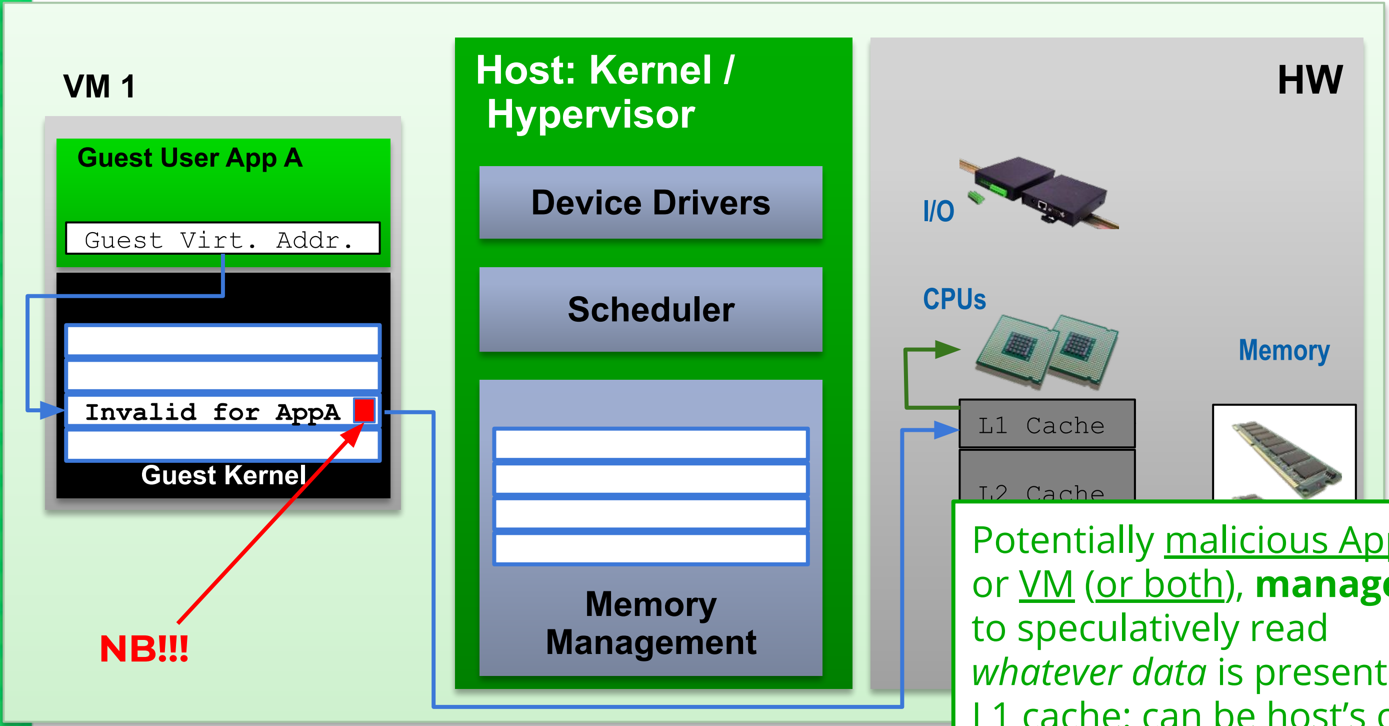
2. Check L1 cache

   ---

3. **Hit!** Load data in CPU

Wait... What?!?!

L1TF - Virtualization



Potentially malicious App A, or VM (or both), **managed** to speculatively read *whatever data* is present in L1 cache: can be host's or other VMs' secrets!

Speculative execution

App (speculatively) accesses data in non present page:

1. Guest page tables

   ~~page !present~~

~~2. Host page tables~~

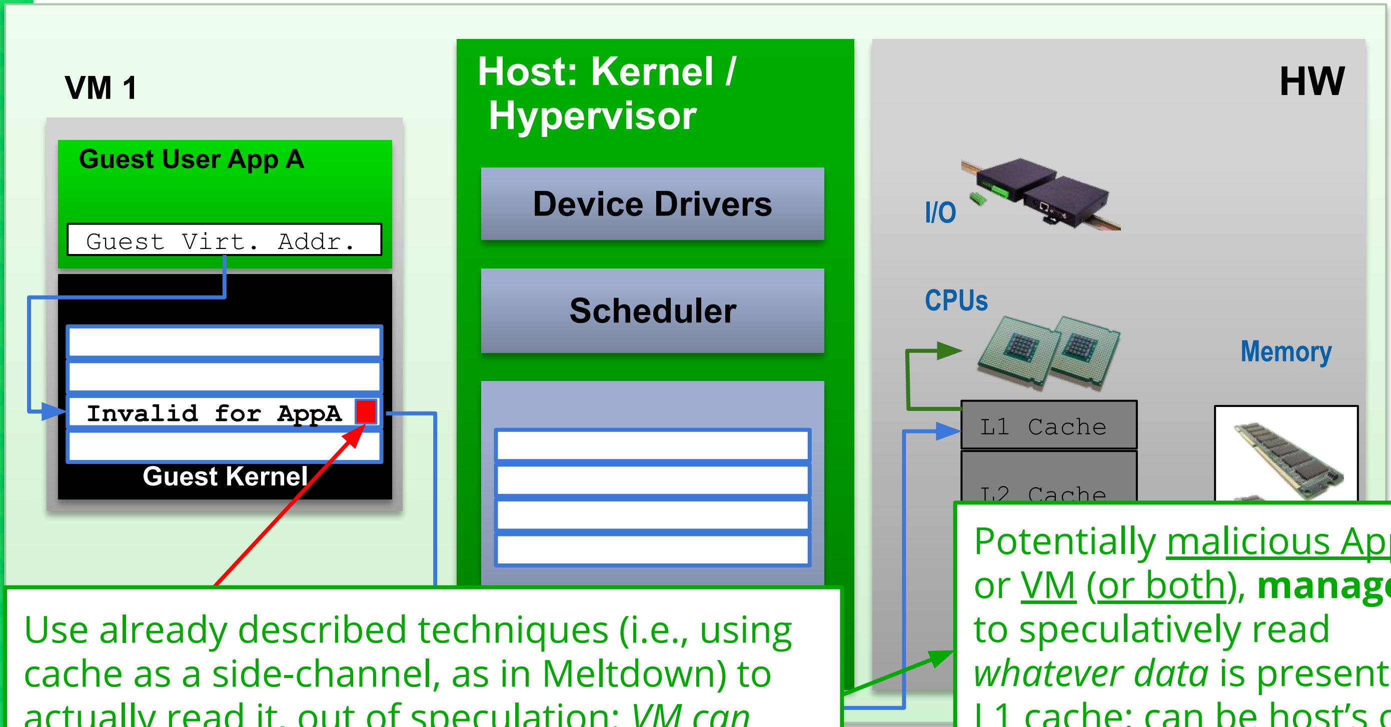
2. Check L1 cache

   ---

3. **Hit!** Load data in CPU

Wait... What?!?!

L1TF - Virtualization



Use already described techniques (i.e., using cache as a side-channel, as in Meltdown) to actually read it, out of speculation: *VM can read arbitrary host data!!*

Potentially malicious App A, or VM (or both), **managed** to speculatively read *whatever data* is present in L1 cache: can be host's or other VMs' secrets!

Speculative execution

App (speculatively) accesses data in non present page:

1. Guest page tables

   ~~page !present~~

~~2. Host page tables~~

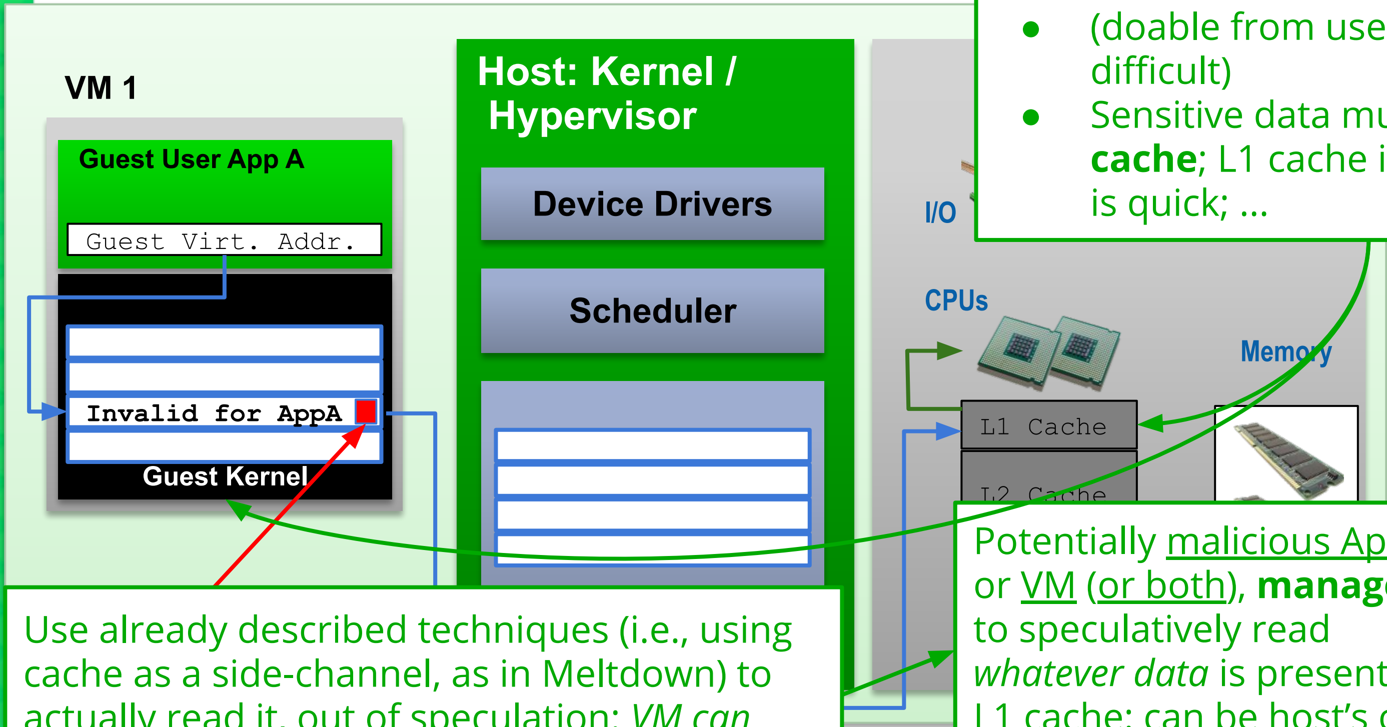
2. Check L1 cache

   ---

3. **Hit!** Load data in CPU

Wait... What?!?!

L1TF - Virtualization



Is this **really** dangerous?

- Attacker must control **VMs' kernel** ⇒ generate malicious guest addresses
- (doable from userspace, but really difficult)
- Sensitive data must be **in host's L1 cache**; L1 cache is small; turnaround is quick; ...

tion

n

bles

~~page !present~~

~~2. Host page tables~~

2. Check L1 cache

---

3. **Hit!** Load data in CPU

Wait… What?!?!

Use already described techniques (i.e., using cache as a side-channel, as in Meltdown) to actually read it, out of speculation: *VM can read arbitrary host data!!*

Potentially malicious App A, or VM (or both), **managed** to speculatively read *whatever data* is present in L1 cache: can be host's or other VMs' secrets!

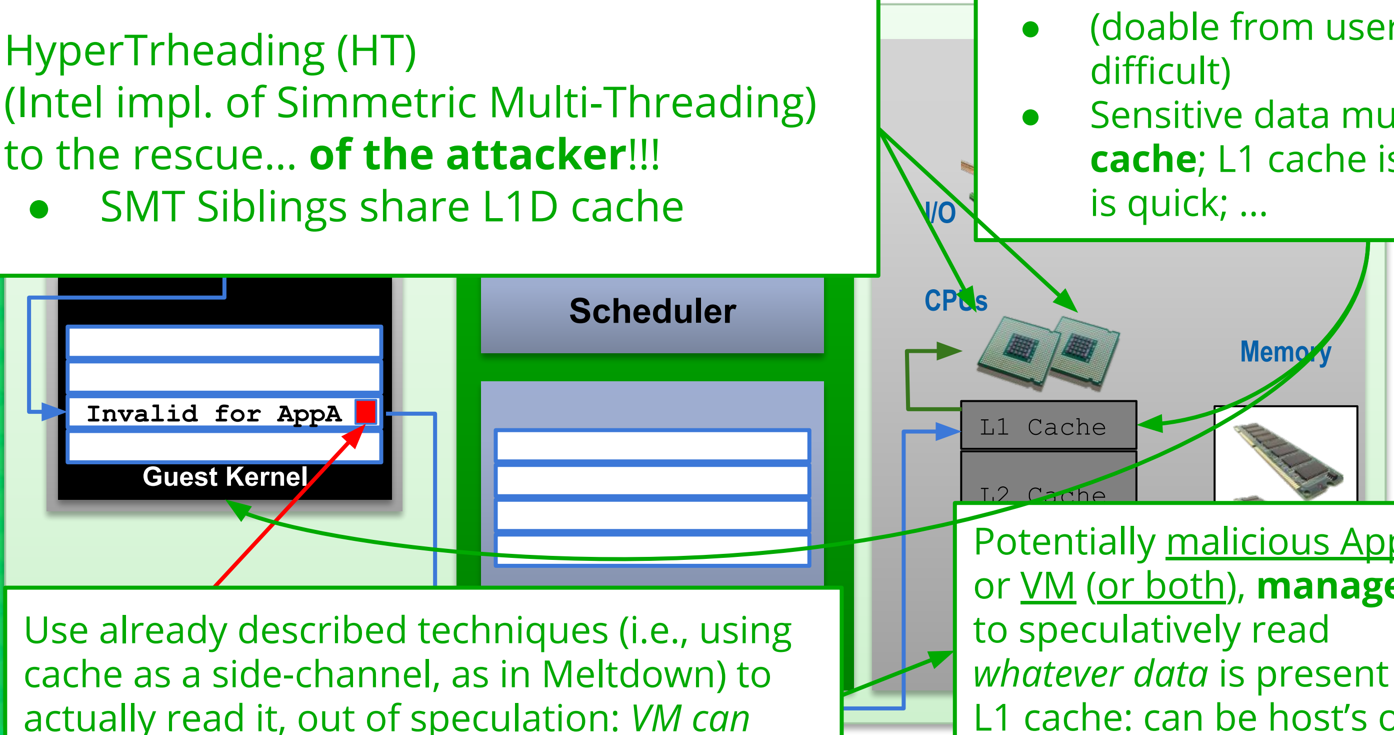
L1TF - Virtualization

HyperTrheading (HT)
(Intel impl. of Simmetric Multi-Threading)
to the rescue... **of the attacker**!!!

- SMT Siblings share L1D cache

Is this **really** dangerous?

- Attacker must control **VMs' kernel** ⇒ generate malicious guest addresses
- (doable from userspace, but really difficult)
- Sensitive data must be **in host's L1 cache**; L1 cache is small; turnaround is quick; ...



Use already described techniques (i.e., using cache as a side-channel, as in Meltdown) to actually read it, out of speculation: *VM can read arbitrary host data!!*

Potentially malicious App A, or VM (or both), **managed** to speculatively read *whatever data* is present in L1 cache: can be host's or other VMs' secrets!

tion

n

bles

~~page !present~~

~~2. Host page tables~~

2. Check L1 cache

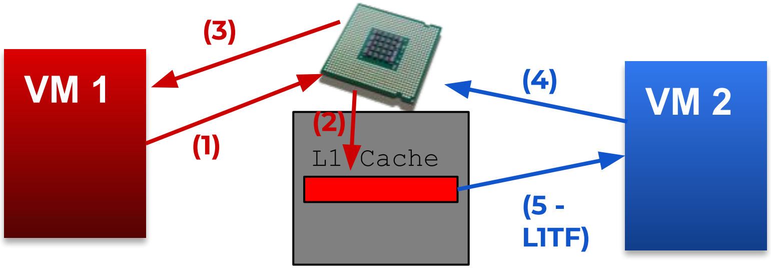
---

3. **Hit!** Load data in CPU

Wait... What?!?!

L1TF: HyperThreading

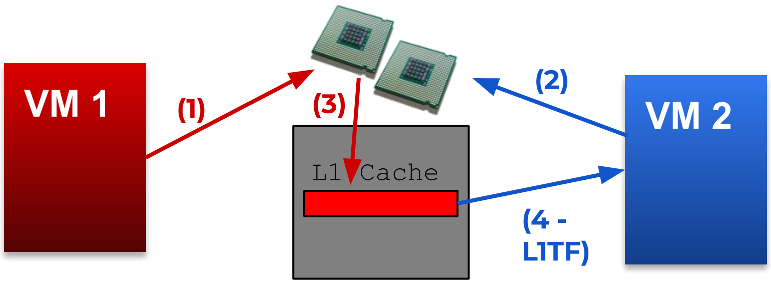
Without Hyperthreading:



1. VM 1 runs on CPU
2. VM 1 puts secrets in L1 cache
3. VM 1 leaves CPU
4. VM 2 runs on CPU
5. **VM 2 reads VM 1's secrets!**

Context Switch

With Hyperthreading:



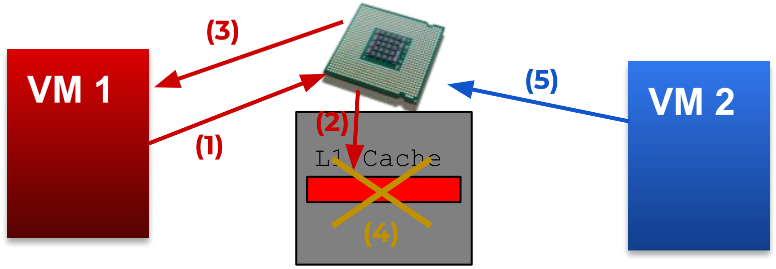
1. VM 1 runs on Thread A
2. VM 2 runs on Thread B
3. VM 1 puts secrets in L1 cache
4. **VM 2 reads VM 1's secret from L1 cache**

No context switch needed...

Guest (Kernel) to Other Guest(s) attack

L1TF: HyperThreading

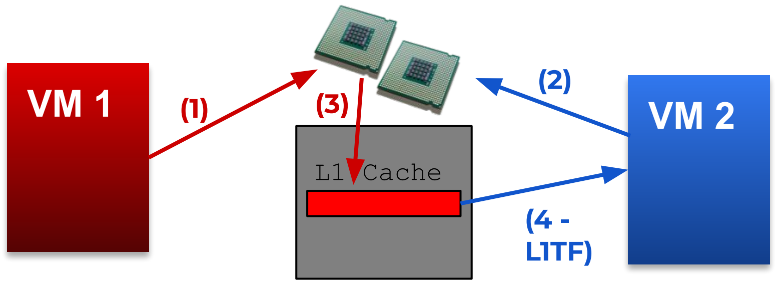
Without Hyperthreading: mitigation



1. VM 1 runs on CPU
2. VM 1 puts secrets in L1 cache
3. VM 1 leaves CPU
4. Hypervisor: flush L1 cache
5. VM 2 runs on CPU
6. ~~VM 2 reads VM 1's secrets!~~

Context Switch

With Hyperthreading: err... mitigation?

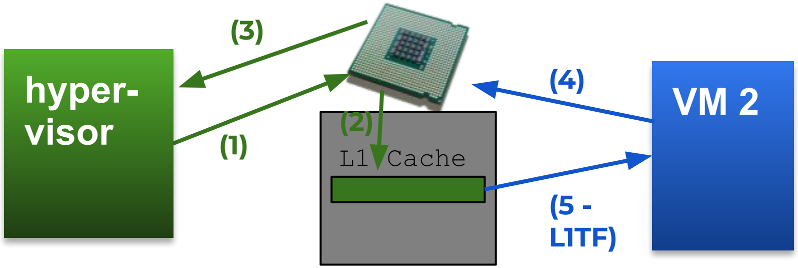


1. VM 1 runs on Thread A
2. VM 2 runs on Thread B
3. VM 1 puts secrets in L1 cache
   Hypervisor: THERE'S NOTHING I CAN DO !!!
4. **VM 2 reads VM 1's secret from L1 cache**

Guest (kernel) to Other Guest(s) attack

L1TF: hyperthreading

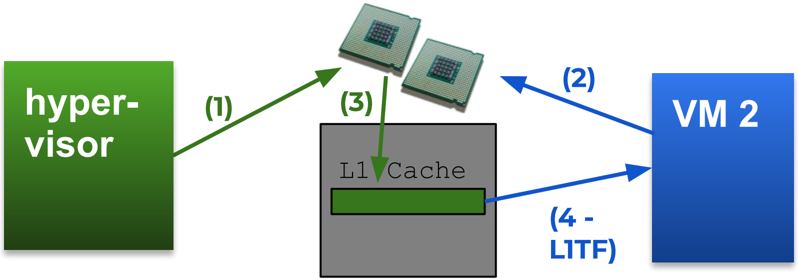
Without Hyperthreading:



1. Hypervisor runs on CPU
2. Hypervisor puts secrets in L1
3. Hypervisor leaves CPU
4. VM 2 runs on CPU
5. **VM 2 reads hypervisor's secrets!**

VMEntry

With Hyperthreading:



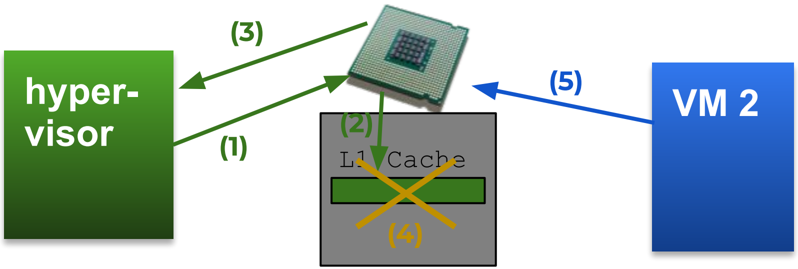
1. Hypervisor runs on Thread A
2. VM 2 runs on Thread B
3. Hypervisor puts secrets in L1
4. **VM 2 reads VM 1's secret from L1 cache**

No VMEntry needed...

Guest Kernel to Other Guest(s) attack

L1TF: hyperthreading

Without Hyperthreading: mitigation

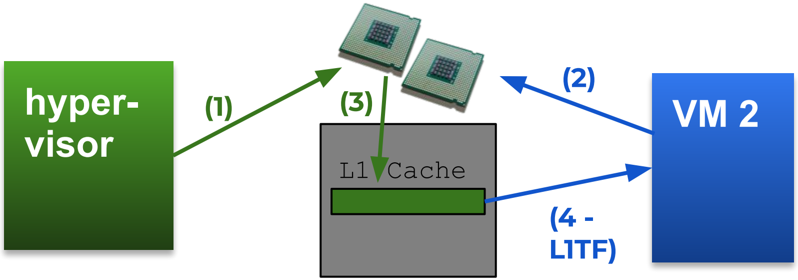


1. Hypervisor runs on CPU
2. Hypervisor puts secrets in L1
3. Hypervisor leaves CPU
4. Hypervisor: flush L1 cache
5. VM 2 runs on CPU

VMEntry

~~6. VM 2 reads hypervisor's secrets!~~

With Hyperthreading: err… mitigation?



1. Hypervisor runs on Thread A
2. VM 2 runs on Thread B
3. Hypervisor puts secrets in L1
   Hypervisor: THERE'S NOTHING I CAN DO !!!
4. **VM 2 reads Hypervisor's secret from L1 cache**

Guest kernel to Other Guest(s) attack

L1TF: Impact

- **Host User to Host Kernel (Host User to Other Host User(s), Containers):**
  - yes (but easy to mitigate, **zero** perf. cost)
- **Guest Kernel to Hypervisor (Guest to Other Guest(s)):**
  - **Xen PV:** yes (but easy to mitigate, ~= zero perf. cost)
  - **Xen HVM, PVH:** yes
  - **KVM:** yes

- **Not that hard** to exploit !

L1TF: Mitigation

- **Host, Containers:**
  - Flip address bits in page tables when `present` bit is 0
  - Resulting address will never be in L1 cache
    - Unless you have terabytes of swap space
    - ⇒ swap size limited on vulnerable CPUs (x86/speculation/l1tf: Limit swap file size to MAX_PA/2)
- **Xen PV:**
  - Xen intercepts PV guets' page table updates: sanitize/crash malicious guests
- **Xen HVM, Xen PVH, KVM:**
  - Flush L1 cache on VMEntry
  - Disable hyperthreading
  - If not wanting to disable hyperthreading... *disable hyperthreading!*
  - … … … Did I say disable hyperthreading?

L1TF: Performance Impact

- **Host, Containers, Xen PV:**
  - Negligible
- **Xen HVM, Xen PVH, KVM:**
  - L1 cache: limited (so small and so fast!)
  - Disable hyperthreading: depends
    - Varies with workloads: realistically, **-15%** in some of the common cases. Not more than **-20%**, or **-30%**, in most
    - **-50%** claimed, but only seen in specific microbenchmarks

L1TF: Performance Impact

- **Alternative ideas?** (to disabling HT)
  - Shadow Page Tables: we'd detect attacks ⇒ slow
  - Core-scheduling: only vCPUs of same VM on SMT-siblings
    - In the works, for both Xen and Linux: complex
    - ok for Guest to Other Guests, not ok for Guest to Hypervisor
  - Core-scheduling + "Coordinated VMExits": complex
  - Secret hiding:
    - Hyper-V ~done
    - Xen maybe doable
    - KVM really hard
  - Shadow Page Table, make it fast by "abusing" Intel CPU feats:
    - CR3-whitelisting, PML (was for live-migration), ...
    - ⇒ in the ~~works~~ brains...

KVM Forum '18: Alexander Graf - L1TF and KVM (has a demo!!! :-D)

Your Current Protection Status + Tunables

Your Current Situation

On a Linux host/guest. PTI, IBRS, IBPB, STIBP:

```
$ grep -E 'pti|ibrs|ibpb|stibp' -m1 /proc/cpuinfo
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca
sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin ssbd mba ibrs
ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2
smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb
intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc
cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req
flush_l1d
```

Your Current Situation

On a Linux host/guest. PTI, IBRS, IBPB, STIBP:

```
$ grep -E 'pcid' -m1 /proc/cpuinfo
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca
sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin ssbd mba ibrs
ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2
smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb
intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc
cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req
flush_l1d
```

Your Current Situation

On a Linux host/guest:

```
$ ls /sys/devices/system/cpu/vulnerabilities/
l1tf  meltdown  spec_store_bypass  spectre_v1  spectre_v2
```

```
$ grep -H . /sys/devices/system/cpu/vulnerabilities/*
/sys/devices/system/cpu/vulnerabilities/l1tf:
     Mitigation: PTE Inversion; VMX: conditional cache flushes, SMT vulnerable
/sys/devices/system/cpu/vulnerabilities/meltdown:
     Mitigation: PTI
/sys/devices/system/cpu/vulnerabilities/spec_store_bypass:
     Mitigation: Speculative Store Bypass disabled via prctl and seccomp
/sys/devices/system/cpu/vulnerabilities/spectre_v1:
     Mitigation: __user pointer sanitization
/sys/devices/system/cpu/vulnerabilities/spectre_v2:
     Mitigation: Indirect Branch Restricted Speculation, IBPB: conditional, IBRS_FW,
      STIBP: conditional, RSB filling
```

Your Current Situation TODO

On a Xen host:

```
$ ls /sys/devices/system/cpu/vulnerabilities/
l1tf  meltdown  spec_store_bypass  spectre_v1  spectre_v2

$ grep -H . /sys/devices/system/cpu/vulnerabilities/*
/sys/devices/system/cpu/vulnerabilities/l1tf:
     Mitigation: PTE Inversion; VMX: conditional cache flushes, SMT vulnerable
/sys/devices/system/cpu/vulnerabilities/meltdown:
     Mitigation: PTI
/sys/devices/system/cpu/vulnerabilities/spec_store_bypass:
     Mitigation: Speculative Store Bypass disabled via prctl and seccomp
/sys/devices/system/cpu/vulnerabilities/spectre_v1:
     Mitigation: __user pointer sanitization
/sys/devices/system/cpu/vulnerabilities/spectre_v2:
     Mitigation: Indirect Branch Restricted Speculation, IBPB: conditional, IBRS_FW,
      STIBP: conditional, RSB filling
```

Tunables

~~<<Greetings, how **slow** do you want to go today?>>~~
<<Greetings, how **secure** do you want to be today?>>

- **KVM:**
  - pti = on|off| auto
  - spectre_v2 = on|off|auto|retpoline,generic| retpoline,amd
  - spec_store_bypass_disable = on|off|auto|prctl|seccomp
  - l1tf = full|flush|flush,nosmt
  - kvm-intel.vmentry_l1d_flush = always|cond|never

- **XEN:**
  - xpti = [dom0 = TRUE/FALSE , domu = TRUE/FALSE]
  - bti-thunk = retpoline|lfence|jmp
  - {ibrs,ibpb,ssbd,eager-fpu,l1d-flush} = TRUE/FALSE
  - {smt,pv-l1tf} = TRUE/FALSE

Conclusions

- "Hardware bugs" are **difficult**
  - Not only to ~~fix~~ mitigate
  - But also to work on, collaboratively (NDAs, etc)
  - Getting better
- Issues like these will **really** hunt us for a few time…
- Speculative Execution has **shaped** Computing World
- We focused on **performance first**, now we deal with consequences. As grandma used to say: <<*L'hai voluta la bicicletta, oh pedala!!!*>>
- Do **update** your firmware/microcode; do **update** your kernel
- Threats are real **but** don't panic: analyze your system, assess risks
- Performance impact may be really high but don't panic: **benchmark** your own workload, look for tunables

Some Examples / Anecdotes / Curiosities

NoTimers, NoFlush? Still party!

Cache as a side channel:

- Some control cache content (flush, place own array)
- Accurately time array elements accesses

So… Is forbidding user-space code to flush cache a mitigation?

- No! User code can still cause cache flushes, via memory allocation
- No! User code can “displace” array elements

So… Is reducing timers’ resolution for user-space code a mitigation?

- No! If I have shared memory (& multi-core/multi-thread) I can setup a counter thread == a timer
- (Actually done, e.g., in Android and in some browsers…)

“Microcode” What ?

Hardware bugs (yes, we’ve had those before!)

- Cyrix Coma, Pentium FDIV or Pentium F00F)
- ⇒ Hardware replacement!

We don’t want that:

- CPUs executes “micro-operations” (µops), not real x86 opcodes
- Translation between opcodes and µops: microcode, inside CPUs
- Can be changed/updated (distributed only in binary form)
- Change CPU behavior “in the field”
- Well, up to a certain extent!
- (NB updates are not persistent, reload at boot)

Chicken bits

“Chicken bits”

- A control bit stored in a register, used in ASICs and other integrated circuits to disable or enable features within a chip. https://www.urbandictionary.com/define.php?term=Chicken%20Bit
- (electronics) A bit on a chip that can be used to disable one of the features of the chip if it proves faulty or negatively impacts performance. https://en.wiktionary.org/wiki/chicken_bit

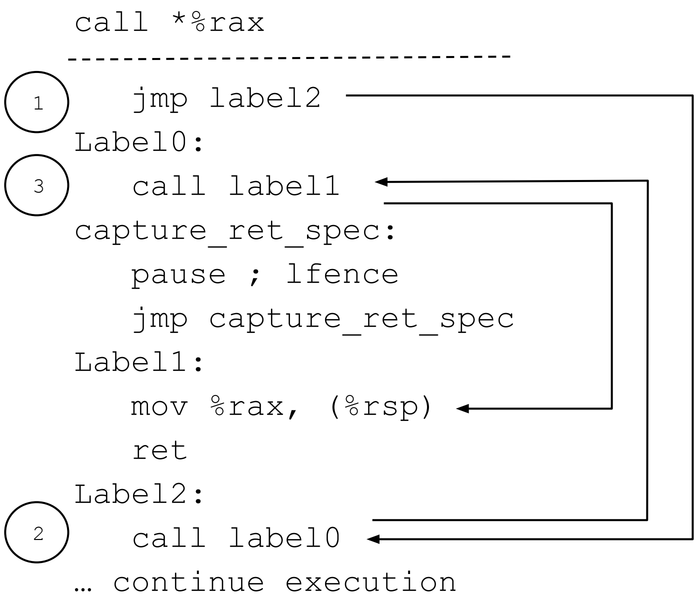
2010, Ilya Wagner & Valeria Bertacco, *Post-Silicon and Runtime Verification for Modern Processors*, Springer, page 165: <<As an example, modules such as branch predictors and speculative execution units can be turned off with a variant of the “chicken bits”, control bits common to many design developments to control the activation of specific features.>>

Retpoline for `call`

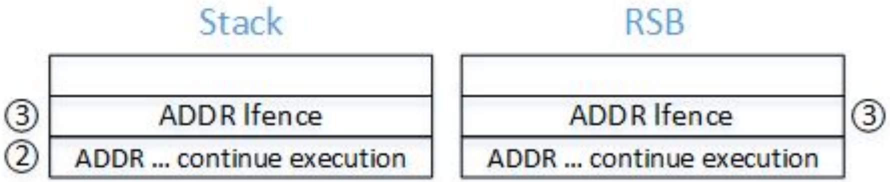
```
call *%rax
------------------
   jmp label2
Label0:
   call label1
capture_ret_spec:
   pause ; lfence
   jmp capture_ret_spec
Label1:
   mov %rax, (%rsp)
   ret
Label2:
   call label0
… continue execution
```

https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation

Retpoline for `call`



```
call *%rax
------------------------------
    jmp label2
Label0:
    call label1
capture_ret_spec:
    pause ; lfence
    jmp capture_ret_spec
Label1:
    mov %rax, (%rsp)
    ret
Label2:
    call label0
… continue execution
```

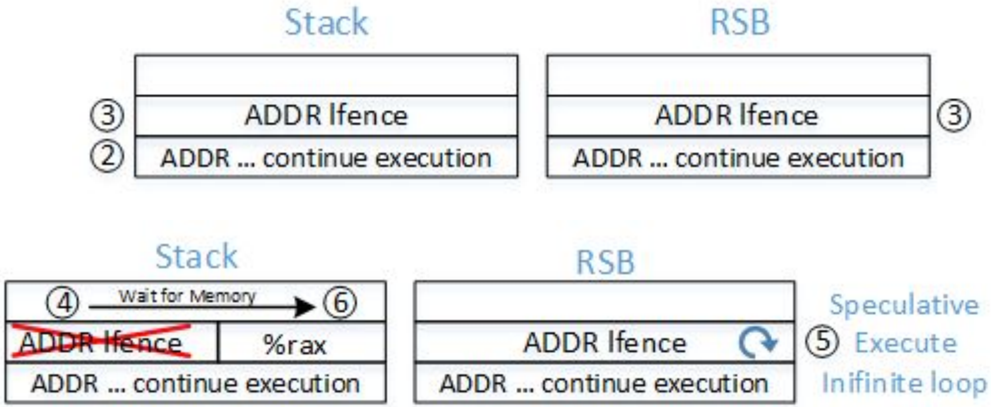


	Stack	RSB	
3	ADDR lfence	ADDR lfence	3
2	ADDR ... continue execution	ADDR ... continue execution	

https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation

Retpoline for `call`

```
     call *%rax
    ------------------------------
1        jmp label2
     Label0:
3        call label1
     capture_ret_spec:
         pause ; lfence
         jmp capture_ret_spec
     Label1:
4        mov %rax, (%rsp)   5
         ret
     Label2:
2        call label0
     … continue execution
```

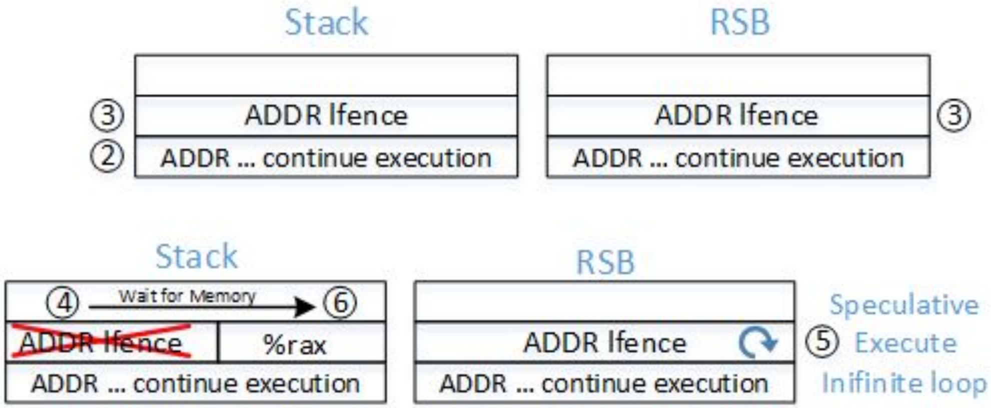


(5) Speculation (while waiting for the mov to memory). Where? At the “trap”

https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation

Retpoline for `call`

```
    call *%rax
    -------------------------------
1      jmp label2
    Label0:
3      call label1
    capture_ret_spec:
       pause ; lfence
       jmp capture_ret_spec
    Label1:
4      mov %rax, (%rsp)   5
6      ret
    Label2:
2      call label0
    … continue execution
```

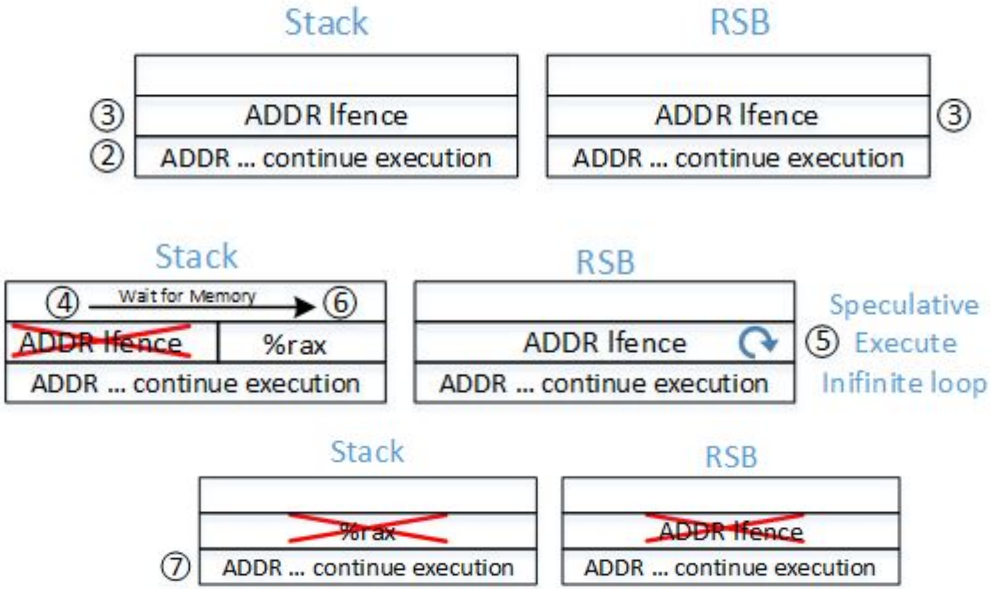


ret → To *(%rax), as that's what was at top of stack

https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation

Retpoline for `call`

```
   call *%rax
  ------------------------------
1     jmp label2
   Label0:
3     call label1
   capture_ret_spec:
      pause ; lfence
      jmp capture_ret_spec
   Label1:
4     mov %rax, (%rsp)  5
6     ret
   Label2:
2     call label0
7  … continue execution
```



Stack

RSB

③ ADDR lfence

② ADDR ... continue execution

ADDR lfence ③

ADDR ... continue execution

Stack

④ Wait for Memory ⑥

~~ADDR lfence~~ | %rax

ADDR ... continue execution

RSB

ADDR lfence

ADDR ... continue execution

Speculative
⑤ Execute
Inifinite loop

Stack

~~%rax~~

⑦ ADDR ... continue execution

RSB

~~ADDR lfence~~

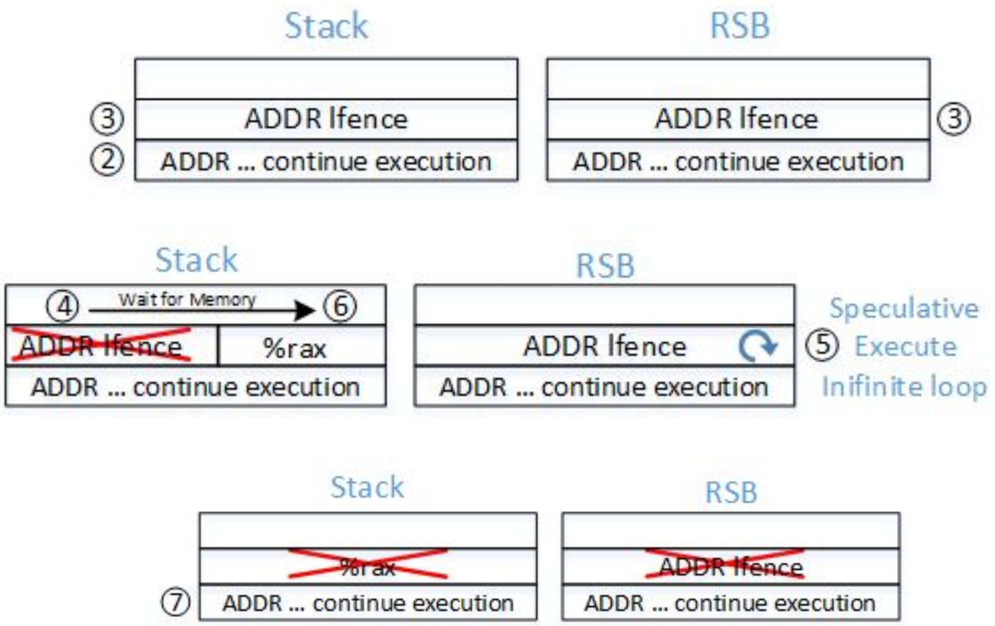
ADDR ... continue execution

Function at *(%rax) returns here

https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation

Retpoline for `call`

```
 call *%rax
------------------------------
1    jmp label2
 Label0:
3    call label1
 capture_ret_spec:
     pause ; lfence
     jmp capture_ret_spec
 Label1:
     mov %rax, (%rsp)
     ret
 Label2:
2    call label0
 … continue execution
```



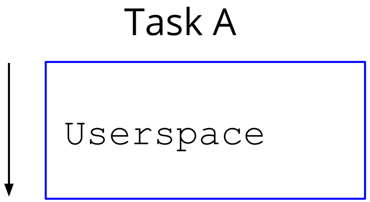
Stack
RSB
③
②
ADDR lfence
ADDR ... continue execution
ADDR lfence
ADDR ... continue execution
③
Stack
④
Wait for Memory
⑥
~~ADDR lfence~~
%rax
ADDR ... continue execution
RSB
ADDR lfence
ADDR ... continue execution
Speculative
⑤ Execute
Inifinite loop
Stack
RSB
~~%rax~~
⑦
ADDR ... continue execution
~~ADDR lfence~~
ADDR ... continue execution

https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation

IBRS_FW

Talking about Spectre-v2, IBRS vs. retpoline

```
/sys/devices/system/cpu/vulnerabilities/spectre_v2:
     Mitigation: full generic retpoline, IBPB: conditional, IBRS_FW,
      STIBP: conditional, RSB filling
```



Task A

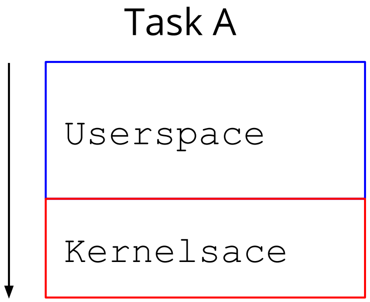
Userspace

Compiled with retpoline enabled compiler: **safe**

IBRS_FW

Talking about Spectre-v2, IBRS vs. retpoline

```
/sys/devices/system/cpu/vulnerabilities/spectre_v2:
     Mitigation: full generic retpoline, IBPB: conditional, IBRS_FW,
      STIBP: conditional, RSB filling
```



Task A

Userspace

Kernelsace

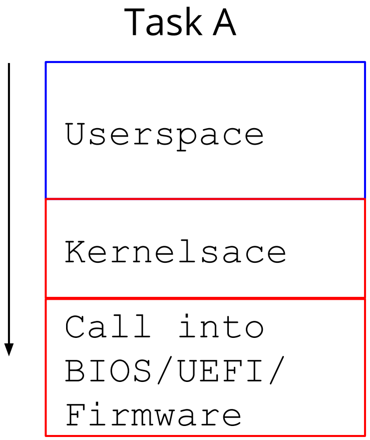
Compiled with `retpoline` enabled compiler: **safe**

`retpoline` enabled as in-kernel mitigation: **safe**

IBRS_FW

Talking about Spectre-v2, IBRS vs. retpoline

```
/sys/devices/system/cpu/vulnerabilities/spectre_v2:
     Mitigation: full generic retpoline, IBPB: conditional, IBRS_FW,
      STIBP: conditional, RSB filling
```



Task A

Userspace

Kernelsace

Call into BIOS/UEFI/ Firmware

Compiled with `retpoline` enabled compiler: safe

`retpoline` enabled as in-kernel mitigation: safe

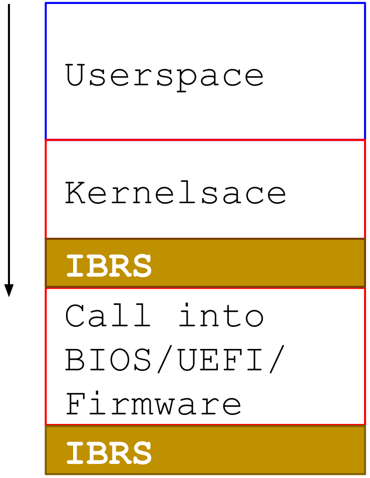
- Is firmware using `IBRS`?
- Is firmware compiler with `retpoline`?
  We can’t know: unsafe!

IBRS_FW

Talking about Spectre-v2, IBRS vs. retpoline

```
/sys/devices/system/cpu/vulnerabilities/spectre_v2:
     Mitigation: full generic retpoline, IBPB: conditional, IBRS_FW,
      STIBP: conditional, RSB filling
```

Task A



Userspace

Kernelsace

IBRS

Call into BIOS/UEFI/ Firmware

IBRS

Compiled with `retpoline` enabled compiler: **safe**

`retpoline` enabled as in-kernel mitigation: **safe**

- Is firmware using `IBRS`?
- Is firmware compiler with `retpoline`?
  We can’t know: unsafe!

Wrap firmware calls/services around `IBRS`

http://www.eventhelix.com/realtimemantra/Basics/CToAssemblyTranslation3.htm
https://en.wikipedia.org/wiki/Branch_table

Compiling `switch() {...}`

```
int global;

int foo3 (int x)
{
  switch (x) {
    case 0:
      return 11;
    case 1:
      return 123;
    case 2:
      global += 1;
      return 3;
    case 3:
      return 44;
    case 4:
      return 444;
    default:
      return 0;
  }
}
```

```
gcc jt.c -O2  -S -o/dev/stdout
      .file "jt.c"
      .text
      .p2align 4,,15
      .globlfoo3
      .type foo3, @function
foo3:
.LFB0:
      .cfi_startproc
      cmpl  $4, %edi
      ja    .L2
      movl  %edi, %edi
      jmp   *.L4(,%rdi,8)
      .section    .rodata
      .align 8
      .align 4
```

```
.L4:
      .quad .L9
      .quad .L7
      .quad .L6
      .quad .L5
      .quad .L3
      .text
      .p2align 4,,10
      .p2align 3
.L9:
      movl  $11, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L3:
      movl  $444, %eax
      ret
      .p2align 4,,10
      .p2align 3
```

```
.L6:
      addl  $1, global(%rip)
      movl  $3, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L5:
      movl  $44, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L7:
      movl  $123, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L2:
      xorl  %eax, %eax
      ret
      .cfi_endproc
```

http://www.eventhelix.com/realtimemantra/Basics/CToAssemblyTranslation3.htm
https://en.wikipedia.org/wiki/Branch_table

Compiling `switch() {...}`

```
int global;

int foo3 (int x)
{
  switch (x) {
    case 0:
      return 11;
    case 1:
      return 123;
    case 2:
      global += 1;
      return 3;
    case 3:
      return 44;
    case 4:
      return 444;
    default:
      return 0;
  }
}
```

```
gcc jt.c -O2  -S -o/dev/stdout
      .file "jt.c"
      .text
      .p2align 4,,15
      .globlfoo3
      .type foo3, @function
foo3:
.LFB0:
      .cfi_startproc
      cmpl  $4, %edi
      ja    .L2
      movl  %edi, %edi
      jmp   *.L4(,%rdi,8)
      .section    .rodata
      .align 8
      .align 4
```

Jump Table

```
.L4:
      .quad .L9
      .quad .L7
      .quad .L6
      .quad .L5
      .quad .L3
      .text
      .p2align 4,,10
      .p2align 3
.L9:
      movl  $11, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L3:
      movl  $444, %eax
      ret
      .p2align 4,,10
      .p2align 3
```

```
.L6:
      addl  $1, global(%rip)
      movl  $3, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L5:
      movl  $44, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L7:
      movl  $123, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L2:
      xorl  %eax, %eax
      ret
      .cfi_endproc
```

http://www.eventhelix.com/realtimemantra/Basics/CToAssemblyTranslation3.htm
https://en.wikipedia.org/wiki/Branch_table

Compiling `switch() {...}`

```
int global;

int foo3 (int x)
{
  switch (x) {
    case 0:
      return 11;
    case 1:
      return 123;
    case 2:
      global += 1;
      return 3;
    case 3:
      return 44;
    case 4:
      return 444;
    default:
      return 0;
  }
}
```

```
gcc jt.c -O2  -S -o/dev/stdout
      .file "jt.c"
      .text
      .p2align 4,,15
      .globlfoo3
      .type foo3, @function
foo3:
.LFB0:
      .cfi_startproc
      cmpl  $4, %edi
      ja    .L2
      movl  %edi, %edi
      jmp   *.L4(,%rdi,8)
      .section    .rodata
      .align 8
      .align 4
```

Jump Table

```
.L4:
      .quad .L9
      .quad .L7
      .quad .L6
      .quad .L5
      .quad .L3
      .text
      .p2align 4,,10
      .p2align 3
.L9:
      movl  $11, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L3:
      movl  $444, %eax

        gn 4,,10
        gn 3
```

```
.L6:
      addl  $1, global(%rip)
      movl  $3, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L5:
      movl  $44, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L7:
      movl  $123, %eax
      ret
      .p2align 4,,10
      .p2align 3
.L2:
      xorl  %eax, %eax
      ret
      .cfi_endproc
```

- Faster (alternative: ~if/else)
- Better fits in cache
- Perf. independent than nr. cases

Compiling `switch() {...}`

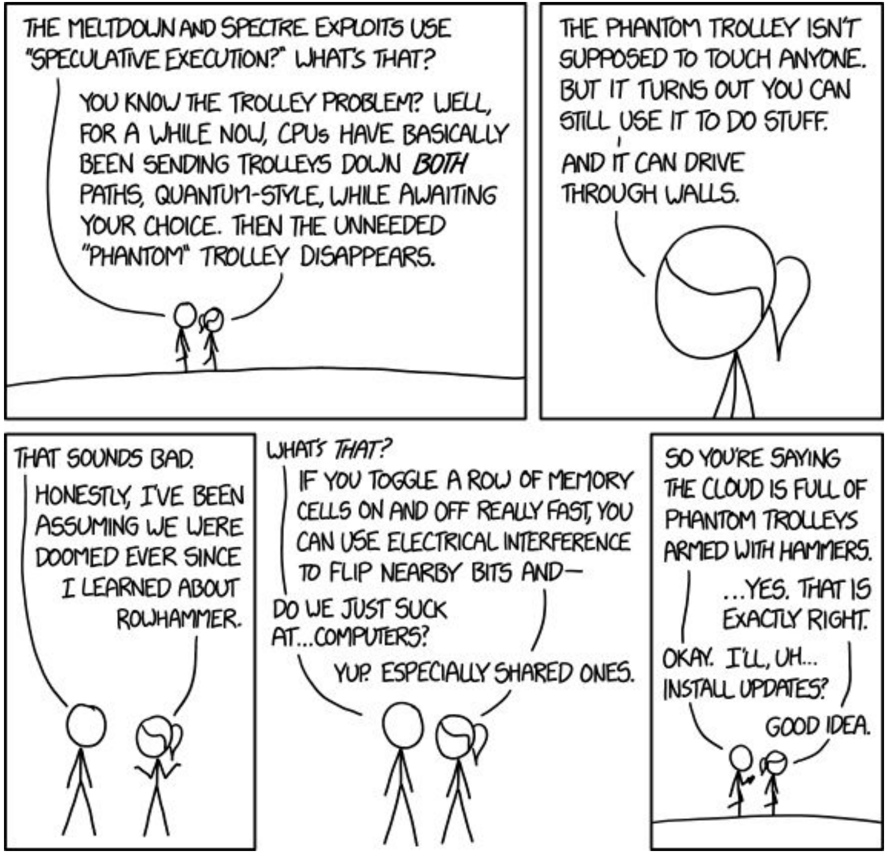
```
            normal       retpoline    retpo+no-JT  retpo+JT=20  retpo+JT=40
cases:    8: 0.70  (100%)  2.98  (425%)  0.75  (107%)  0.75  (107%)  0.75  (107%)
cases:   16: 0.70  (100%)  2.98  (425%)  0.82  (117%)  0.82  (117%)  0.82  (117%)
cases:   32: 0.70  (100%)  3.01  (430%)  0.87  (124%)  2.98  (426%)  0.87  (124%)
cases:   64: 0.70  (100%)  3.52  (501%)  0.94  (134%)  3.52  (501%)  3.52  (501%)
cases:  128: 0.71  (100%)  3.51  (495%)  1.07  (151%)  3.50  (495%)  3.50  (494%)
cases:  256: 0.76  (100%)  3.14  (414%)  1.27  (167%)  3.14  (414%)  3.14  (414%)
cases: 1024: 1.46  (100%)  3.36  (230%)  1.49  (102%)  3.36  (230%)  3.36  (230%)
cases: 2048: 2.25  (100%)  3.19  (142%)  2.70  (120%)  3.19  (142%)  3.19  (142%)
cases: 4096: 2.90  (100%)  3.74  (129%)  4.48  (155%)  3.73  (129%)  3.72  (129%)
```

“I'm going to prepare a patch that will disable JTs for retpolines.”

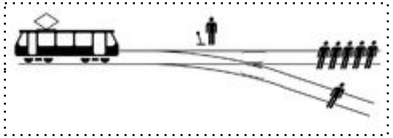
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86952 (https://github.com/marxin/microbenchmark-1)

Thanks Everyone!

Questions?


THE MELTDOWN AND SPECTRE EXPLOITS USE "SPECULATIVE EXECUTION?" WHAT'S THAT?
YOU KNOW THE TROLLEY PROBLEM? WELL, FOR A WHILE NOW, CPUs HAVE BASICALLY BEEN SENDING TROLLEYS DOWN BOTH PATHS, QUANTUM-STYLE, WHILE AWAITING YOUR CHOICE. THEN THE UNNEEDED "PHANTOM" TROLLEY DISAPPEARS.
THE PHANTOM TROLLEY ISN'T SUPPOSED TO TOUCH ANYONE. BUT IT TURNS OUT YOU CAN STILL USE IT TO DO STUFF.
AND IT CAN DRIVE THROUGH WALLS.
THAT SOUNDS BAD.
HONESTLY, I'VE BEEN ASSUMING WE WERE DOOMED EVER SINCE I LEARNED ABOUT ROWHAMMER.
WHAT'S THAT?
IF YOU TOGGLE A ROW OF MEMORY CELLS ON AND OFF REALLY FAST, YOU CAN USE ELECTRICAL INTERFERENCE TO FLIP NEARBY BITS AND—
DO WE JUST SUCK AT...COMPUTERS?
YUP. ESPECIALLY SHARED ONES.
SO YOU'RE SAYING THE CLOUD IS FULL OF PHANTOM TROLLEYS ARMED WITH HAMMERS.
...YES. THAT IS EXACTLY RIGHT.
OKAY. I'LL, UH... INSTALL UPDATES?
GOOD IDEA.

https://xkcd.com/1938/



Trolley problem