

# Comp 790-184: Hardware Security and Side-Channels

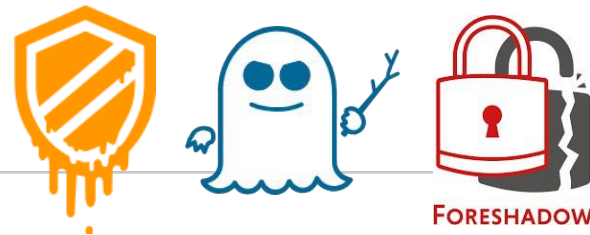
## Lecture 3: Transient Execution Attacks

January 30, 2025  
Andrew Kwong



THE UNIVERSITY  
*of* NORTH CAROLINA  
*at* CHAPEL HILL

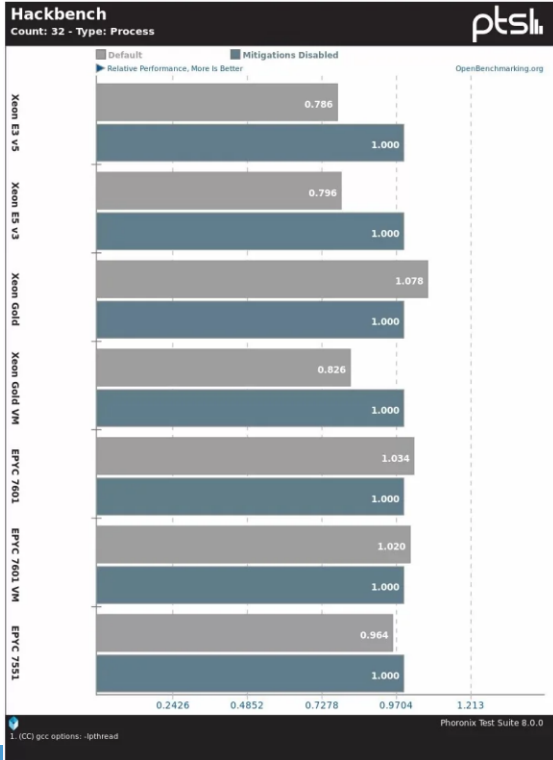
# Outline



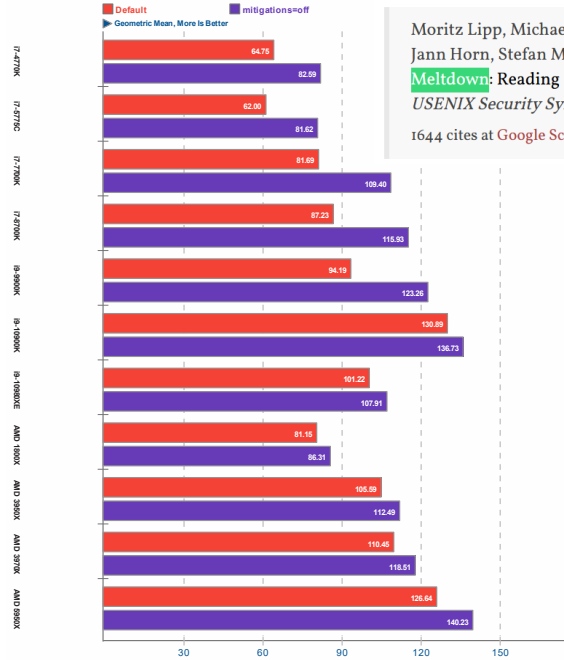
- What are transient execution attacks?
- How does Meltdown work?
  - We will connect the dots between a hardware optimization and a software optimization.
- How do Spectre and its variations work?
  - Let's try to see through these variations and understand the fundamental problem.

Slides adapted from Mengjia Yan  
([shd.mit.edu](http://shd.mit.edu))

# Impact



**Geometric Mean Of All Test Results**  
Result Composite



Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom:

**Spectre Attacks: Exploiting Speculative Execution.**  
*IEEE Symposium on Security and Privacy (S&P), 2019*

2731 cites at [Google Scholar](#) | 3286% above average of year | Last visited: Jan-2024 | Paper: DOI

6

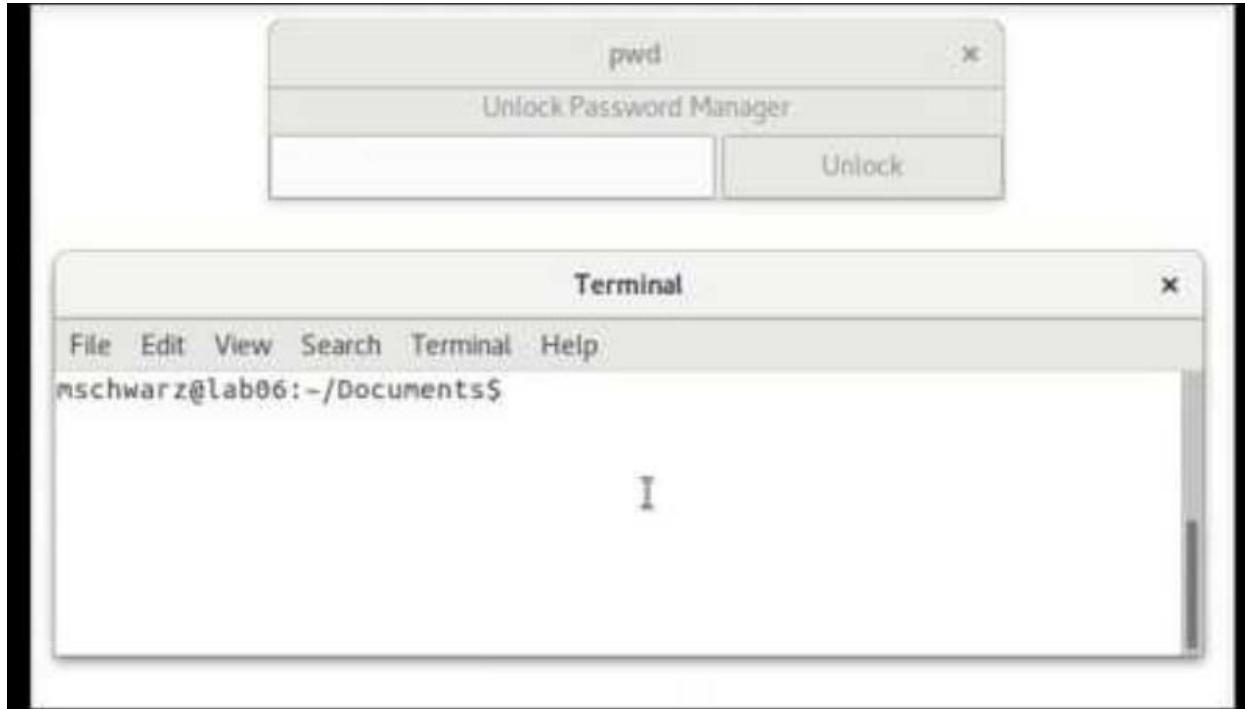
Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg:  
**Meltdown: Reading Kernel Memory from User Space.**  
*USENIX Security Symposium, 2018*

1644 cites at [Google Scholar](#) | 1415% above average of year | Last visited: Jan-2024 | Paper: DOI

55

# Meltdown

---



# Meltdown

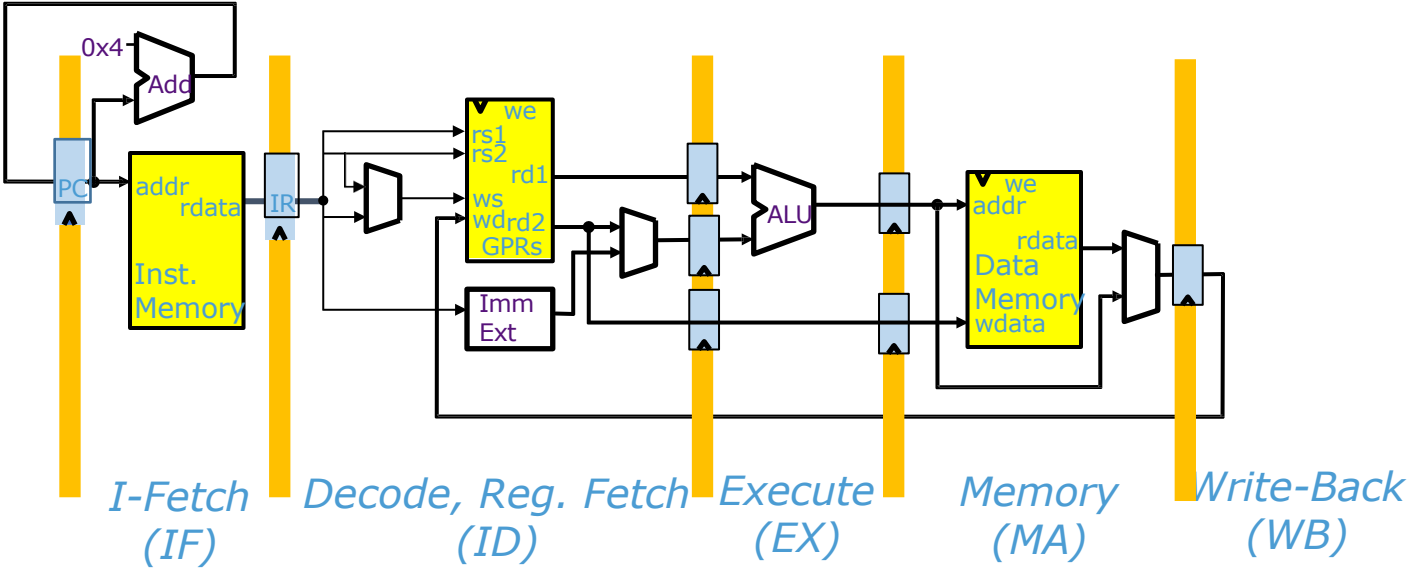


## Meltdown Root Causes

---

- Due to the combination of both a hardware and software optimization
  - Out of order execution
  - Mapping kernel memory into user space

# Recap: 5-stage Pipeline



## Recap: 5-stage Pipeline

---

- In-order execution:
  - Execute instructions according to the program order
  - What is the ideal instruction throughput? -- instruction per cycle (IPC)

<i>time</i>	t0	t1	t2	t3	t4	t5	t6	t7	...
instruction1	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	WB <sub>1</sub>				
instruction2		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	MA <sub>2</sub>	WB <sub>2</sub>			
instruction3			IF <sub>3</sub>	ID <sub>3</sub>	EX <sub>3</sub>	MA <sub>3</sub>	WB <sub>3</sub>		
instruction4				IF <sub>4</sub>	ID <sub>4</sub>	EX <sub>4</sub>	MA <sub>4</sub>	WB <sub>4</sub>	
instruction5					IF <sub>5</sub>	ID <sub>5</sub>	EX <sub>5</sub>	MA <sub>5</sub>	WB <sub>5</sub>



# Build High-Performance Processors

---

Example #1:

```
FMUL f1, f2, f3 ; 10 cycles  
ADD r4, r4, r1 ; 1 cycle -> repeat 7 times  
.....
```

Example #2:

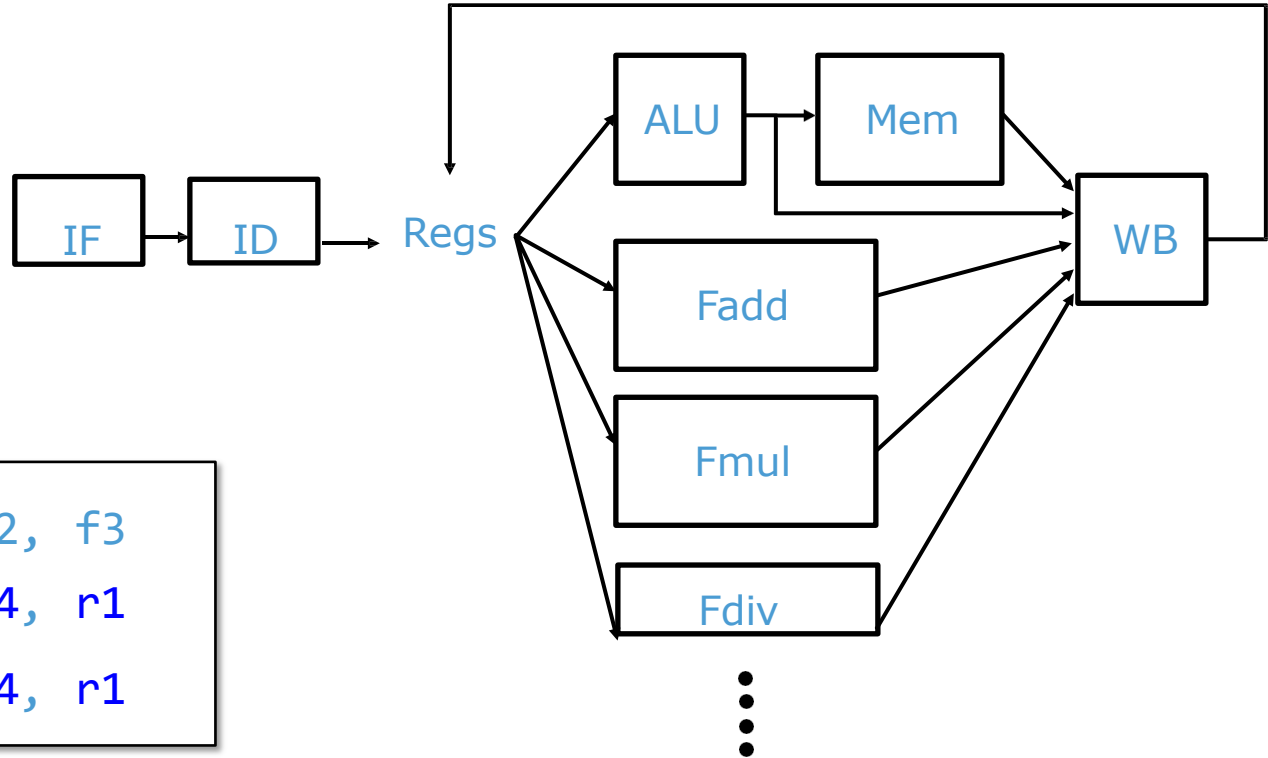
```
LD r3, 0(r2) ; 1-100 cycles  
ADD r4, r4, r1 ; 1 cycle -> repeat 10 times  
.....
```



Instruction-Level  
Parallelism (ILP)

when there is no data-dependency or  
control-flow dependency between  
instructions

## Technique #1: Add More Functional Units

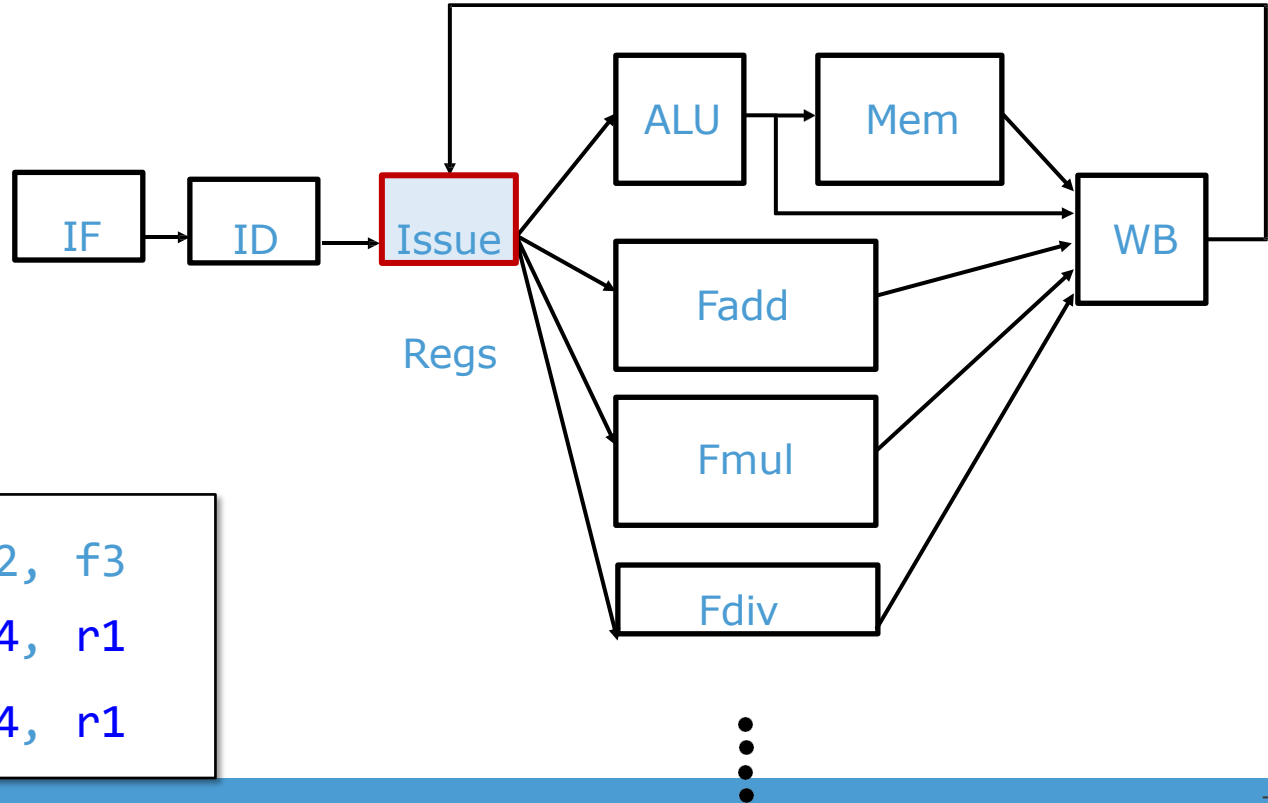


1: FMUL f1, f2, f3

2: ADD r4, r4, r1

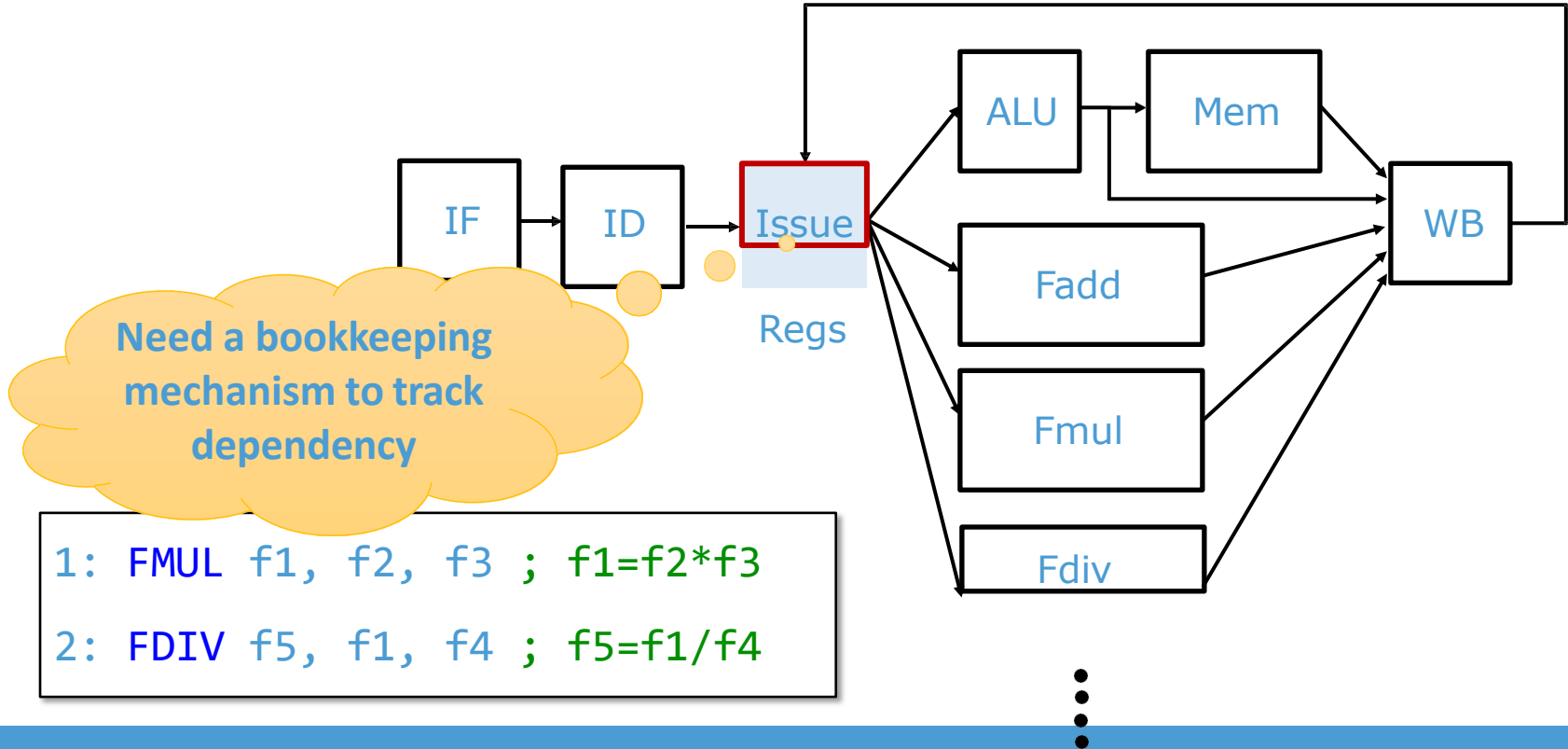
3: ADD r4, r4, r1

## Technique #1: Add More Functional Units



```
1: FMUL f1, f2, f3
2: ADD  r4, r4, r1
3: ADD  r4, r4, r1
```

# Technique #1: Add More Functional Units



## Technique #2: Scoreboard

---

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul				
Fdiv				

```
1: FMUL f1, f2, f3
```

```
2: ADD r4, r4, r1
```

## Technique #2: Scoreboard

---

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul	Y	f1	f2	f3
Fdiv				

1: FMUL f1, f2, f3

2: ADD r4, r4, r1

## Technique #2: Scoreboard

---

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd	Y	r4	r4	r1
Fmul	Y	f1	f2	f3
Fdiv				

1: FMUL f1, f2, f3

2: ADD r4, r4, r1

## Technique #2: Scoreboard

---

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul				
Fdiv				

1: FMUL f1, f2, f3

2: FDIV f5, f1, f4



## Technique #2: Scoreboard

---

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul	Y	f1	f2	f3
Fdiv				

1: FMUL f1, f2, f3

2: FDIV f5, f1, f4

## Technique #2: Scoreboard

---

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul	Y	f1	f2	f3
Fdiv	Y	f5	f1	f4

Data  
Hazard!

1: FMUL f1, f2, f3

2: FDIV f5, f1, f4

## Technique #2: Scoreboard

---

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul				
Fdiv				

1: FMUL f1, f2, f3 ;10 cycles

2: FADD f1, f4, f5 ;4 cycles

## Technique #2: Scoreboard

---

- Upon issue of an instruction, check:
  1. Whether any ongoing instructions will generate values for my source registers
  2. Whether any ongoing instructions will modify my destination register
- We call such a processor: **in-order issue, out-of-order completion.**
- A problem: how to handle interrupts/exceptions?

## Exception in OoO Processors: Example #1

```
1: LD  r3, 0(r2)    ; Exception in 3 cycles
2: ADD r4, r4, r1    ; 1 cycle
```

Need to delay WB

	1	2	3	4	5	6	7	8
1: LD	IF	ID	Issue	ALU	Mem	Mem.	Mem	Exception
2: ADD		IF	ID	Issue	ALU	WB		

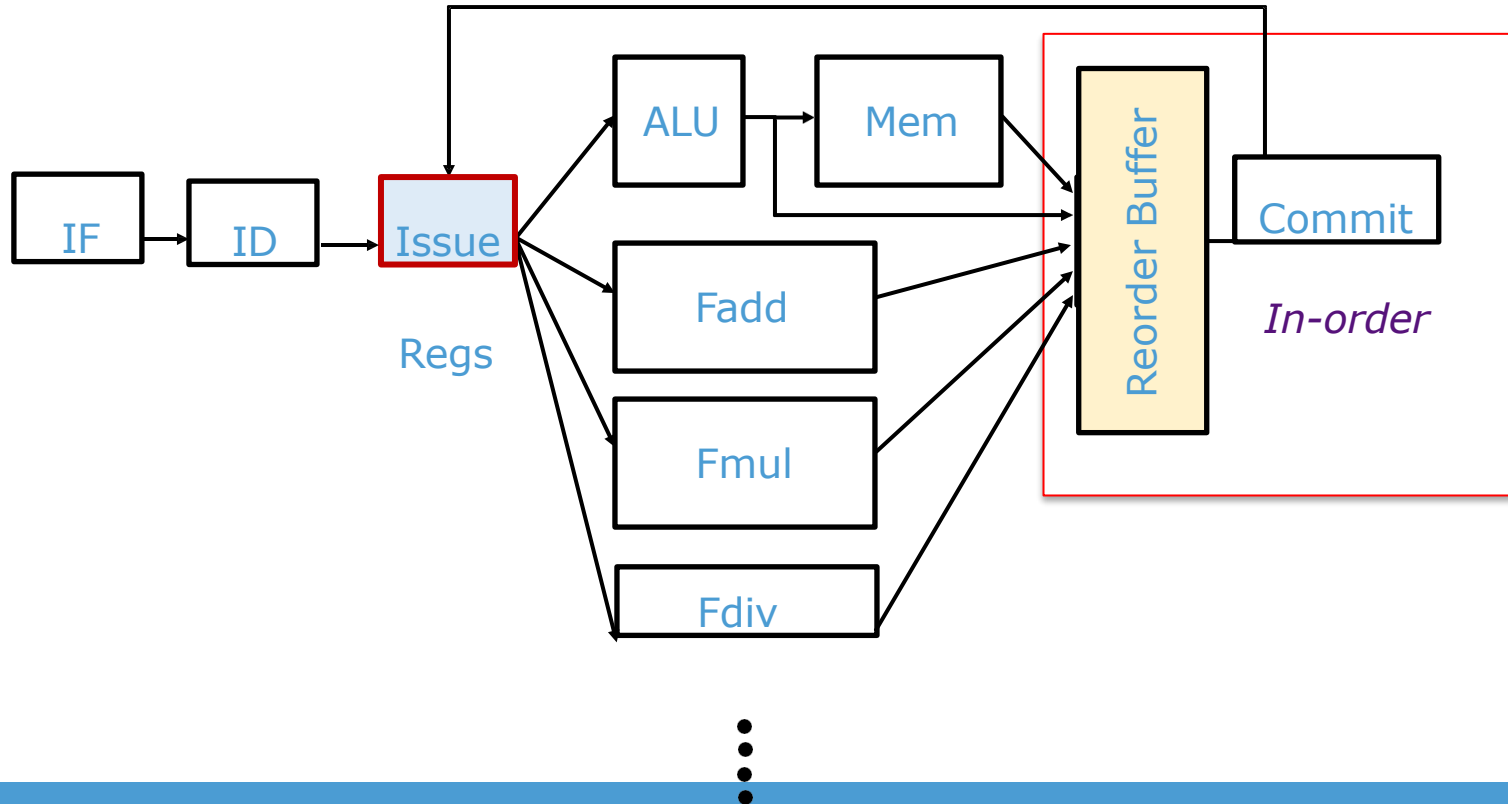
## Exception in OoO Processors: Example #2

```
1: FMUL f1, f2, f3 ; 10 cycles
2: LD r3, 0(r2) ; Exception in 1 cycle
```

Need to delay  
Exception

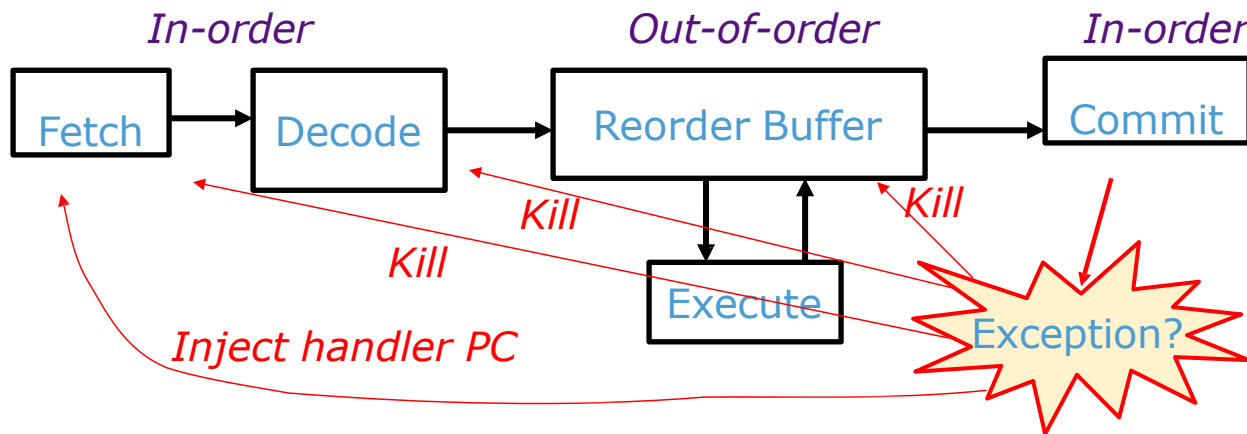
	1	2	3	4	5	6	7	8
1: FMUL	IF	ID	Issue	FMUL	FMUL	FMUL	FMUL	...
2: LD		IF	ID	Issue	ALU	Mem	Exception	

## Technique #3: In-order Commit



## Another Way to Draw It

---





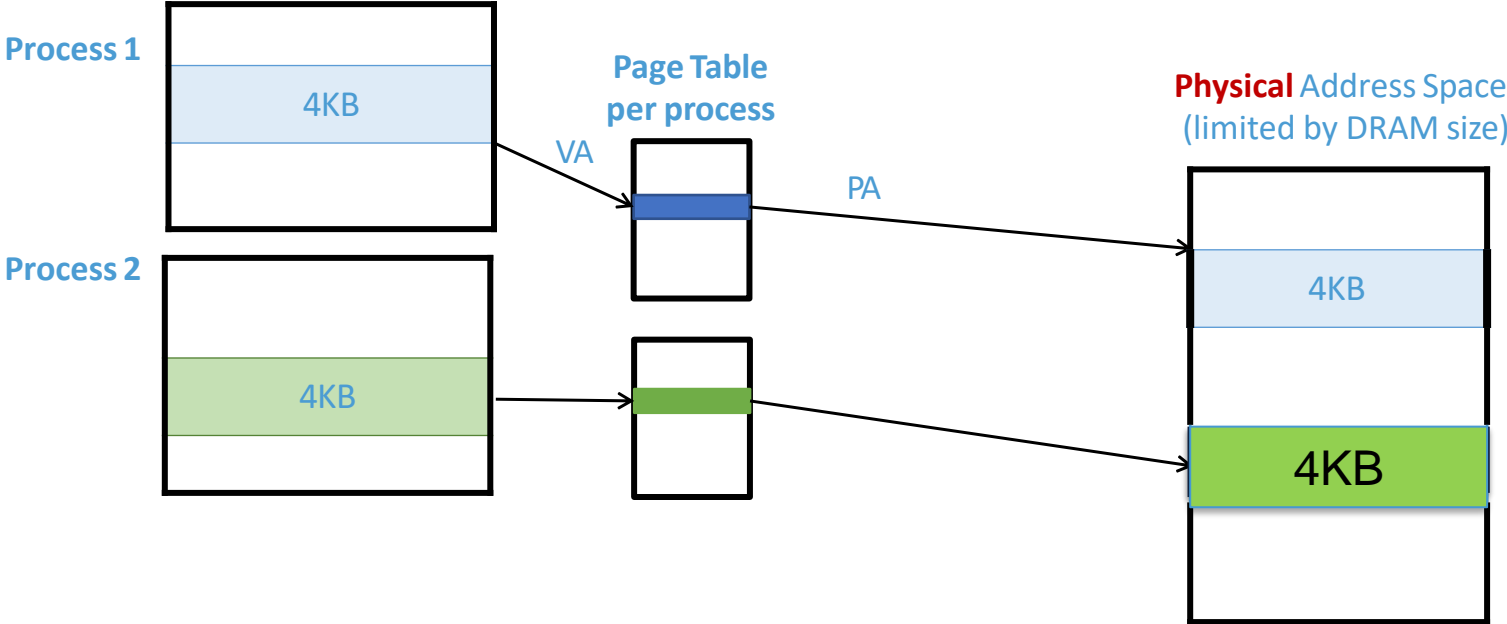
## Re-examine Examples With In-order Commit

---

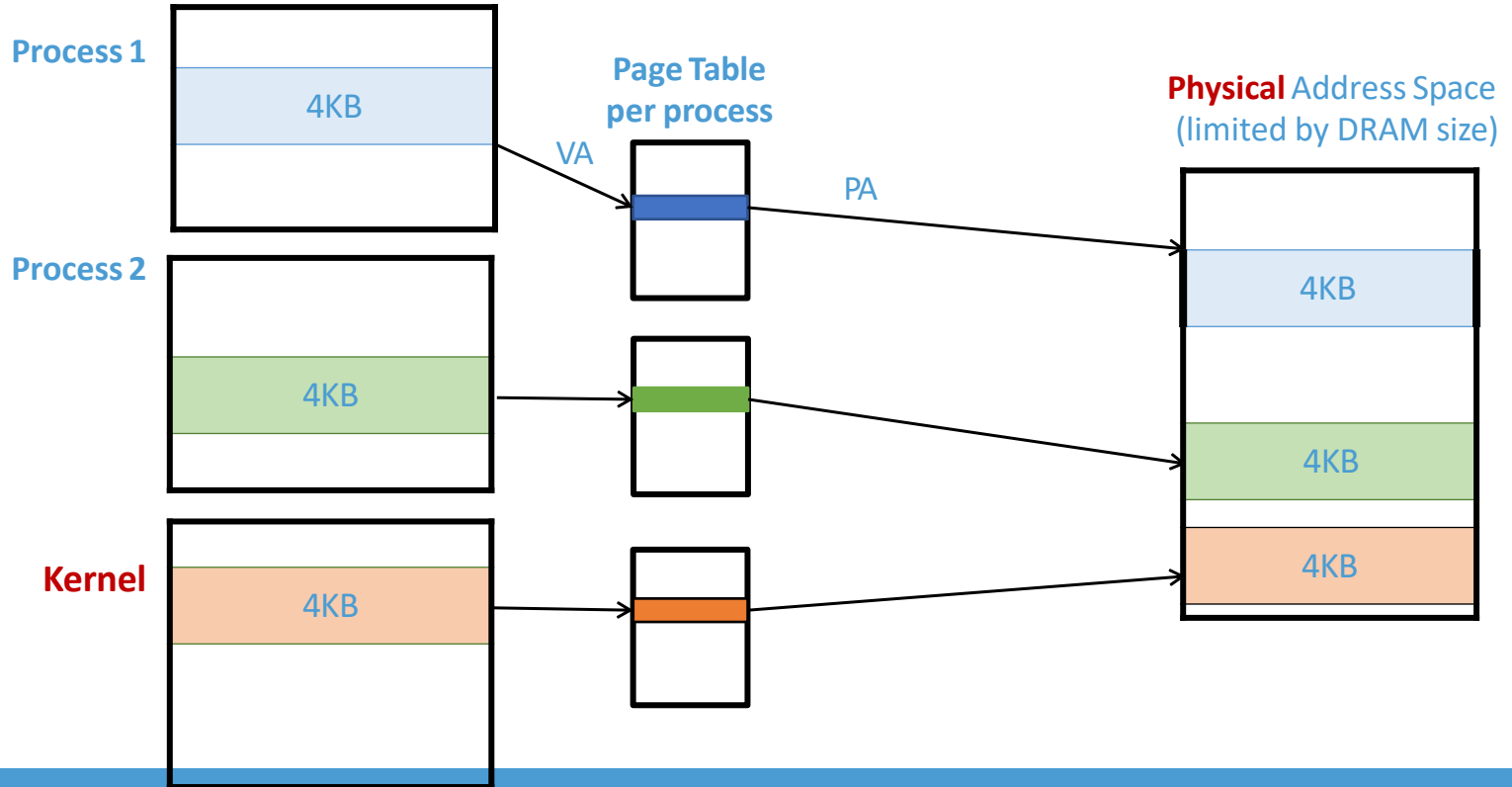
```
1: LD  r3, 0(r2)    ; Exception in 3 cycles  
2: ADD r4, r4, r1   ; 1 cycle
```

```
1: FMUL f1, f2, f3  ; 10 cycles  
2: LD  r3, 0(r2)    ; Exception in 1 cycle
```

# Recap: Page Mapping



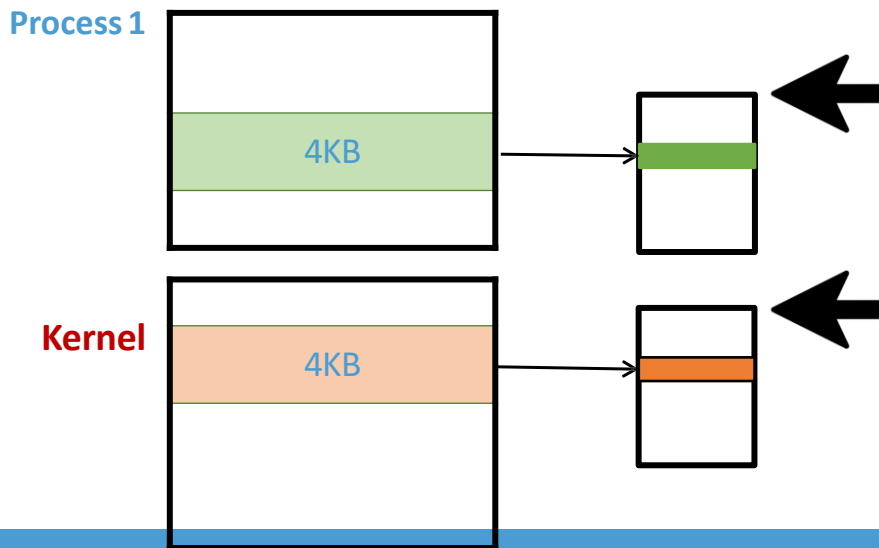
# Mapping Kernel Pages



# Jumping Between User and Kernel Space

---

- Key challenge: need to make sure we use the correct page table
  - CR3 (in x86) or satp (in RISC-V) stores the page table physical address

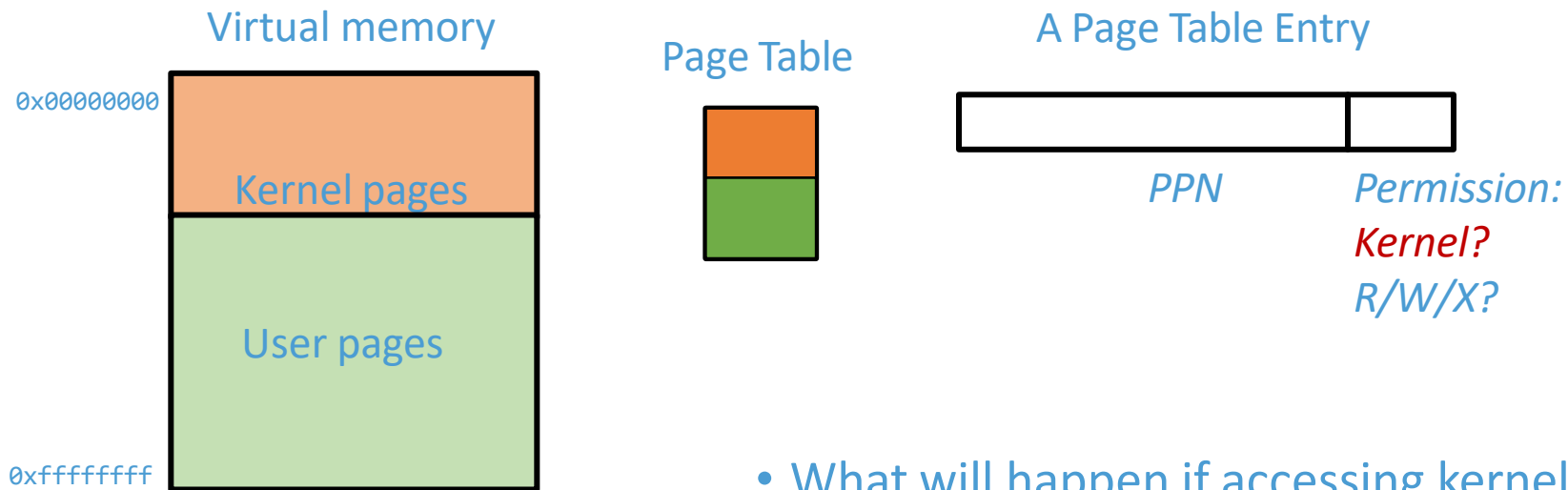


# A Performance Optimization

---

- Context switch overhead:
  - Page table changes, so in many processors, we need to flush TLB
- But sometimes, we only go to kernel to do some simple things
  - E.g., `getpid()`
- The optimization: map kernel address into user space in a **secure** way

# Map Kernel Pages Into User Space



- What will happen if accessing kernel addresses in user mode?
- Protection fault

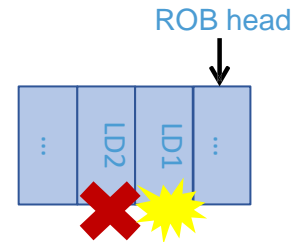
# Meltdown

---

- Put two optimizations together, we have Meltdown
  - Hardware optimization: out-of-order execution
  - Software optimization: mapping kernel addresses into user space
- Attack outcome: user space applications can read arbitrary kernel data

```
.....  
Ld1: uint8_t secret = *kernel_address;  
Ld2: uint8_t dummy = probe_array[secret*64];
```

2<sup>nd</sup> line of code can transiently execute before the exception occurs!



## Meltdown w/ Flush+Reload

---

1. Setup: Attacker allocates `probe_array`, with 256 cache lines. Flushes all its cache lines
2. Transmit: Attacker executes

```
.....  
Ld1: uint8_t secret = *kernel_address;  
Ld2: uint8_t dummy = probe_array[secret*64];
```

3. Receive: After handling protection fault, attacker performs cache side channel attack to figure out which line of `probe_array` is accessed → recovers **byte**



# Meltdown Mitigations

---

- Stop one of the optimizations should be sufficient
  - SW: Do not let user and kernel share address space (KPTI) -> broken by several groups (e.g., *EntryBleed*)
  - HW: Stall speculation; Register poisoning

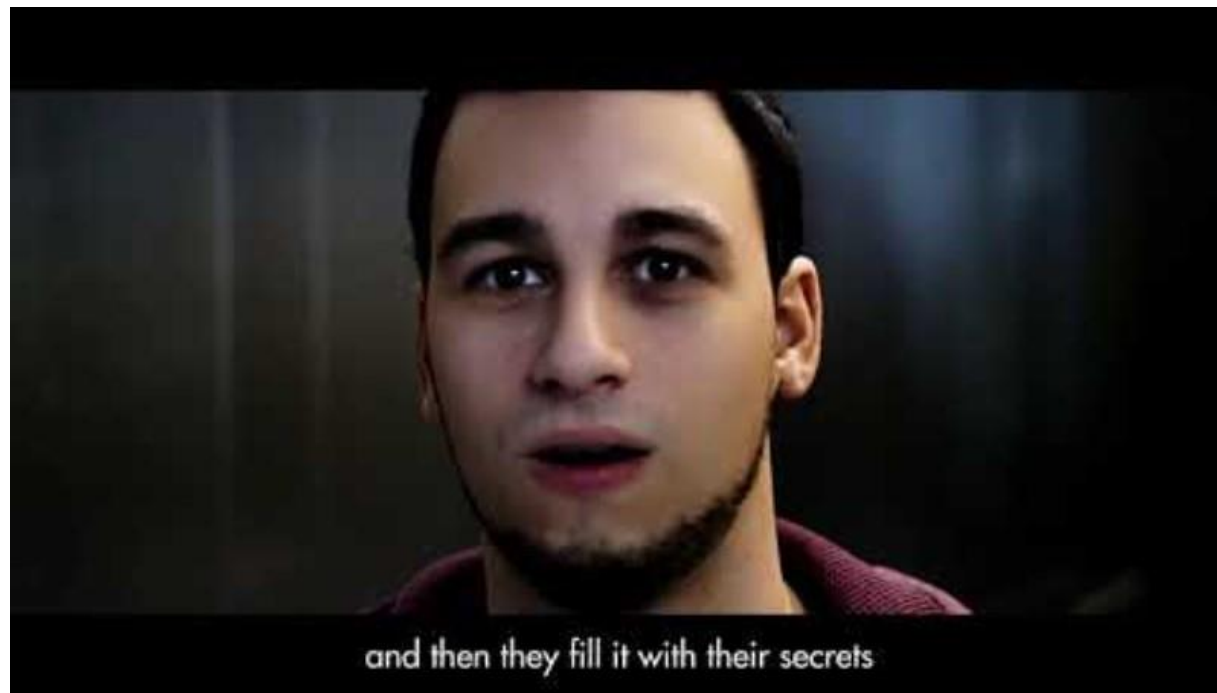
```
.....  
Ld1: uint8_t secret = *kernel_address;  
Ld2: uint8_t dummy = probe_array[secret*64];
```

- We generally consider Meltdown as a design **bug**
  - Similar “bugs” followed however

## Meltdown Followups

---

- MDS-microarchitectural data sampling
  - RIDL
  - Cacheout
  - Zombieload
- Crosstalk
- Downfall
- Reptar
- LVI-load value injection

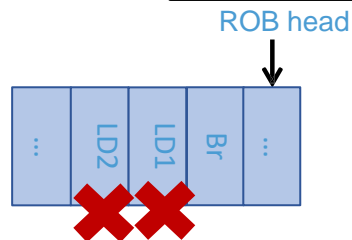


# Spectre Variant 1 – Exploit Branch Condition

- Consider the following kernel code, e.g., in a system call

```
Br:  if (x < size_array1) {  
Ld1:    secret = array1[x]  
Ld2:    y = array2[secret*64]  
      }
```

Always malicious?  
No. It may be a benign misprediction.  
We do not consider Spectre to be a bug.



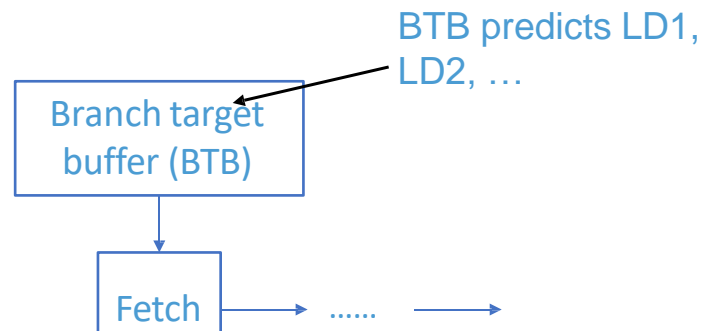
Attack to read arbitrary memory:

1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; `&array1[x]` maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of `array2` was fetched

## Spectre Variant 2 – Exploit Branch Target

- Most BTBs store partial tags **and targets...**
  - <last n bits of current PC, target PC>

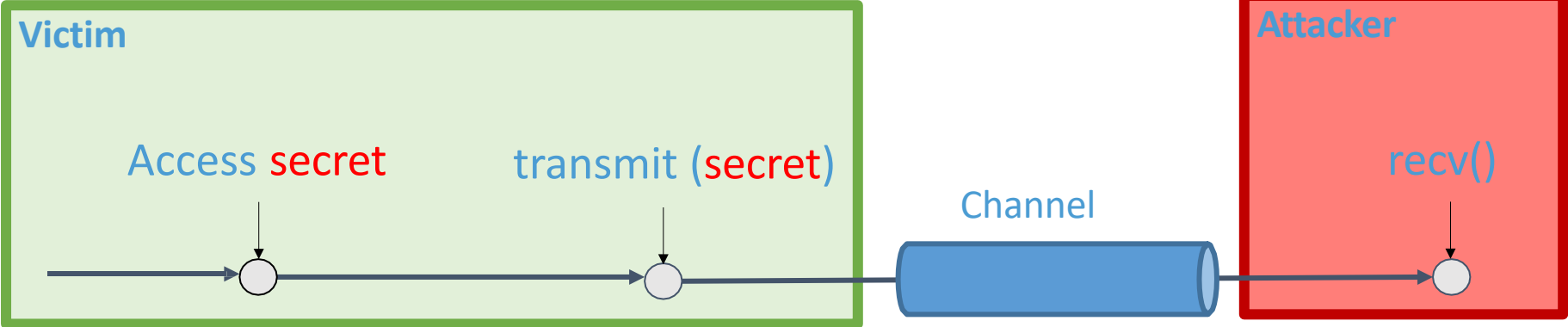
```
0xfff110 Br: jmp %eax
...
...
0xfff234 Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]
```



Train BTB properly → Execute arbitrary gadgets speculatively

# General Attack Schema

---



## Apply the General Attack Scheme

```
.....  
Ld1: uint8_t secret = *kernel_address;  
Ld2: unit8_t dummy = probe_array[secret*64];
```

```
Br:  if (x < size_array1) {  
Ld1:      secret = array1[x]  
Ld2:      y = array2[secret*64]  
      }
```

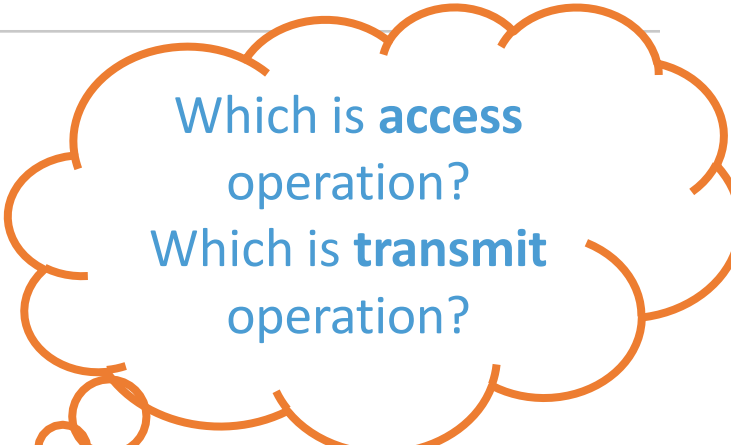
```
Br: jmp %eax
```

```
...
```

```
...
```

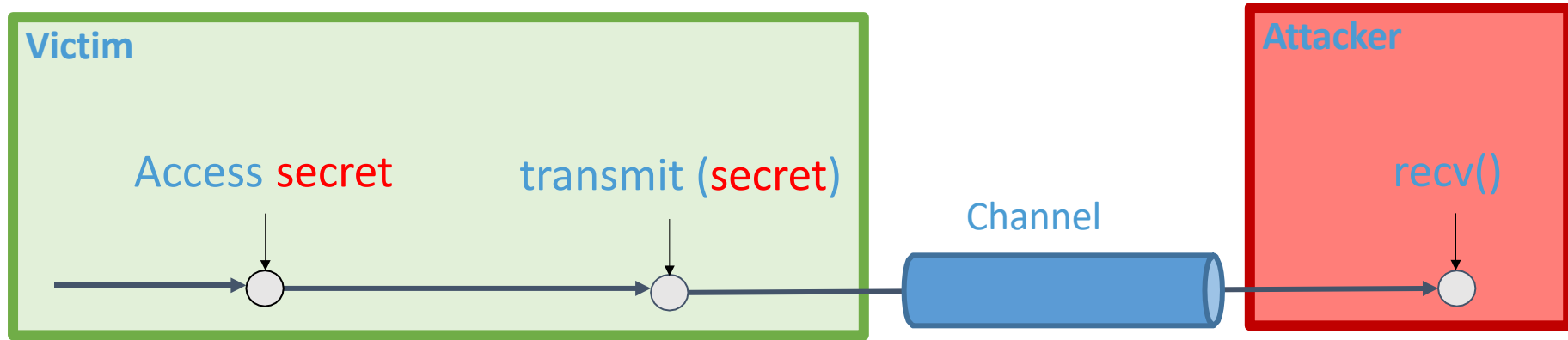
```
Ld1: secret = array1[x]
```

```
Ld2: y = array2[secret*4096]
```



Which is **access** operation?  
Which is **transmit** operation?

# General Attack Schema



- Transient attacks: can leak data-at-rest
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

"Easy" to fix

Hard to fix





THE UNIVERSITY  
*of* NORTH CAROLINA  
*at* CHAPEL HILL