Maximilian Mosler
Seminar Computer Architecture 2019
ETH Zurich

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Daniel Genkin[2], Daniel Gruss[3], Werner Haas[4], Mike Hamburg[5], Moritz Lipp[3], Stefan Mangard[3], Thomas Prescher[4], Michael Schwarz[3], Yuval Yarom[6]

[1]Independent, [2]University of Pennsylvania and University of Maryland, [3]Graz University of Technology, [4]Cyberus Technology, [5]Rambus Cryptography Research Division, [6]University of Adelaide and Data61

https://arxiv.org/pdf/1801.01203.pdf

Outline

- Executive Summary
- Background
- Overview
- Mechanisms in Detail
- Key Results
- Methodolgy
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Outline

- **Executive Summary**
- Background
- Overview
- Mechanisms in Detail
- Key Results
- Methodolgy
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Executive Summary

- Problem
  - Speculative execution can leak secret information
  - Growing focus on performance while neglecting system security
- Goal
  - Exploit speculative execution to gain access to confidential information
- Novelty
  - First showcase of exploiting speculative execution
- Key Approach
  - Exploiting conditional branches
  - Exploiting indirect branches
- Results
  - Attacks using native code and JavaScript
  - Unpatchable user space privilege attacks on correct code

Outline

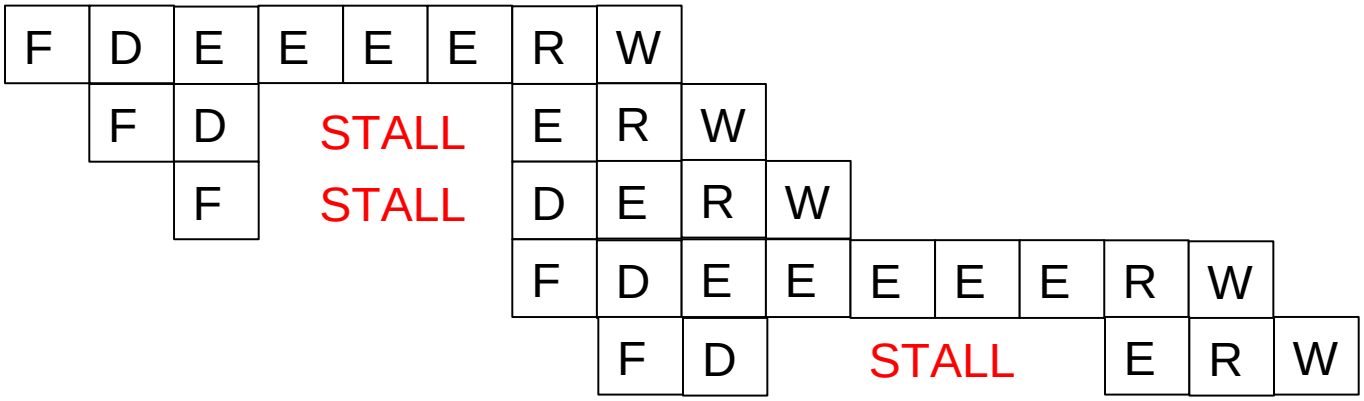
- Executive Summary
- **Background**
- Overview
- Mechanisms in Detail
- Key Results
- Methodolgy
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Background

- Out-of-order Execution
- Speculative Execution
- Branch Prediciton
- Memory Hierarchy
- Side-Channel Attacks
- Return-Oriented-Programming

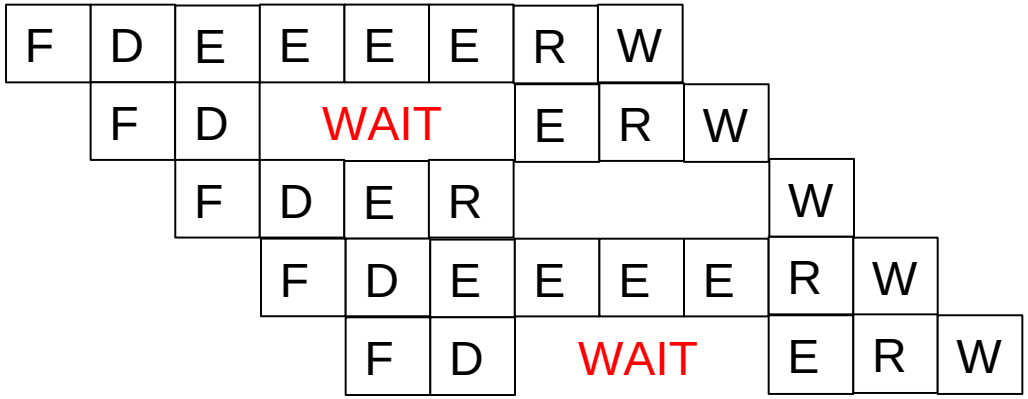
Out-of-order Execution

- In order dispatch + precise exceptions:



IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5

- Out-of-order dispatch + precise exceptions:



- 16 vs. 12 cycles

Speculative Execution

- Processor does not know future instruction stream of program
- Idea: Predict and speculatively execute likely execution path
  - Preserve current register state as checkpoint
- Abandon or commit changes made, based on if prediction turns out to be right
  - Revert to checkpoint if condition false
- Same worst case performance as non speculative execution, but reduced idling in all other cases

Speculative Execution

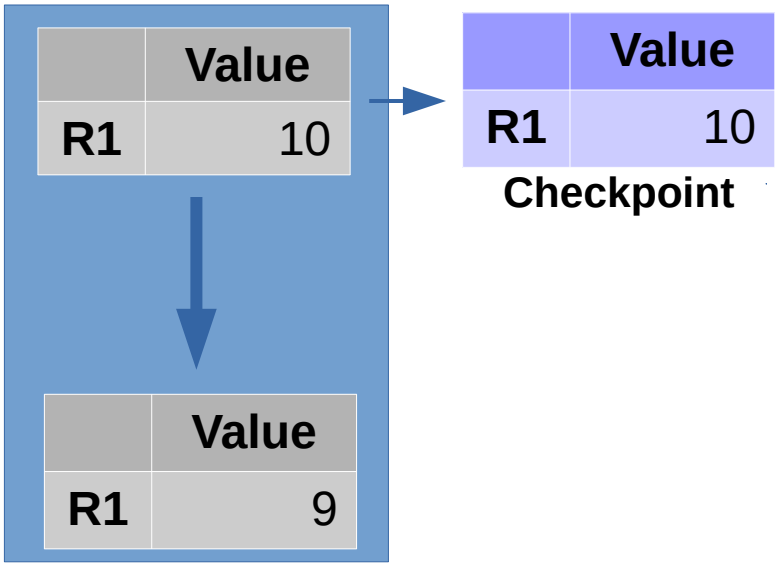
- Example:

```
loop:   CMP R1, 10    // compare contents of
                         R1 to 10
        JLE done      // if [R1] <= 10 then
                         end exectuion
        SUB R1, 1     // decrement R1
        JMP loop      // check condition again
done:
```

Speculative Execution

- Example:

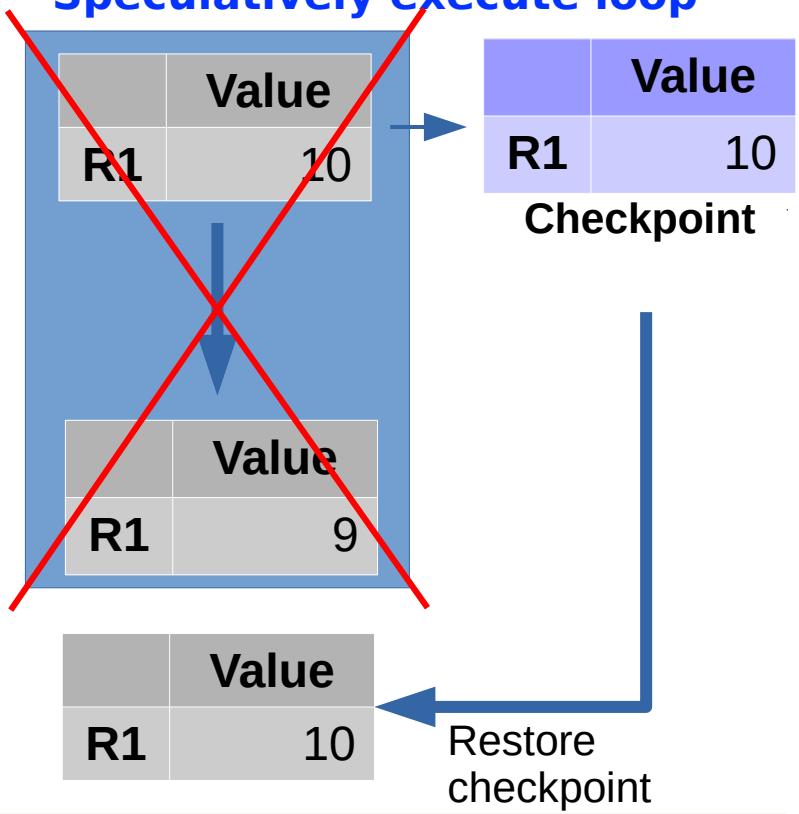
```
loop:   CMP R1, 10        →   Branch takes long to resolve

        JLE done          →   Speculatively execute loop

        SUB R1, 1         →

        JMP loop

done:
```



Speculative Execution

- Example:

```
loop:   CMP R1, 10      →   Branch takes long to resolve
        JLE done        →   Speculatively execute loop
        SUB R1, 1       →
        JMP loop
done:
```



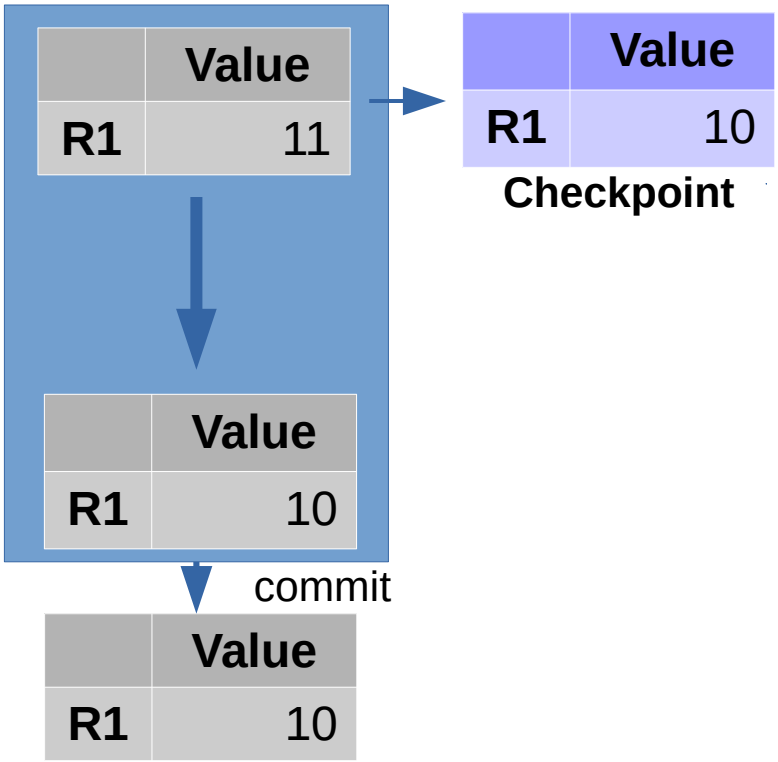
Speculative Execution

- Example:

```
loop:   CMP R1, 10
        JLE done
        SUB R1, 1
        JMP loop
done:
```

→ Branch takes long to resolve

→ Speculatively execute loop



Speculative Execution

- Reverting changes can still leave traces
  - Transient instructions are instructions that were performed erroneously, but may leave microarchitectural traces
- Nominal cache state unmodified, but cache might have new additional entries

Branch Prediction

- Speculative execution requires us to guess the likely execution path on branch instructions

- Branch prediction helps us make better guesses
  - More commited speculative executions

    → Increased perfomance

- Indirect branches can jump to arbitrary target addresses computed at runtime

- Conditional branches for which the execution path depends on a chosen condition

Branch Prediction

- Indirect branches
  - Jumping to an address stored in a register, memory location or stack, e.g., `jmp [eax]` in x86
  - Predictions rely on recent program behaviour

- Branch Target Buffer (BTB) is used to map addresses of recently excecuted instructions to dest. addresses
  - Predict future before decoding branch instruction

Branch Prediction

- Example indirect branch:
  - Assume our branch instruction has address 0x8
  - Assume that the address in eax is uncached

First Run:

```
jmp [eax]
```

Branch Target Buffer

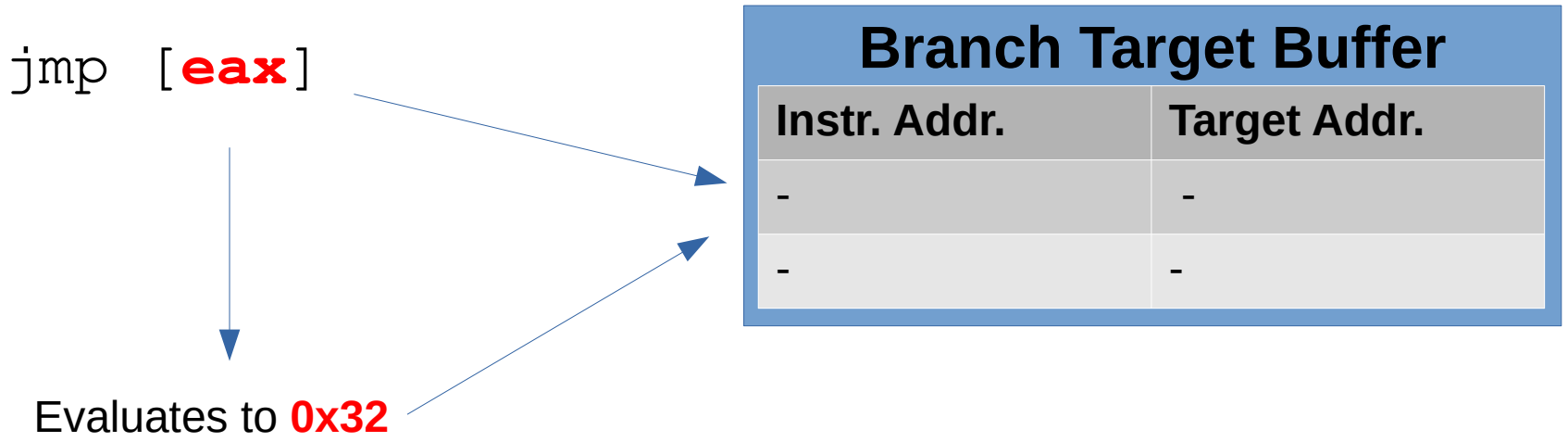
Instr. Addr.	Target Addr.
-	-
-	-

Branch Prediction

- Example Indirect Branch:
  - Assume our branch instruction has address 0x8
  - Assume that the address in eax is uncached

First Run:

```
jmp [eax]
```

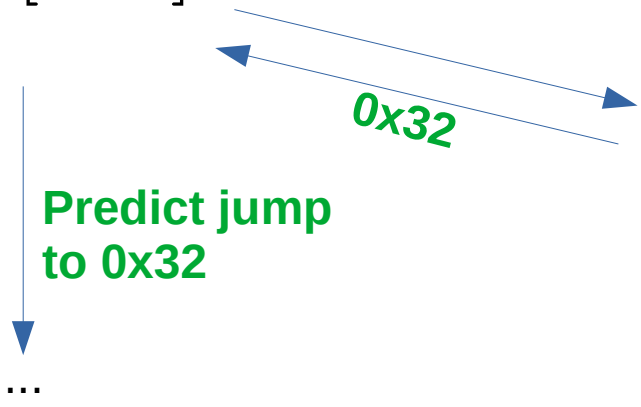


Branch Prediction

- Example Indirect Branch:
  - Assume our branch instruction has address 0x8
  - Assume that the address in eax is uncached

Second Run:

```
jmp [eax]
```



Branch Target Buffer

Instr. Addr.	Target Addr.
0x8	0x32
-	-

Branch Prediction

- Conditional branches
  - Branch instructions like if-statements

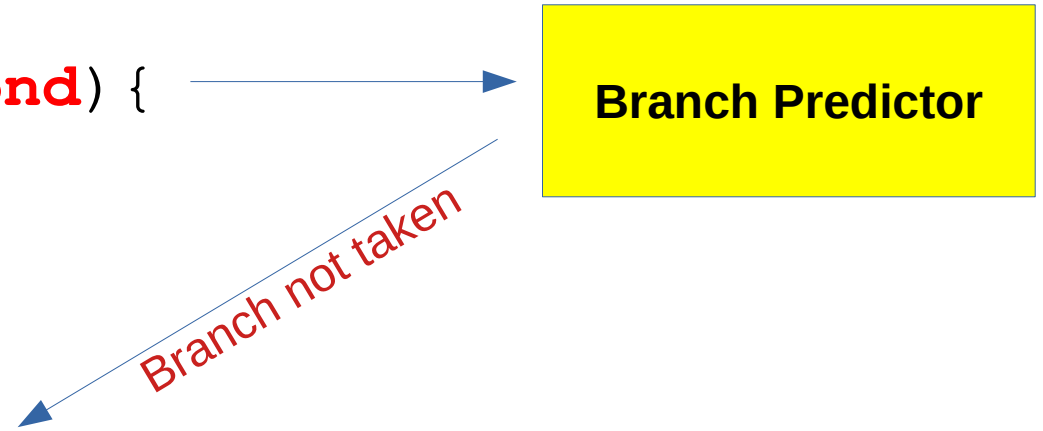
    `if(a) then` *`dest1`* `else` *`dest2`*
  - Recording target address is not required, since the destination is encoded in the instruction
  - Condition is determined at runtime

- Processor maintains a record of recent branch outcomes for indirect and direct branches, called the branch predictor

Branch Prediction

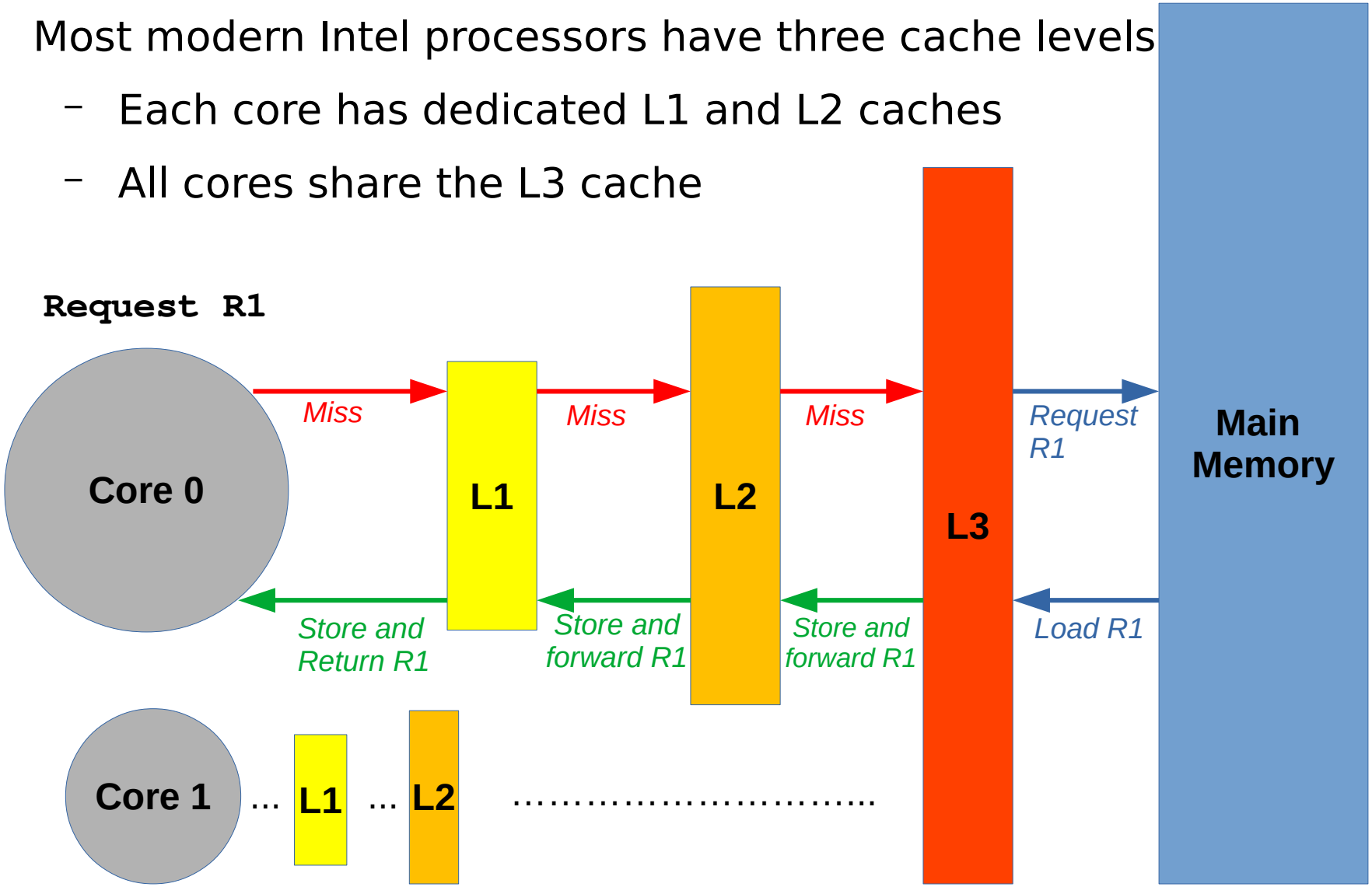
- Example conditional branch:
  - Assume uncached_cond is a uncached boolean variable

```
if (uncached_cond){
   expression1;
}else{
   expression2;
}
```



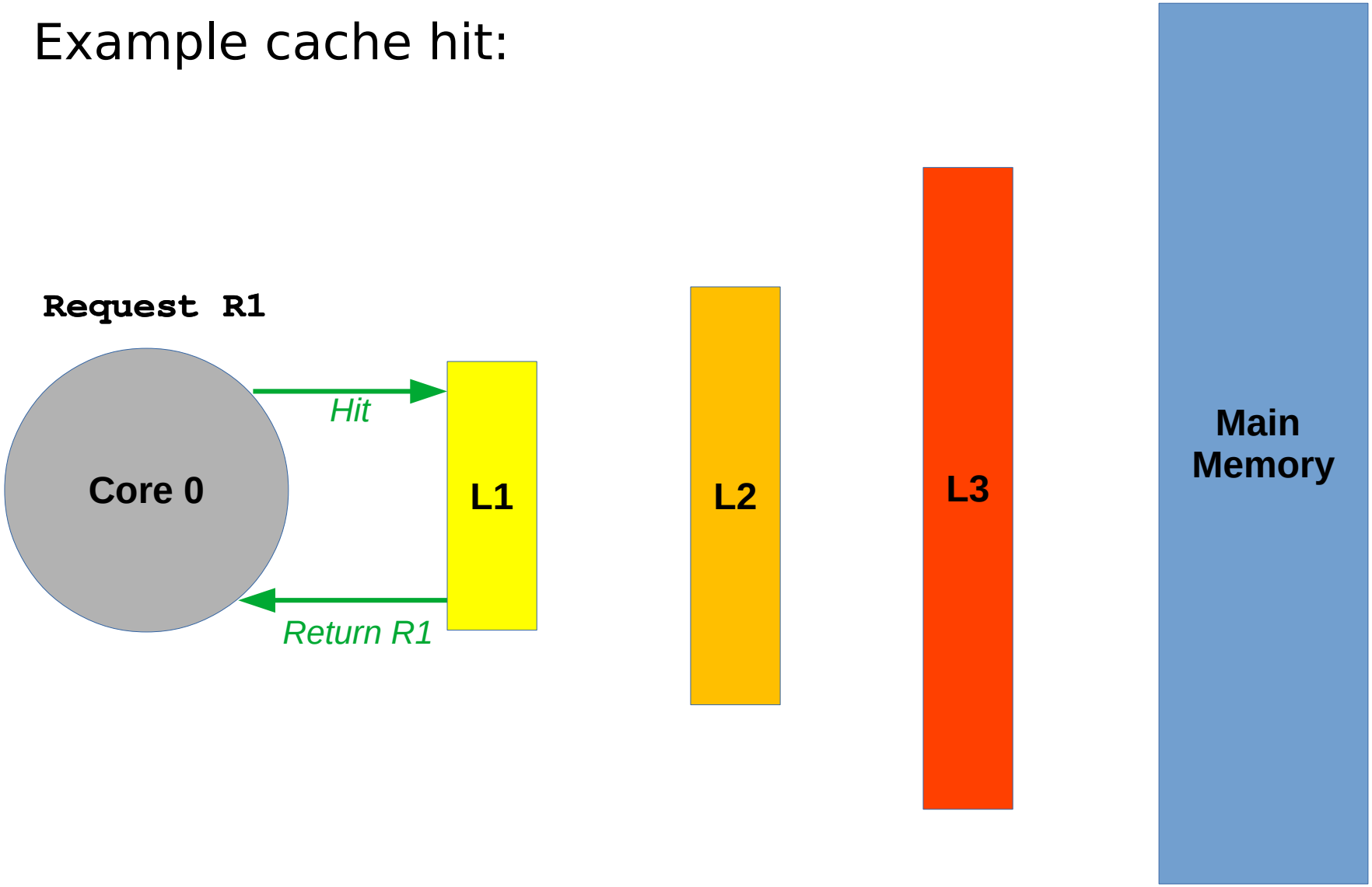
Memory Hierarchy

- Most modern Intel processors have three cache levels
  - Each core has dedicated L1 and L2 caches
  - All cores share the L3 cache



Memory Hierarchy

- Example cache hit:



Memory Hierarchy

- Processor must ensure cache coherence per core
  - Cache coherence protocol like MESI

    → Write on one core leads to invalidation of data in other cores, for L1 and L2

    → cache line bouncing if this happens repeatedly to one specific memory location
- False sharing when two cores bounce the same cache line by accessing nearby memory addresses
- We will later abuse these properties for our Evict+Reload approach of recovering leaked data

Microarchitectural Side-Channel Attacks

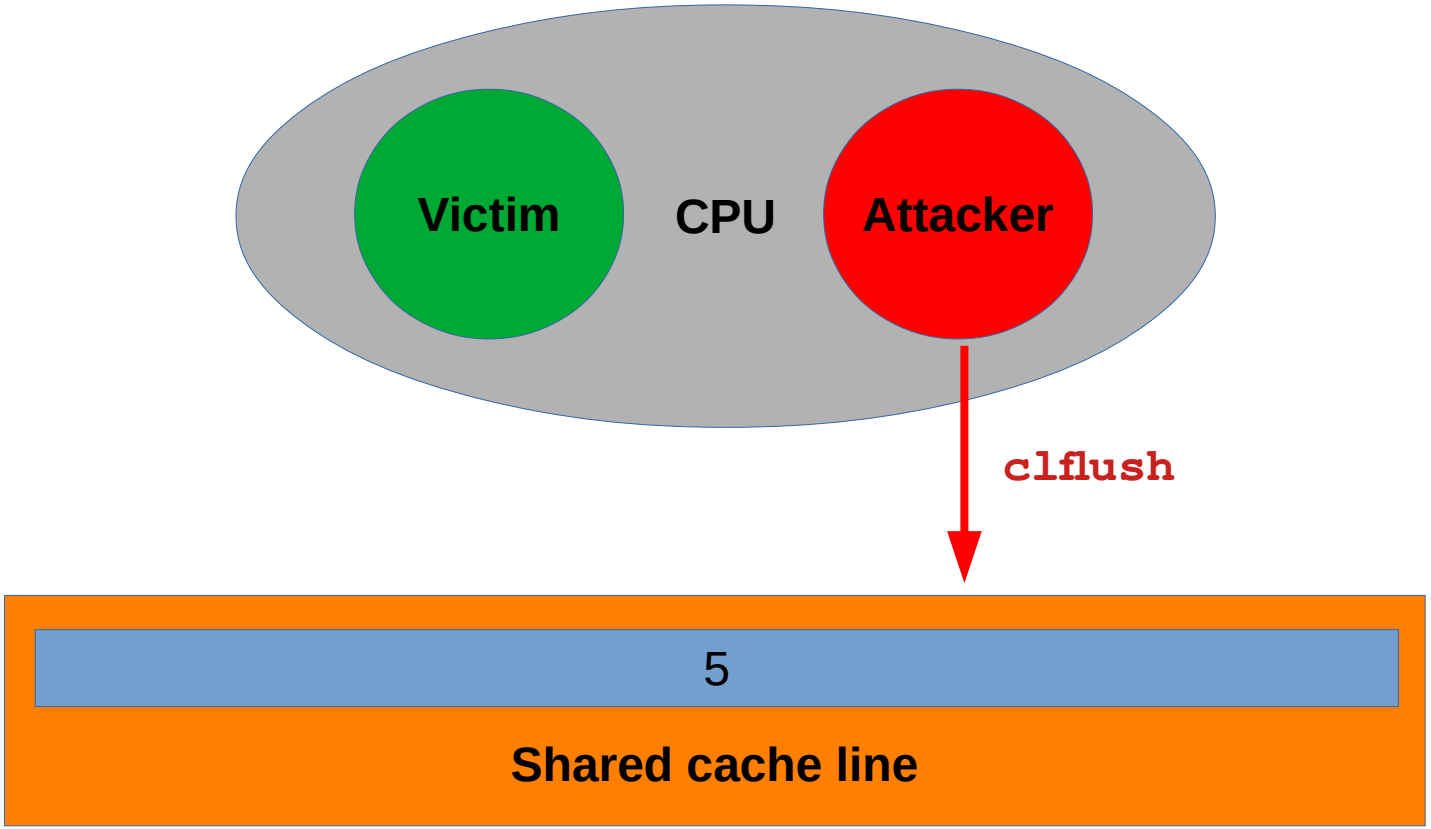
- Changes in the microarchitectural state caused by one program may affect other programs
  - Can leak information from program to program

- We focus on Flush+Reload and Evict+Reload
  - Techniques for recovering the leaked information

- Idea: Evict/Flush victim shared cache lines, let victim execute, and probe the shared lines
  - Probe by measuring access times
    - → *fast access = victim used cache line*
    - → *slow access = cache line not used*

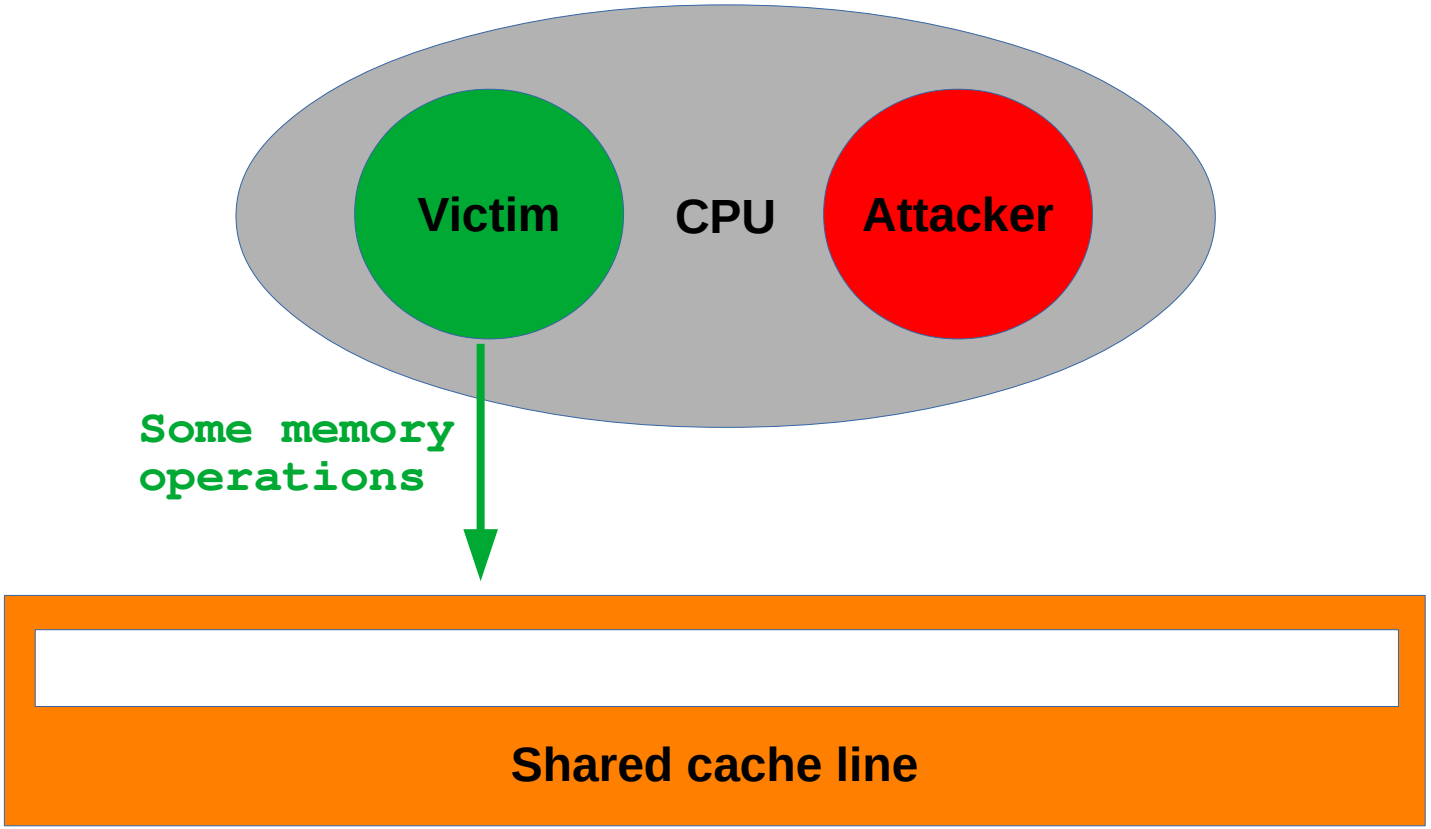
Microarchitectural Side-Channel Attacks

- Example Flush+Reload:
- We use a dedicated machine instruction, like `clflush` in x86, to evict the line



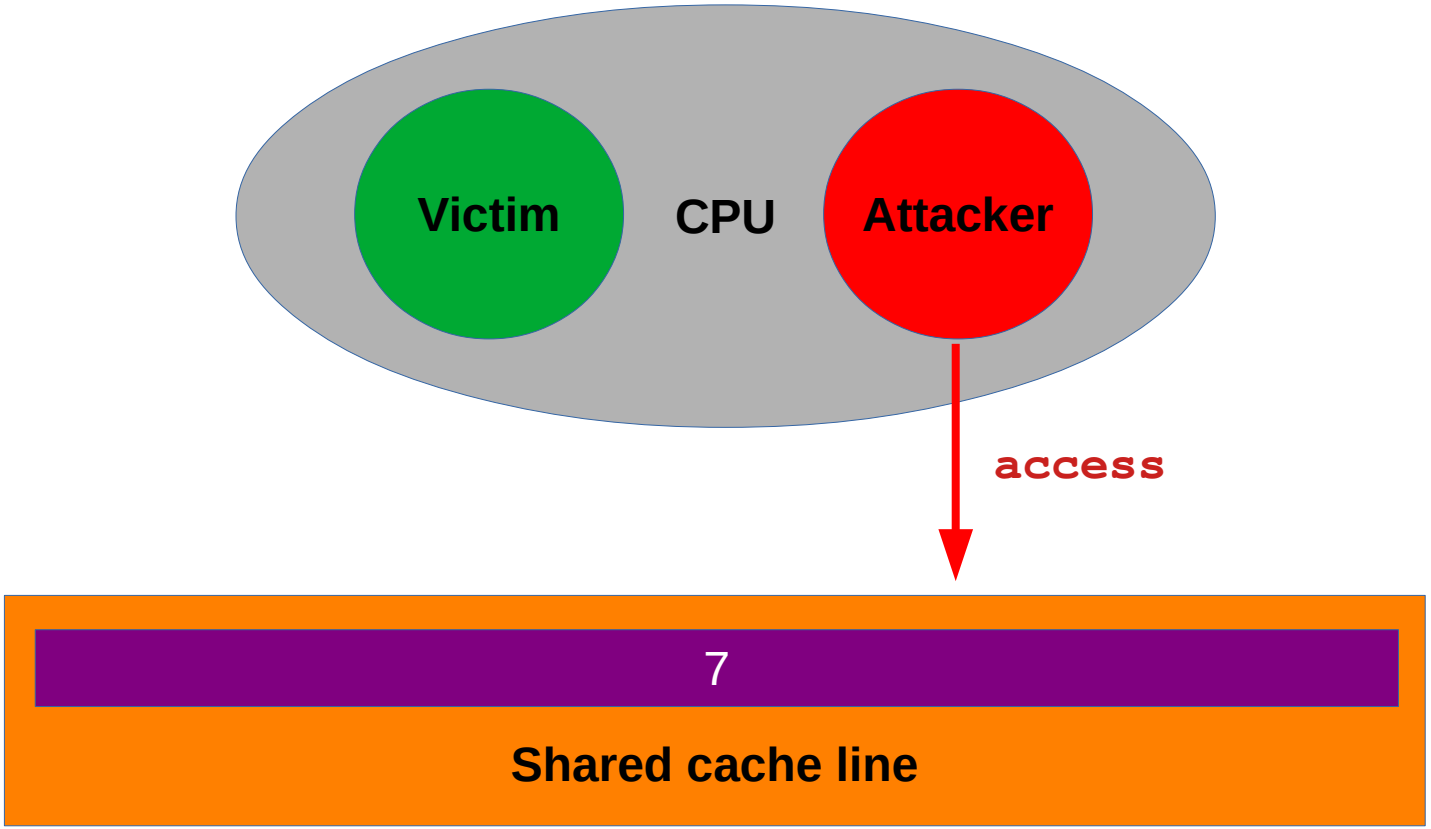
Microarchitectural Side-Channel Attacks

- Example Flush+Reload:
- We use a dedicated machine instruction, like `clflush` in x86, to evict the line



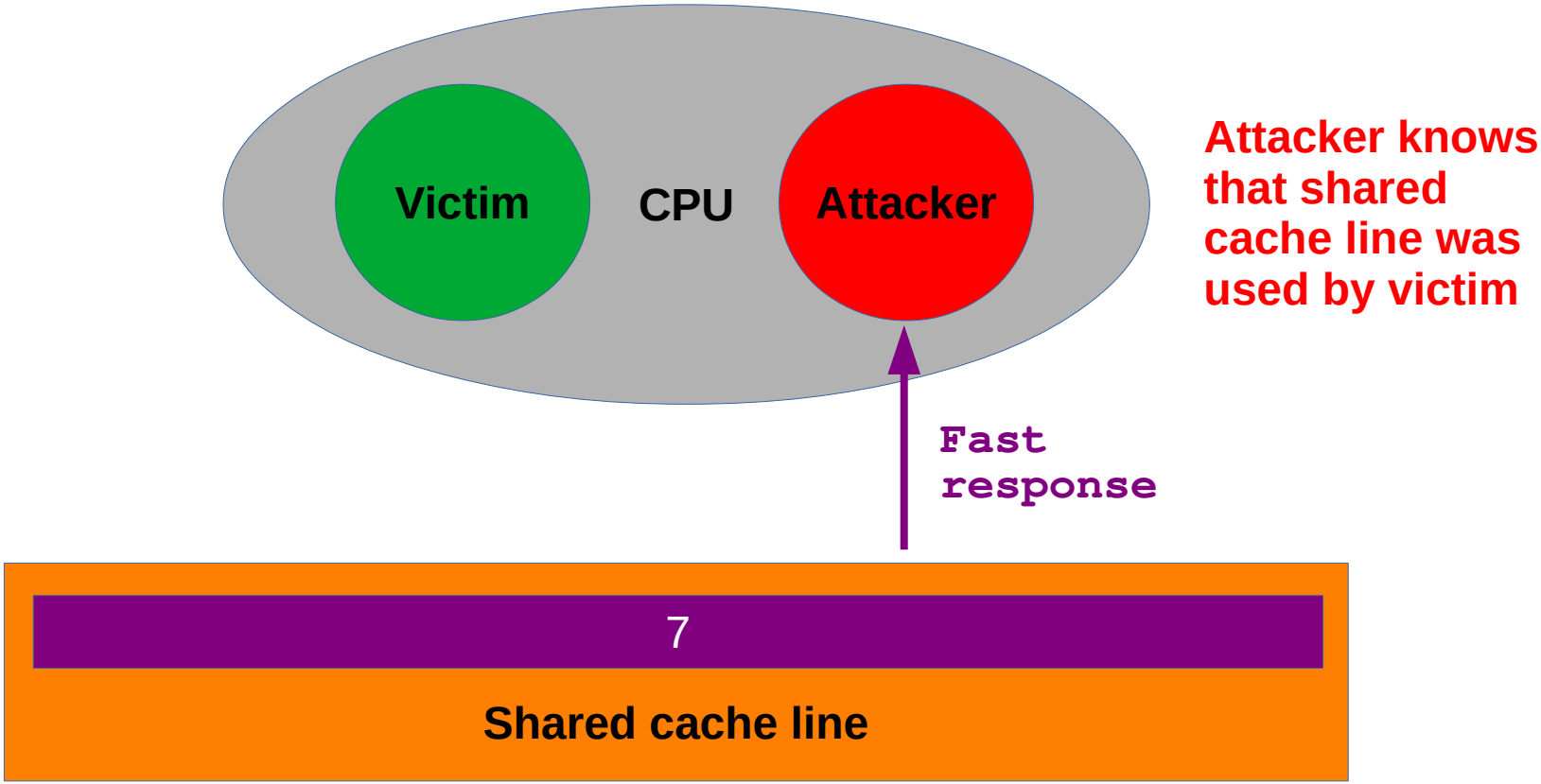
Microarchitectural Side-Channel Attacks

- Example Flush+Reload:
- We use a dedicated machine instruction, like `clflush` in x86, to evict the line



Microarchitectural Side-Channel Attacks

- Example Flush+Reload:
- We use a dedicated machine instruction, like `clflush` in x86, to evict the line



Return-Oriented Programming

- Idea: Hijack control flow of a vulnerable victim program

- Gadgets are machine code snippets found in the victims code
  - Perform some computation and then return
  - Search binary for useful gadgets

- If attacker has control of stack pointer, he can chain execute gadgets by changing the return address
  - Achieved using e.g. buffer overflow exploits

Outline

- Executive Summary
- Background
- **Overview**
- Mechanisms in Detail
- Key Results
- Methodolgy
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Spectre Attack Overview

- Setup Phase
  - Mistrain processor for erroneous speculative execution
  - Manipulate cache state
  - Setup side channel
- Second Phase
  - Invoke speculative execution of victim program
  - Transfer confidential information into side channel
- Third Phase
  - Use Flush+Reload or Evict+Reload to recover leaked information

    → time access on cache line for memory addresses

Outline

- Executive Summary
- Background
- Overview
- **Mechanisms in Detail**
- Key Results
- Methodolgy
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Mechanisms (in some detail)

- Spectre attacks come in many variants
  - Speculative execution used in different contexts

- We will focus on two conepts:
  - Poisoning indirect branches
  - Exploiting conditional branch misprediction

- Furthermore we will also see how mistraining works

Mistraining Branch Prediction

- Methods vary among CPUs
- Attacker mimics the pattern of branches leading up to the branch to be mispredicted
  - Place jumps at the same virtual address as in victim proccess
  - Has to be done on same CPU core
- Predictors also learn from illegal operations

Indirect Branch Poisoning

- Similiar to return oriented programming
- Assume attacker has control over registers R1, R2
- Assume we have located two gadgets in the victims code
  - G1 = adds address of R1 onto R2
  - G2 = access memory at R2
- Attacker controls attack via:
  - R1 → which address to leak
  - R2 → map memory to address to read in G2
- Gagdet must reside in memory executable by victim

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```

- Setup Phase

Shared Cache Line

	Tag	Value	Tag	Value
Set 0	03	0x12	07	0x06

Cache

Branch Target Buffer

Instr. Addr.	Target Addr.
-	-
0x24	0x16

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```

Shared Cache Line

	Tag	Value	Tag	Value
Set 0	03	0x12	07	0x06

Cache

- Setup Phase
  - Mistrain BTB

    → Attacker calls `jmp [eax]`

    with address to G1 in `eax`

Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```

Shared Cache Line

	Tag	Value	Tag	Value
Set 0	03	0x12	07	0x06

Cache

- Setup Phase
  - Make sure `eax` is not in cache

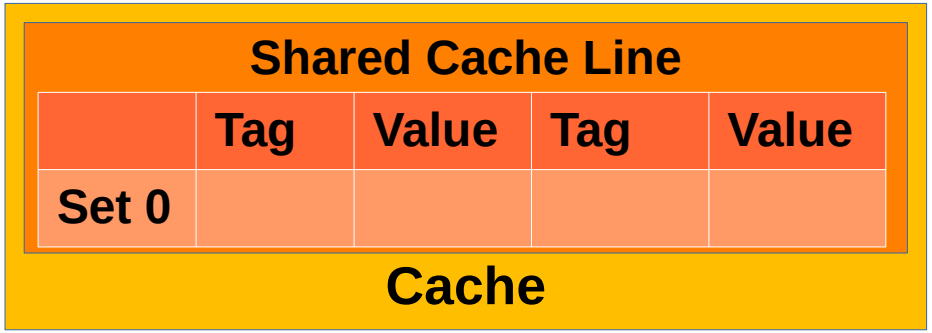
    → evict/flush

Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```



- Setup Phase
  - Make sure `eax` is not in cache

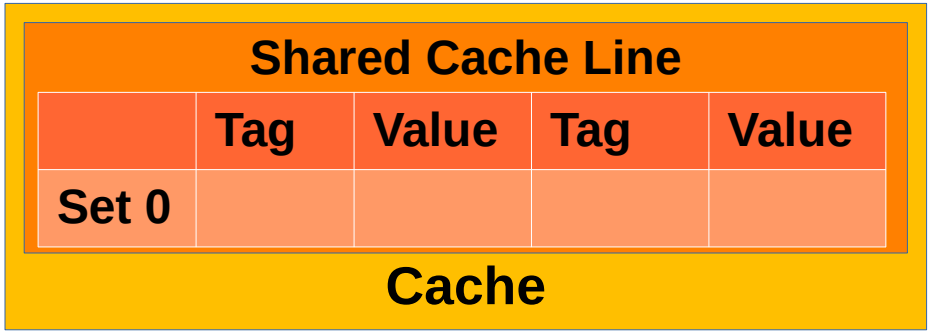
    → evict/flush
  - flush/evict shared cache line

Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```



- Second Phase
  - Victim is invoked and starts executing

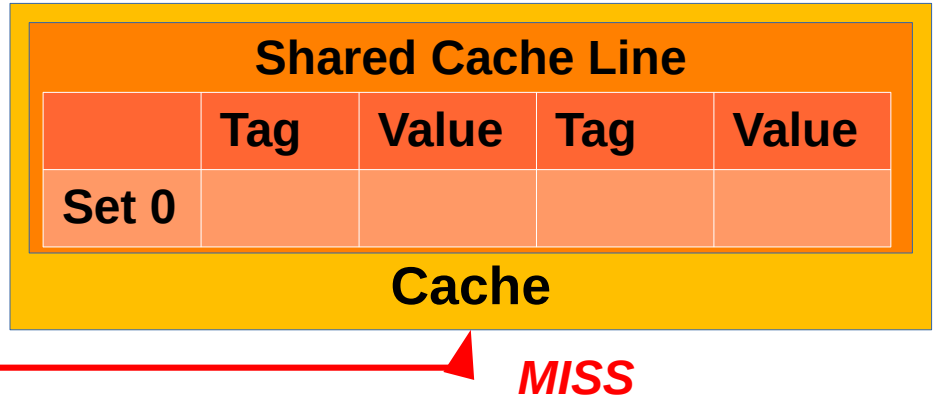
Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```

- Second Phase
  - Victim is invoked and starts executing



Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```

- Second Phase
  - Victim is invoked and starts executing

Execute G1

Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Shared Cache Line

	Tag	Value	Tag	Value
Set 0				

Cache

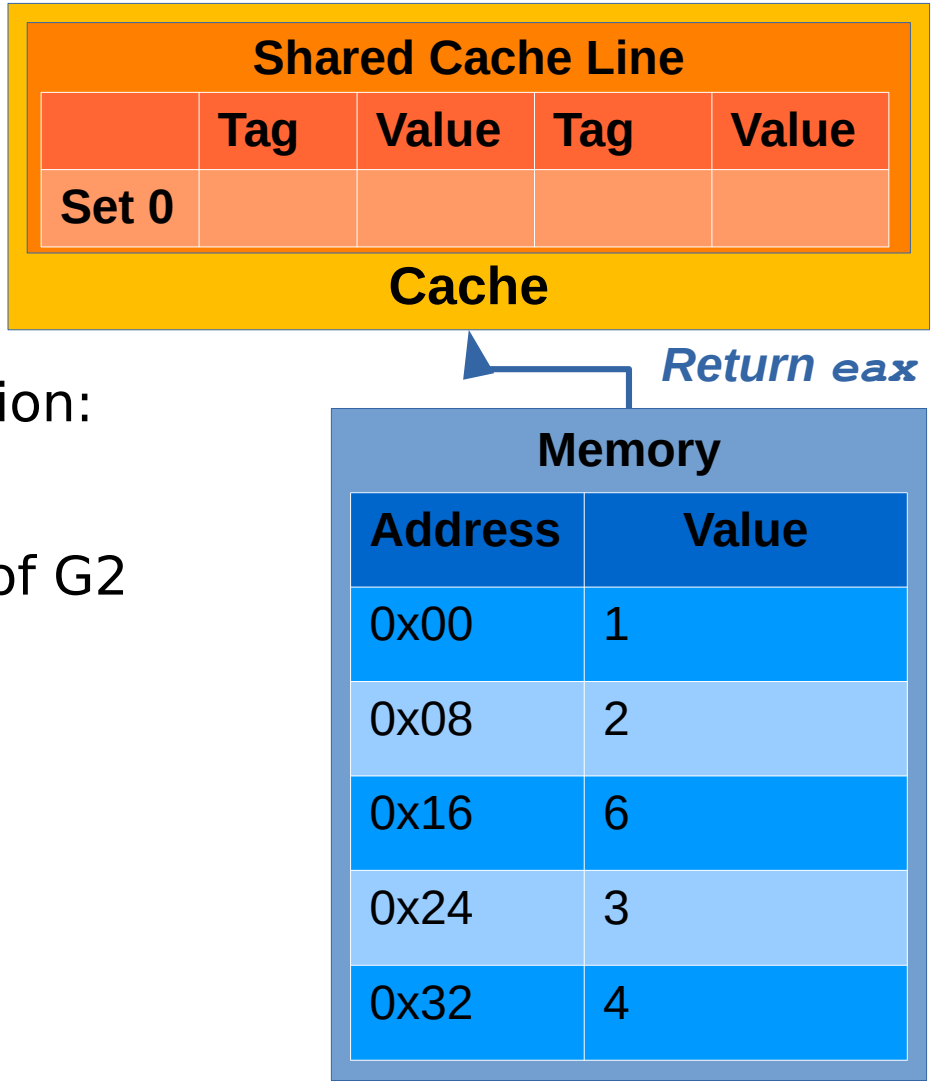
Load eax

Memory

Address	Value
0x00	1
0x08	2
0x16	6
0x24	3
0x32	4

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```



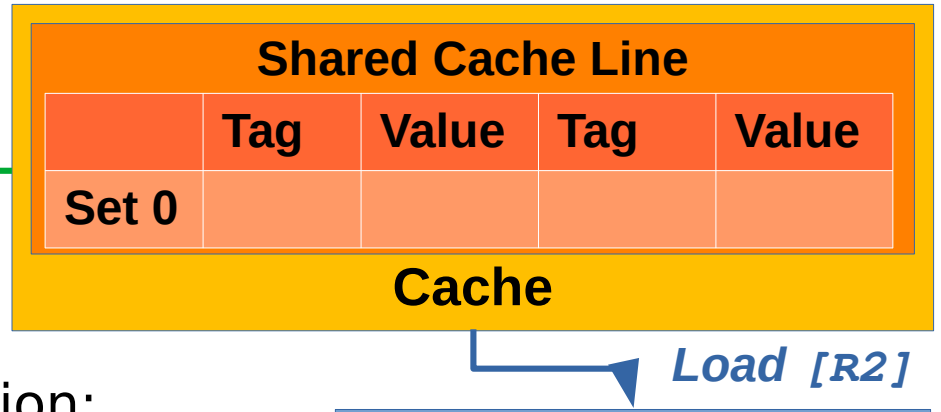
- Second Phase
  - During speculative execution:
    - G1 adds R1 to R2
    - G1 returns to address of G2

Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```



- Second Phase
  - During speculative execution:

    → G2 accesses memory address stored in R2 → leaks it into cache

Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Memory

Address	Value
0x00	1
0x08	2
0x16	6
0x24	3
0x32	4

Spectre Attack: Poisoning Indirect Branches

```
jmp [eax];
```

Shared Cache Line

	Tag	Value	Tag	Value
Set 0			03	6

Cache

- Third Phase
  - Use Flush+Reload or Evict+Reload to recover data from shared cache
    → recover value 6 from second block

Branch Target Buffer

Instr. Addr.	Target Addr.
jmp [eax]	G1
0x24	0x16

Memory

Address	Value
0x00	1
0x08	2
0x16	6
0x24	3
0x32	4

Exploiting Conditional Branch Mispredicition

- Consider the following code:

```
if(x < array1_size)
   y = array2[array1[x] * 4096];
```

- Assume `x` is an input from an untrusted source
- `array1` is of size `array1_size` and `array2` is of size 1MB
- The bounds check keeps program from accessing potentially sensitive memory, supplying

  `x` = (addr. of secret byte to read) – (addr. of `array1`)

Exploiting Conditional Branch Mispredicition

- Consider the following code:

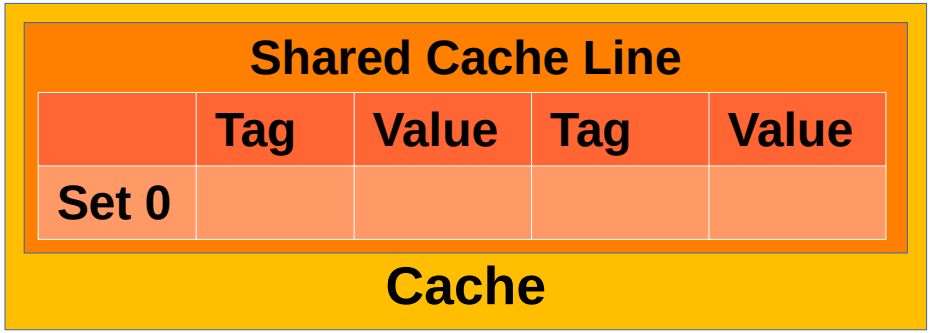
```
if(x < array1_size)
   y = array2[array1[x] * 4096];
```

- Now assume `x` was maliciously chosen
  - `k = array1[x]` resolves to secret byte in victim memory
- Assume `array1_size` and `array2` are uncached
- Assume previous values of `x` were valid

→ if `k` is cached then speculative execution loads `array2[k * 4096]` into cache

Spectre Attack: Conditional Branching

```
if(x < array1_size)
    y = array2[array1[x]];
```



- Setup Phase
  - Train branch predictor with valid `x` values

    `victim(0), victim(1)` ⟷ **Branch Predictor**

  - Manipulate cache by evicting `array1_size` and `array2`
  - Setup side channel by flushing the monitored cache line
  - Get kernel to cache secret byte `k` in legit operation

Spectre Attack: Conditional Branching

```
if(x < array1_size)
    y = array2[array1[x]];
```

MISS

Shared Cache Line				
	Tag	Value	Tag	Value
Set 0				

Cache

- Second Phase
  - Invoke malicious execution

```
victim(3)
```

Branch Predictor

Memory	
Address	Value
0x00	1
0x08	2
0x16	6
0x24	3
0x32	4

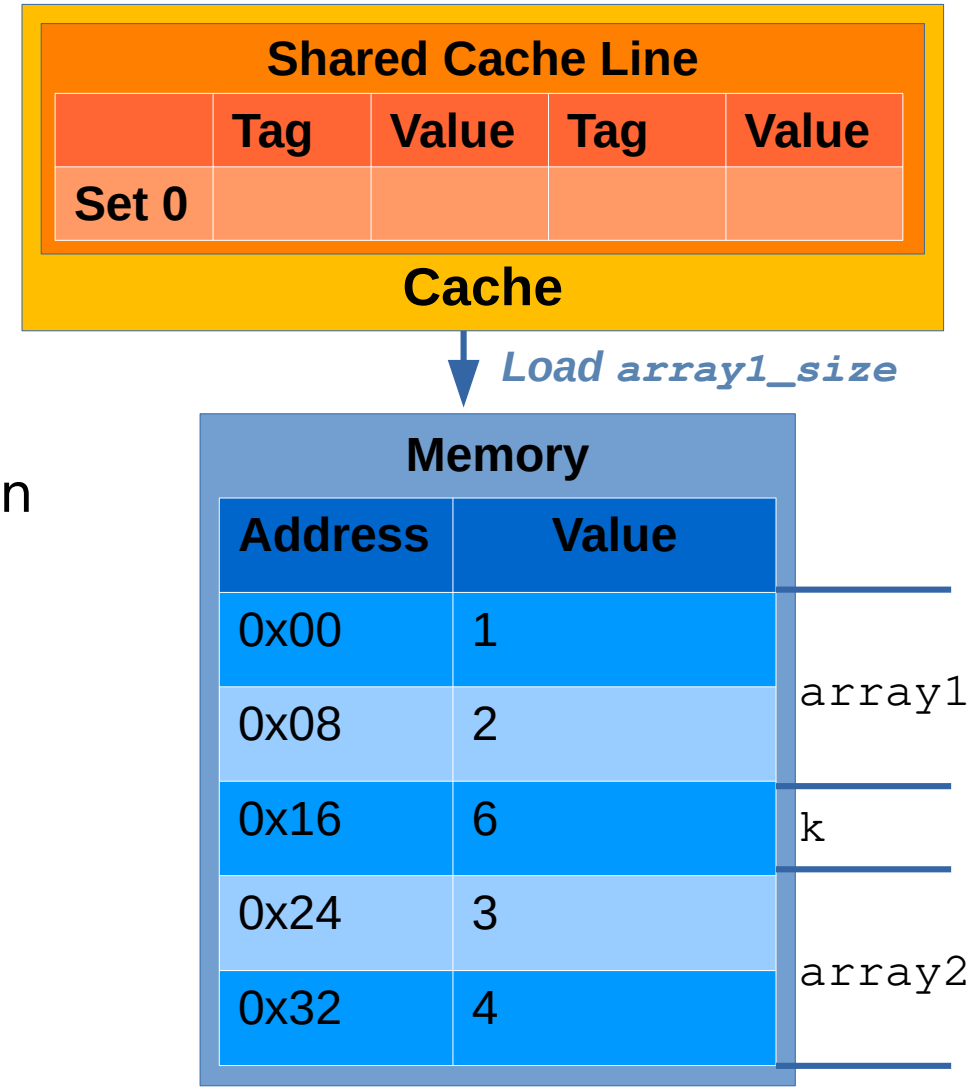
array1

k

array2

Spectre Attack: Conditional Branching

```
if(x < array1_size)
    y = array2[array1[x]];
```



- Second Phase
  - Invoke malicious execution

    `victim(3)`

Spectre Attack: Conditional Branching

```
if(x < array1_size)
    y = array2[array1[x]];
```

HIT

Shared Cache Line				
	Tag	Value	Tag	Value
Set 0				

Cache

ret

Return `array1_size`

- Second Phase
  - Invoke malicious execution

```
victim(3)
```

Branch Predictor

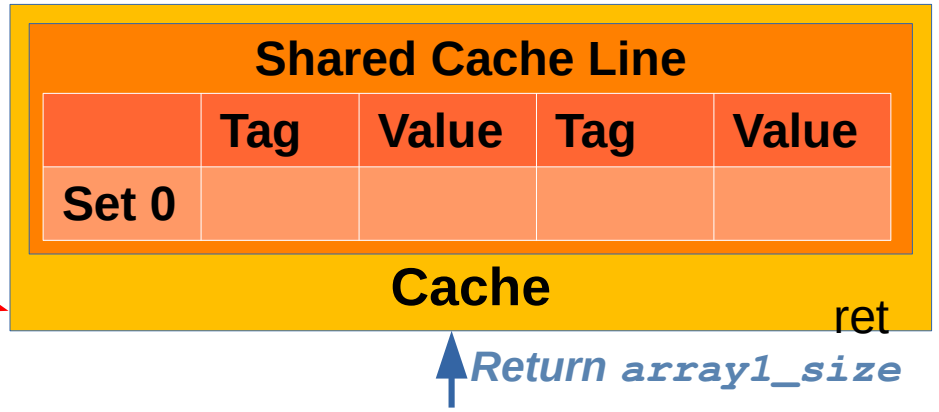
Memory

Address	Value	
0x00	1	array1
0x08	2	
0x16	6	k
0x24	3	array2
0x32	4	

Spectre Attack: Conditional Branching

```
if(x < array1_size)
    y = array2[array1[x]];
```

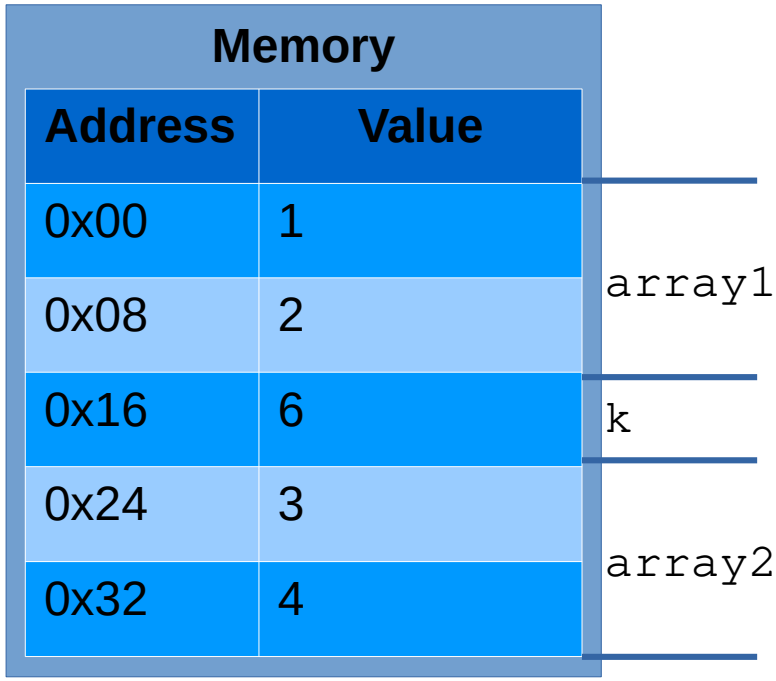
MISS



- Second Phase
  - Invoke malicious execution

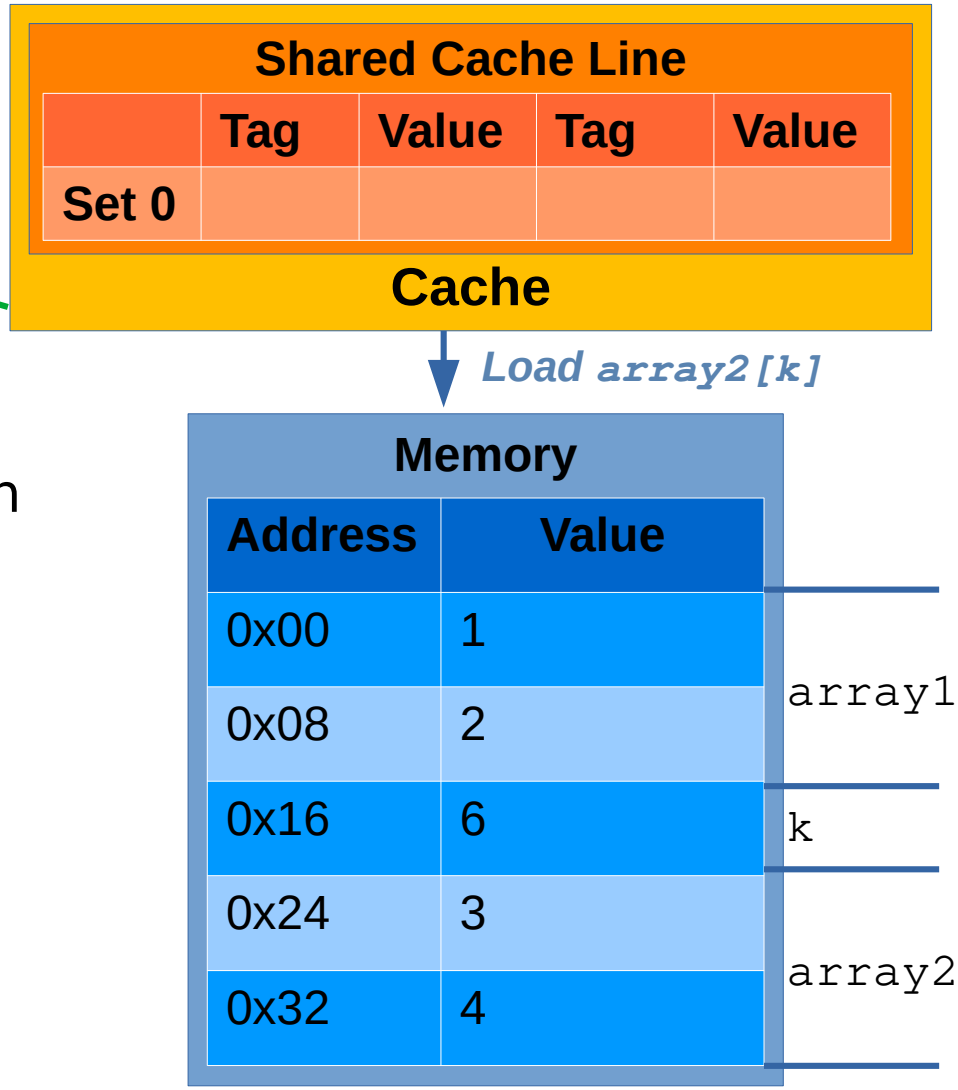
```
victim(3)
```

Branch Predictor



Spectre Attack: Conditional Branching

```
if(x < array1_size)
    y = array2[array1[x]];
```



- Second Phase
  - Invoke malicious execution

```
victim(3)
```

Branch Predictor

Spectre Attack: Conditional Branching

```
if(x < array1_size)
    y = array2[array1[x]];
```

Shared Cache Line

	Tag	Value	Tag	Value
Set 0	01	3		

Cache

- Third Phase
  - Recover leaked information

    → probe for `array2[k]`

Branch Predictor

Memory

Address	Value	
0x00	1	`array1`
0x08	2	
0x16	6	`k`
0x24	3	`array2`
0x32	4	

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- **Methodology**
- Key Results
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Methodology

- Test for conditional branching attacks were performed on multiple x86 processors
  - Intel Ivy Bridge, Haswell, Skylake
  - AMD Ryzen
  - 64- and 32-bit modes
  - Windows and Linux
- ARM processors that support speculative execution
- Implementations in C and JavaScript tested
- Most tests performed on i7 Surface Pro 3 (i7-4650U)

Methodology

- Tests for indirect branch poisoning attacks primarily perfomed on Haswell-based Surface Pro 3
  - 32-bit Windows applications were tested
  - Windows 8 was used as the only OS
- Skylake was also tested for BTB manipulation

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- Methodolgy
- **Key Results**
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Key Results

- Attacks using user space privileges that do not require any code vulnerabilities
  - Not patchable through microcode or software

    → Stop gap measures
- No way to tell whether particular code is safe or not
- Performance implications are harsh
  - Need to disable hyperthreading and flushes during context switches
  - Speculative execution has to be halted on potentially sensitive execution paths
- Updates to ISA and CPU implementations required

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- Methodolgy
- Key Results
- **Summary**
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Executive Summary

- Problem
  - Speculative Execution can leak secret information
  - Growing focus on Performance while neglecting system security
- Goal
  - Exploit speculative execution to gain access to confidential information
- Novelty
  - First showcase of exploiting speculative execution
- Key Approach
  - Exploiting conditional branches
  - Exploiting indirect branches
- Results
  - Attacks using Native Code and JavaScript
  - Unpatchable user space privilege attacks on correct code

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- Methodolgy
- Key Results
- Summary
- **Strengths**
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion

Strengths

- Good introduction
  - Gives refresher on almost all important concepts
  - Easy to read due to abstraction
- First paper to exploit speculative execution in this context
- Explores further ideas to abuse this problem
  - Two main variations thoroughly explained
  - Several others mentioned

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- Methodolgy
- Key Results
- Summary
- Strengths
- **Weaknesses**
- Thoughts and Ideas
- Takeaways
- Open Discussion

Weaknesses

- Very poorly written
  - Reiterates on introduction a lot
  - Structure seems arbitrary
  - Not proofread
- Fails to maintain consistent level of abstraction
  - Jumps between high level concepts and low level implementations
- Inital testing very limited
  - Most tests performed on Surface Pro 3

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- Methodolgy
- Key Results
- Summary
- Strengths
- Weaknesses
- **Thoughts and Ideas**
- Takeaways
- Open Discussion

Thoughts and Ideas

- Read the revised version of the paper
  https://spectreattack.com/spectre.pdf

- Or watch the talk given at the 40th IEEE Symposium on Security and Privacy
  https://youtu.be/zOvBHxMjNls

- Meltdown is different from spectre, since it abuses special privileges given to out-of-order executed instructions on Intel processors
  - Fix applied with KAISER patch

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- Methodolgy
- Key Results
- Summary
- Strengths
- Weaknesses
- Thoughts and Ideas
- **Takeaways**
- Open Discussion

Takeaways

- Possibly one of the biggest media impacts of any system vulnerability of the decade

- Hunt for better performance has lead to negligence concerning system security

The Guardian, Jan. 2018

Takeaways

- Possibly one of the biggest media impacts of any system vulnerability of the decade

- Hunt for better performance has lead to negligence concerning system security

- ***"A Systematic Evaluation of Transient Execution Attacks and Defenses"*** - Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, Daniel Gruss, pub. Nov 2018, last rev. May 2019, https://arxiv.org/pdf/1811.05441.pdf

- ***"A New Memory Type against Speculative Side Channel Attacks"*** - Ke Sun, Rodrigo Branco, Kekai Hu, Intel - STrategic Offensive Research & Mitigations (STORM), pub. September 2019, www.scribd.com

Outline

- Executive Summary
- Background
- Overview
- Mechanisms (in some detail)
- Methodolgy
- Key Results
- Summary
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- **Open Discussion**

Open Discussion

Discussion Starters

- How useful is this in reality?
- How important is it to address this?
- Where do we go from here?