Comp 590-184: Hardware Security and Side-Channels

Lecture 10: Spectre

February 17, 2026

Andrew Kwong

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

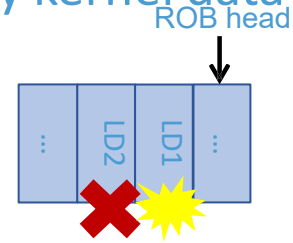
Slides adapted from Wei Wang and Mengjia Yan (shd.mit.edu)

Meltdown

- Put two optimizations together, we have Meltdown
  - Hardware optimization: out-of-order execution
    - Deferred exception handling
  - Software optimization: mapping kernel addresses into user space

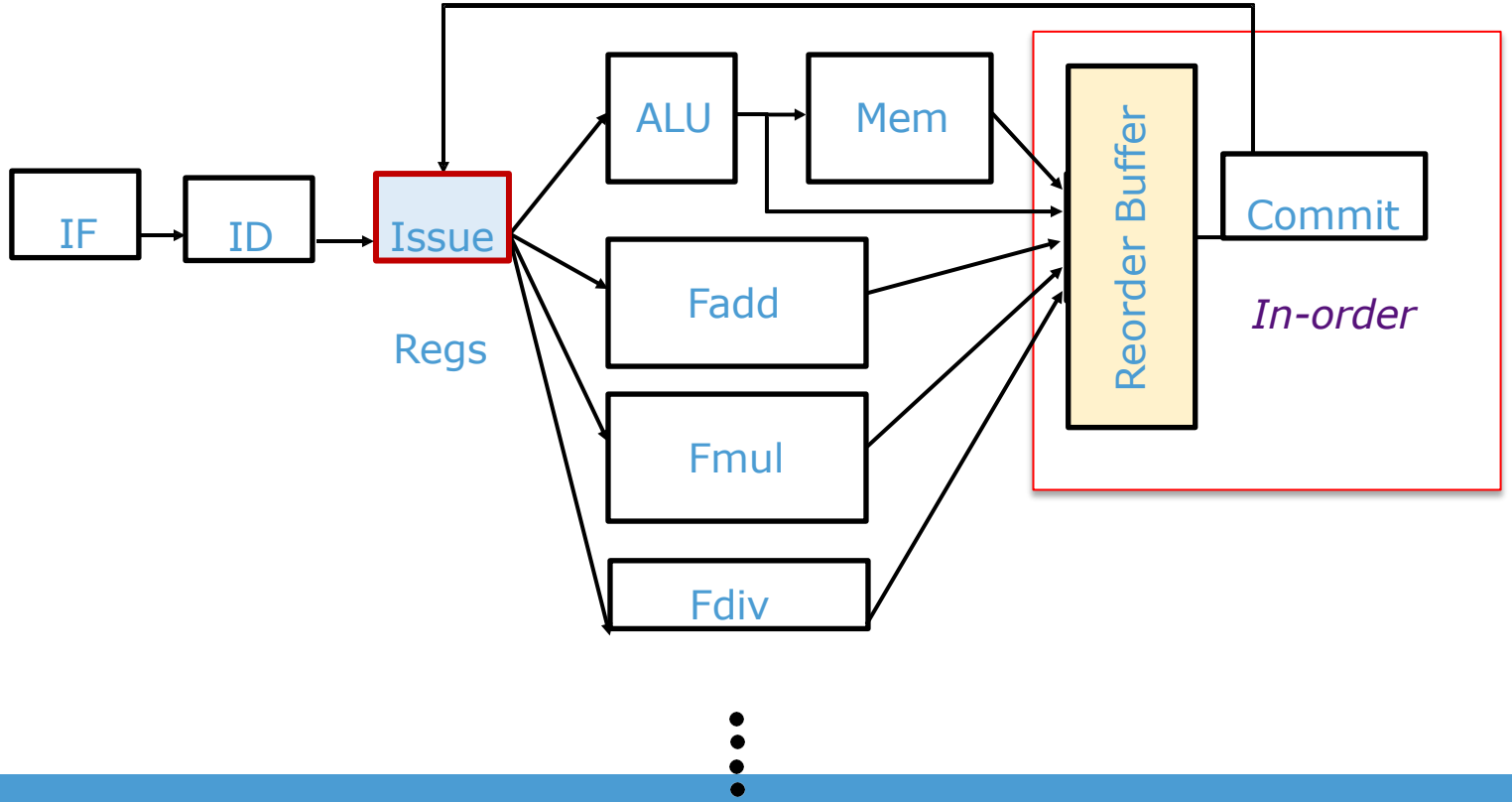
- Attack outcome: user space applications can read arbitrary kernel data

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



2nd line of code can transiently execute before the exception occurs!

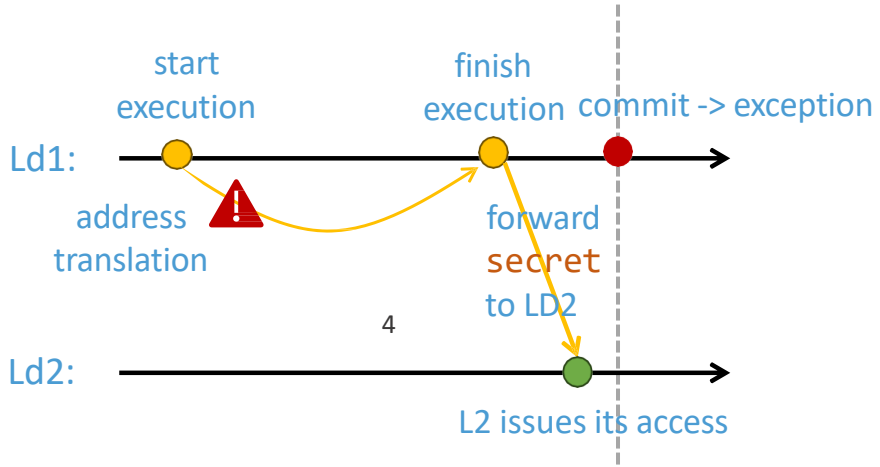
Technique #3: In-order Commit



Meltdown Timing

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

Ld2's request is sent
out before the instruction is squashed.



Meltdown w/ Flush+Reload

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

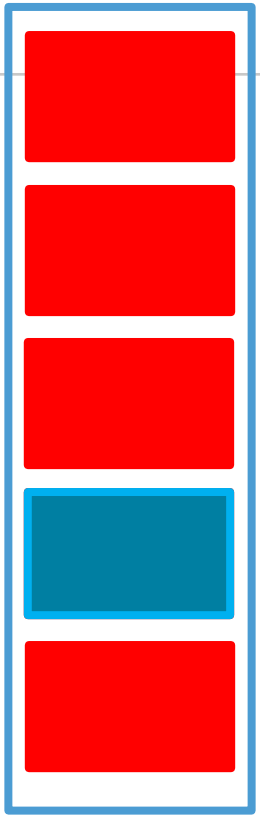
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

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

Flush+Reload

Attacker

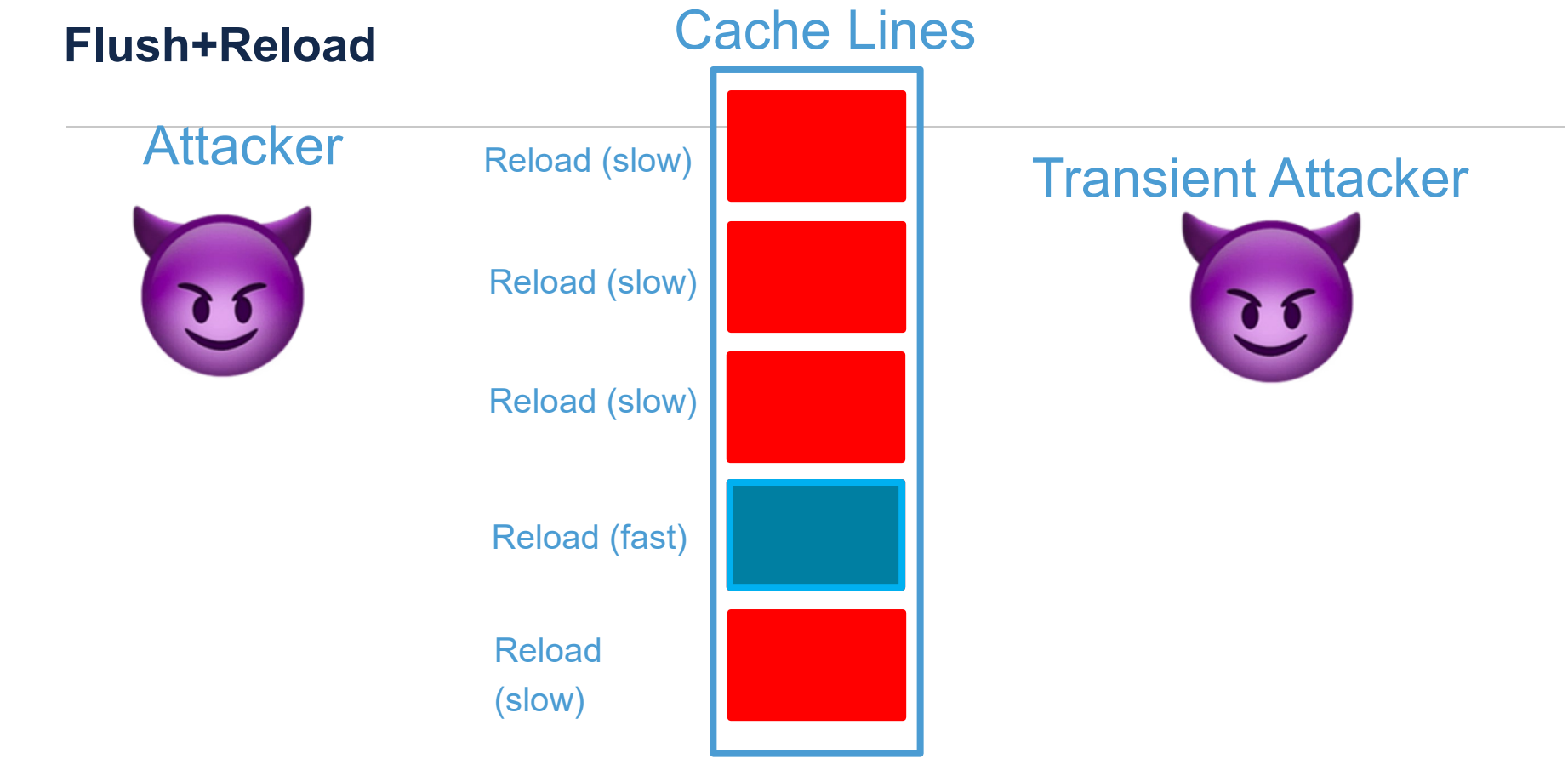
Cache Lines



Victim

Attacker learns secret byte=3

Flush+Reload



Attacker learns secret byte=3

Why did it take so long for Meltdown to be discovered?

Software

Hardware

SW optimization:
Map kernel address
in user space

Contract: Memory access goes through page permission check,
and permission violation raises exceptions

HW optimization: Speculation to
delay exception handling

Meltdown Followups

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

LVI

Speculative Execution

- Reduce impact of pipeline bubbles:
  - out-of-order execution
  - Execute each instruction as soon as its source operands are available

- How do we handle control hazards?
  - When next instruction to fetch is unknown

- Stall ?
  - Wait for the result to be available by freezing earlier pipeline stages
- branch speculation:
  - Predict both direction and target of branches and jumps
  - Guess a value and continue executing anyway
    - When actual value is available, two cases:
      - Guessed correctly: do nothing
      - Guessed incorrectly : squash & restart with correct value

Branch Prediction

- Naïve approach: PC+4
  - Assume not taken
- More advanced, predict two things:
  - Direction of a conditional branch (whether a branch is taken or not)
    - `blt r1, r2, <label>`

Idea: 1-bit predictor for loop

  - The target address of a branch
    - `jalr <reg>`
    - `ret`

Idea: memorizing branch source and destination pairs

An Example of Branch Prediction

- Consider the following the C code and its corresponding assembly codes

```
If (R1 == 2)
     R1 = 0;
else
     R1 = 2;

R2 = 3;
```

```
     cmp R1, 2
     jne L1
     mov R1, 0
     jmp L2
L1:
     mov R1, 2
L2:
     mov R2, 3
```

- Prediction will be basically a guess

Another Example of Branch Prediction

- Consider the following the loop and its corresponding assembly codes

```
for(R1=0;R1<10;R1++){
    *(R2+R1*4) = 0;

}
R3 = 3;
```

```
    mov R1, 0
L0:
    mov [R2+R1*4], 0
    add R1, R1, 1
    cmp R1, 10
    jl  L0
L1:
    mov R3, 3
```

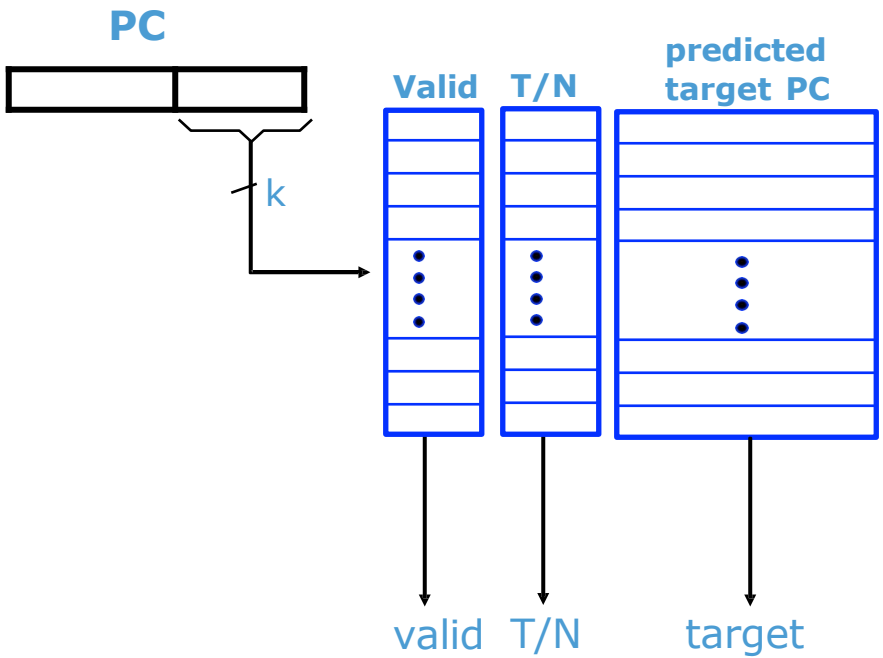
Branch predictor predicts if the `jl L0` instruction will be taken or not. It also predicts the actual memory address of label `L0`, as it can be encoded as PC-relative address.

Prediction for the first execution of this branch is basically guessing. But for the future executions of this branch, we can use past taken or non-taken history and past branch target.

Dynamic Hardware Branch Prediction

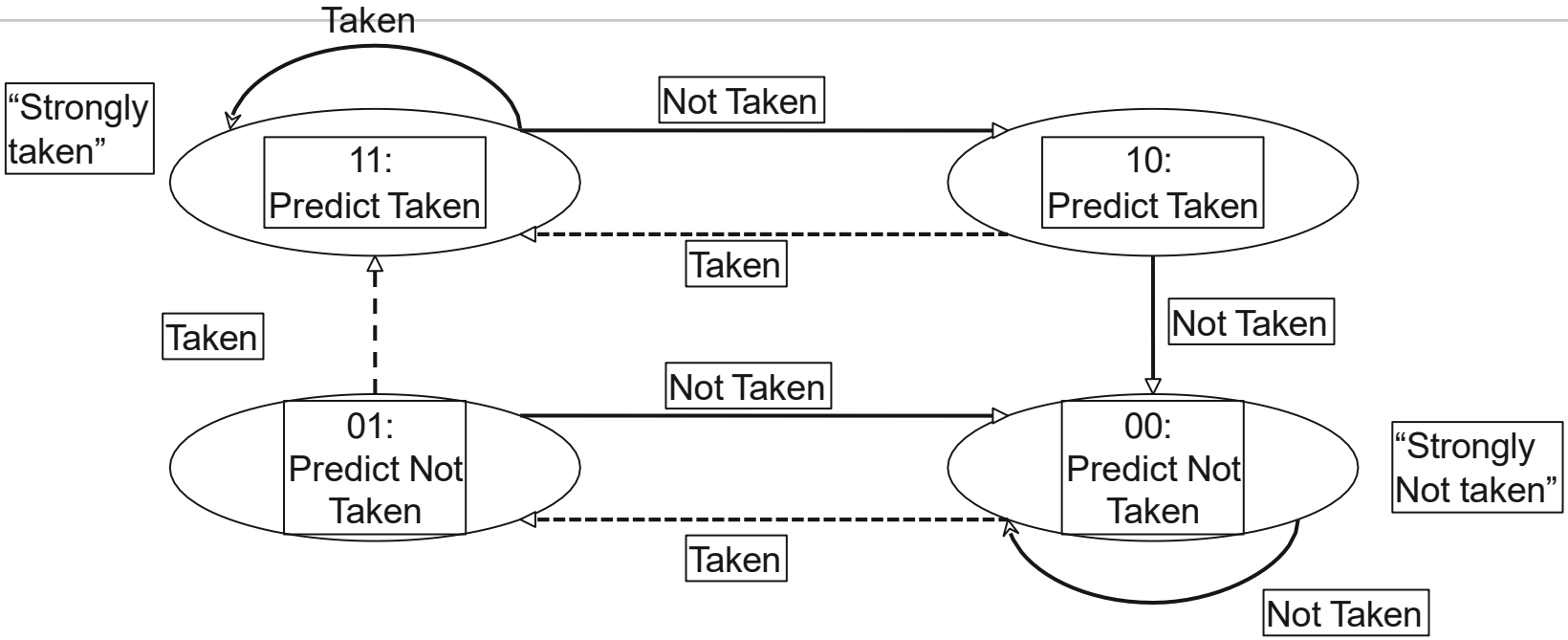
- Branch behavior is monitored during program execution
  - History data can influence prediction of future executions of the branch instruction
- Branches instruction execution has two tasks/predictions
  - Condition evaluation (taken or not-taken)
  - Target address calculation (where to go when taken)
- Target prediction also applies to unconditional branches
  - E.g. jmp %eax

A Simple Branch Predictor Unit (BPU)



- When branch instruction commits
  - Update the predictor
- In the fetch stage
  - Use the predictor to decide what address to fetch next
- Limited space?
  - Use selected bits in PC to index into the predictor

Two-bit Saturating Counters



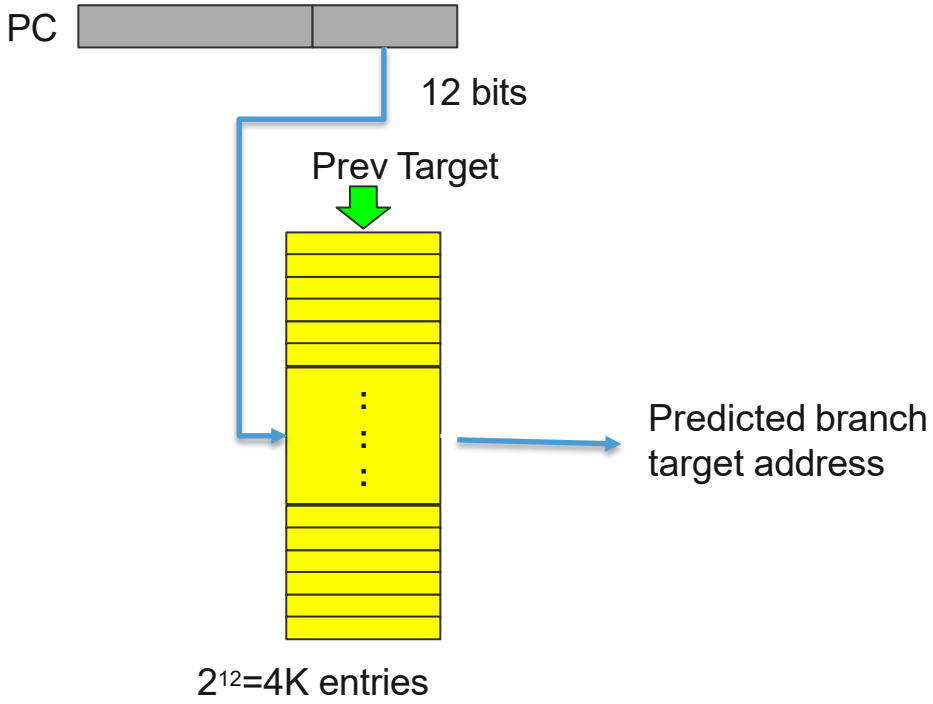
- 2-bit history means prediction must miss twice before change.
- N-bit predictors are possible, but after 2-bits not much benefit.

State	Prediction	Actual Branch Outcome
00	Not Taken	Taken
01	Not Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Not Taken

Branch Target Prediction

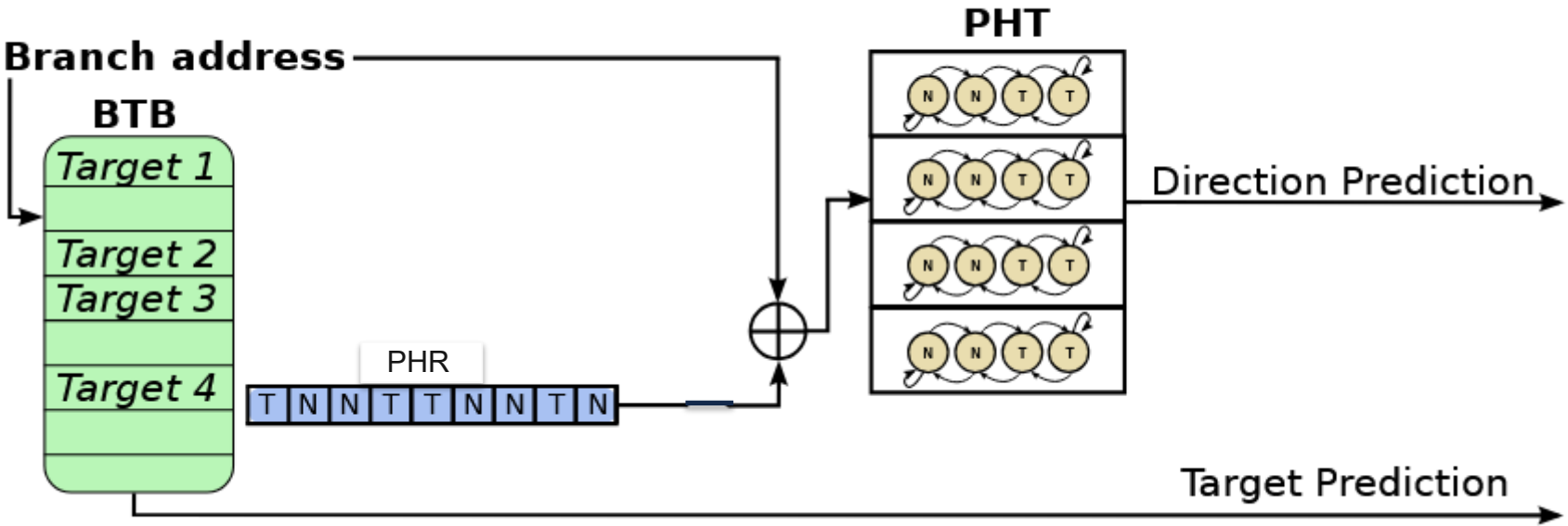
- Besides predicting branch taken or not, branch targets also needs to be predicted.
- Branch Target Buffer (BTB) is the storage used to store the branch target address from previous execution of the branch.
    - This is essentially a cached design.
    - If predicted taken, the cached target address is extracted as predicted branch target.

A Example of Branch Target Buffer

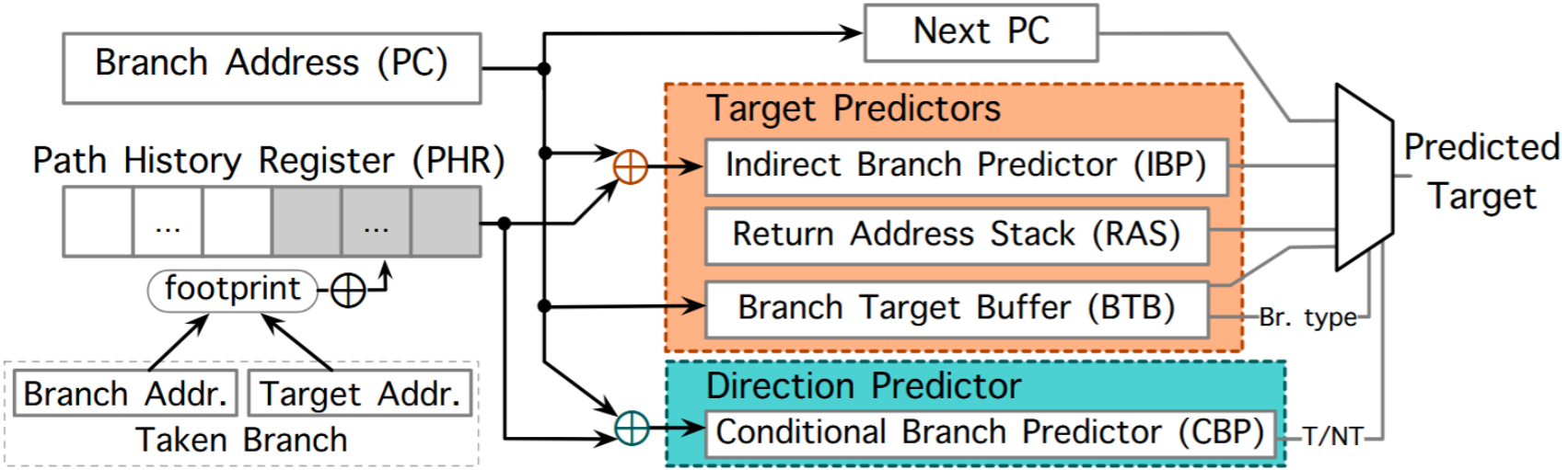


Pattern History Register (PHR)

- Records taken or “not-taken” for last n-branches
  - 194 on modern Intel machines
- Used in conjunction with PC to index into the PHT


Branch address
BTB
Target 1
Target 2
Target 3
Target 4
PHR
T N N T T N N T N
PHT
N N T T
Direction Prediction
Target Prediction

Branch Predictor on modern CPU



Speculative Execution and Security

- “Speculative execution is an optimization technique where a computer system performs some task that may not be needed or should not be executed.”
  - Branch prediction is a type of speculative execution.
- Speculative execution is extensive in modern processors since they are crucial for performance.
- It was until 2018 that we learned the security implications from speculative execution, by the Spectre and Meltdown vulnerabilities.

The Spectre Vulnerability

- Consider the following pseudocode:

```
if (do security check and pass){
    Access_sensitive_data()
}
```

- Normally, the sensitive data can only be accessed if the security check is successful. However, with speculative execution, sensitive data may be accessed before the we know if the security check is passed or not.
  - When the sensitive data is accessed and used as an index into the cache, a separate side-channel attack is used to recover the cache access pattern.

The Spectre Vulnerability cont'd

- Consider the following pseudocode:

```
if (do security check and pass){
    transmit_sensitive_data()
}
```

- The attacker can train the branch predictor to always predicted true for this if-branch.
  - Recall the branch predictor only uses the low bits of the PC.
  - The attacker can find another branch instruction with PC that has the same lower bits as this if-branch. The attacker can then train the branch predictor to predict "taken" by letting this 2nd branch instruction be taken many times

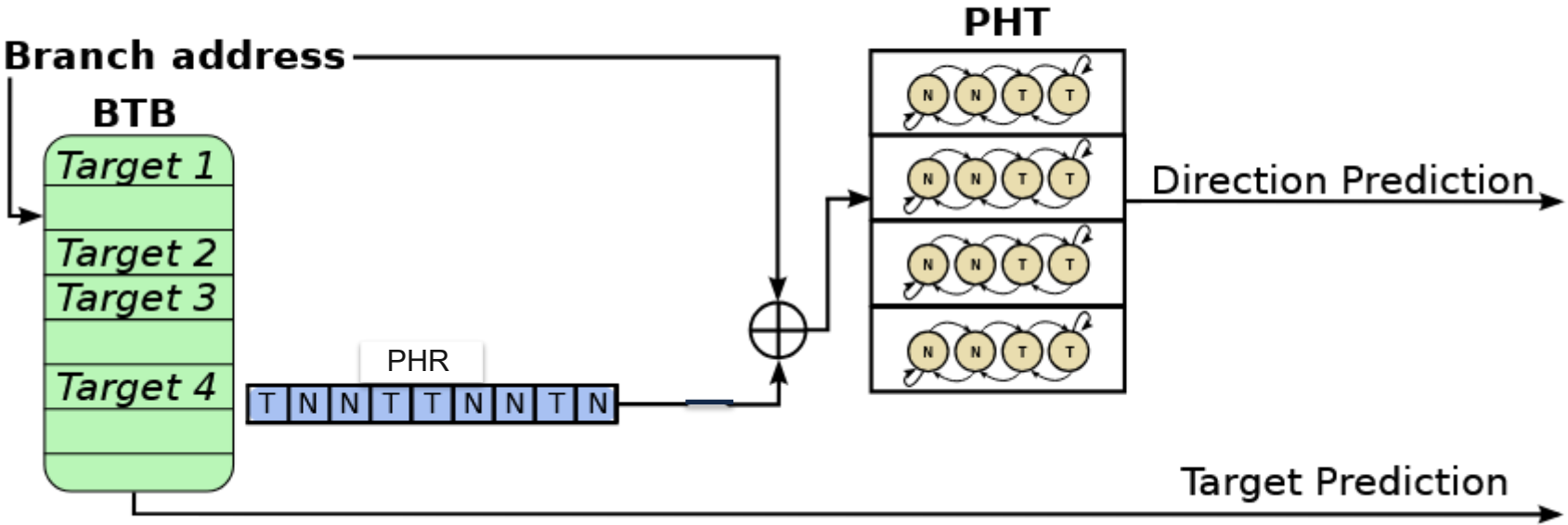
Spectre Variant 1 – Exploit Branch Predictor

- 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]
      }
```

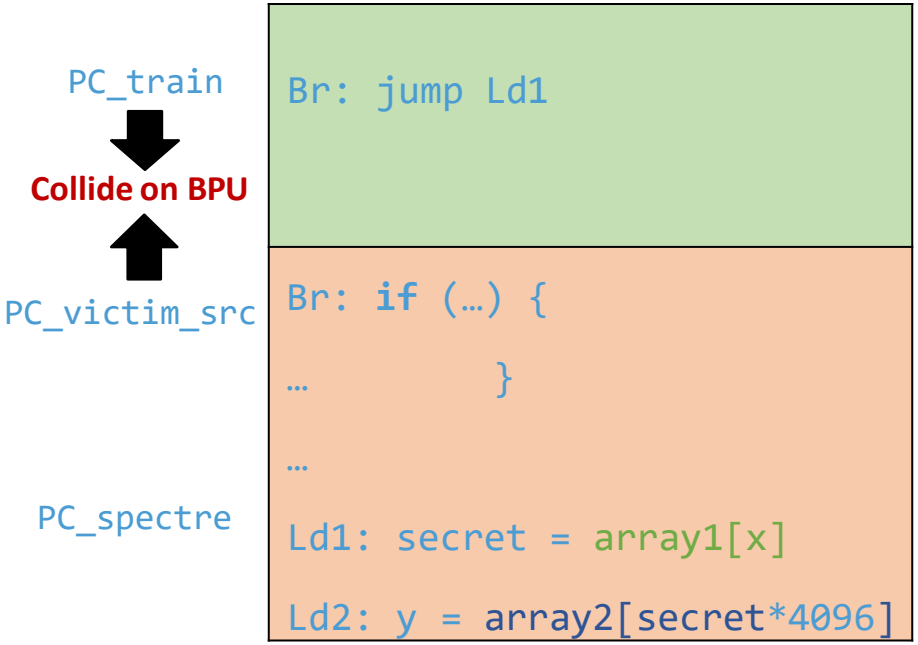
Always malicious?
No. It may be a benign misprediction.
We do not consider Spectre to be a bug.

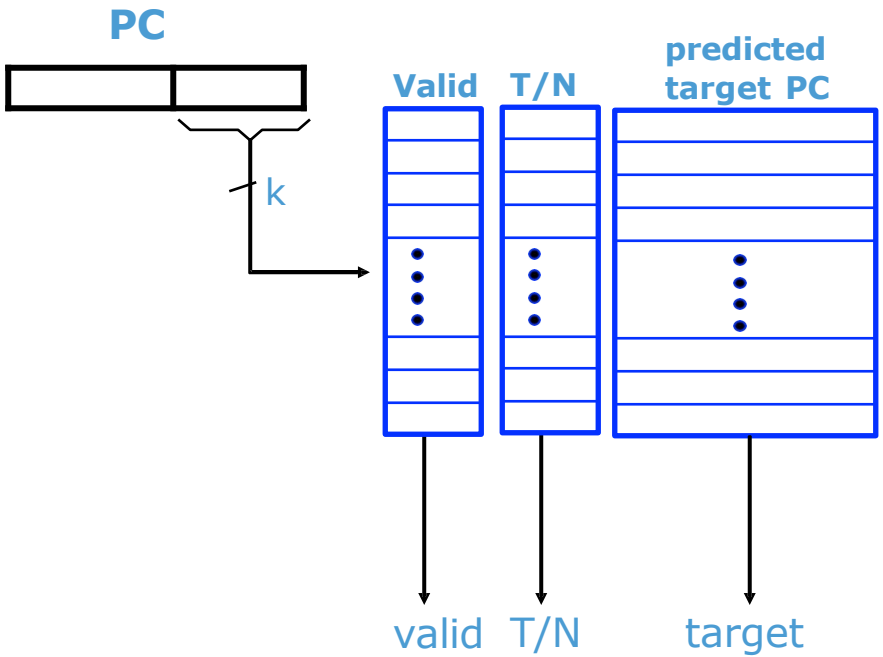
Attack to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of *array2* was fetched


Branch address
BTB
Target 1
Target 2
Target 3
Target 4
PHR
T N N T T N N T N
PHT
N N T T
Direction Prediction
Target Prediction

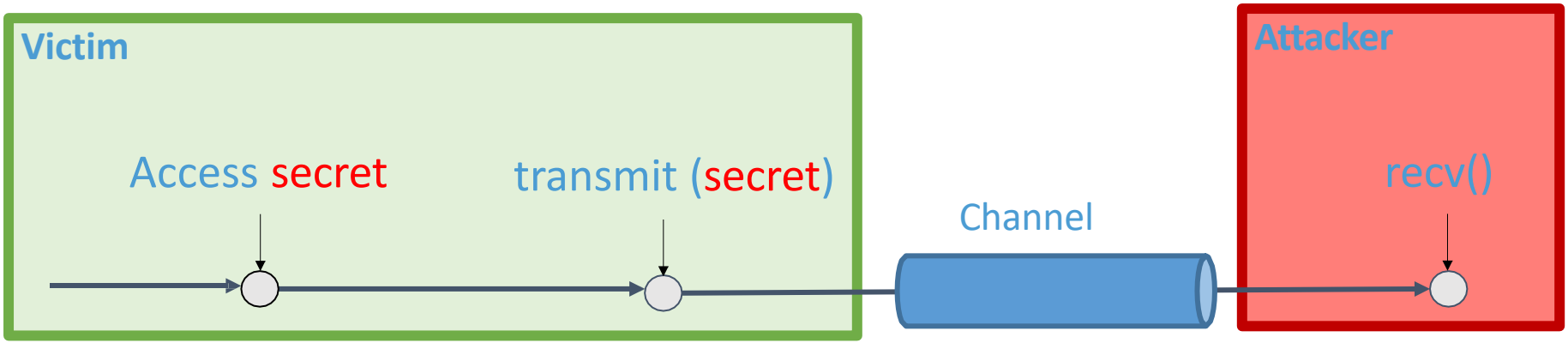
Spectre V2 – Speculative JOP

1. Insert <PC_train, PC_spectre>
2. Trigger PC_victim_src
3. Speculatively execute PC_spectre



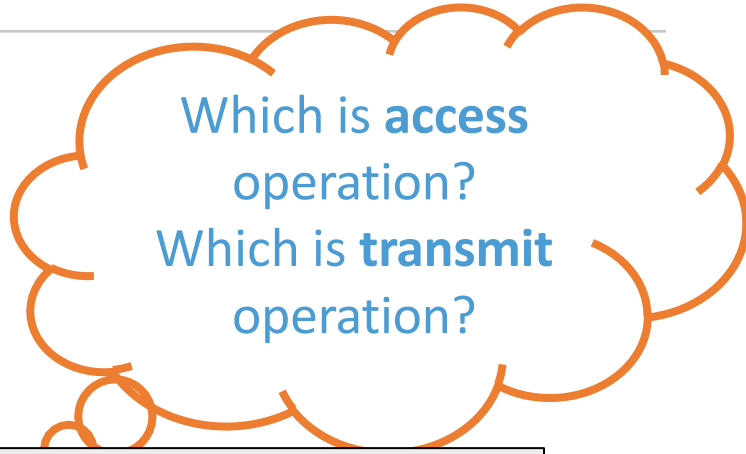


General Attack Schema



Apply the General Attack Scheme

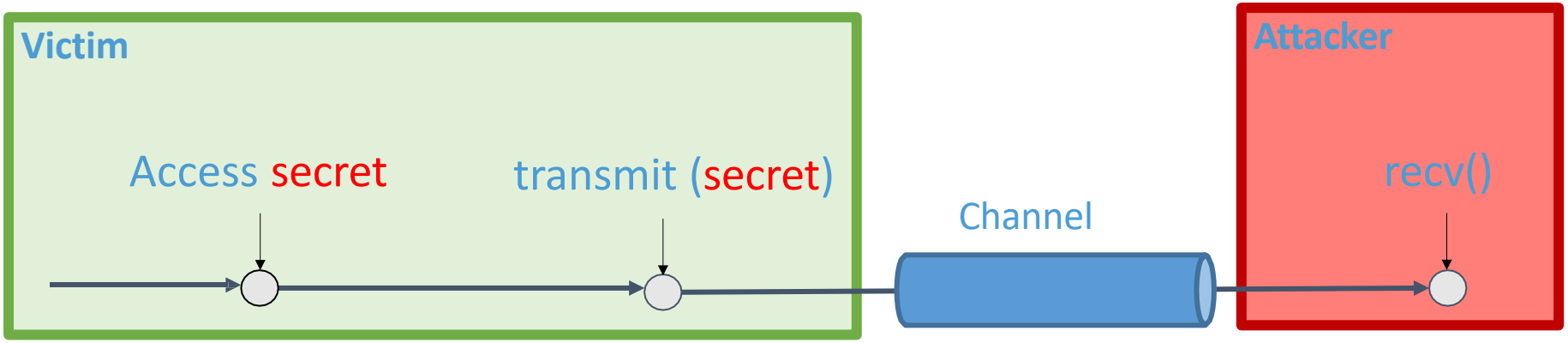
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```




```
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]
```

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