

# More Side Channel Defenses: A Cat-and-Mouse Game

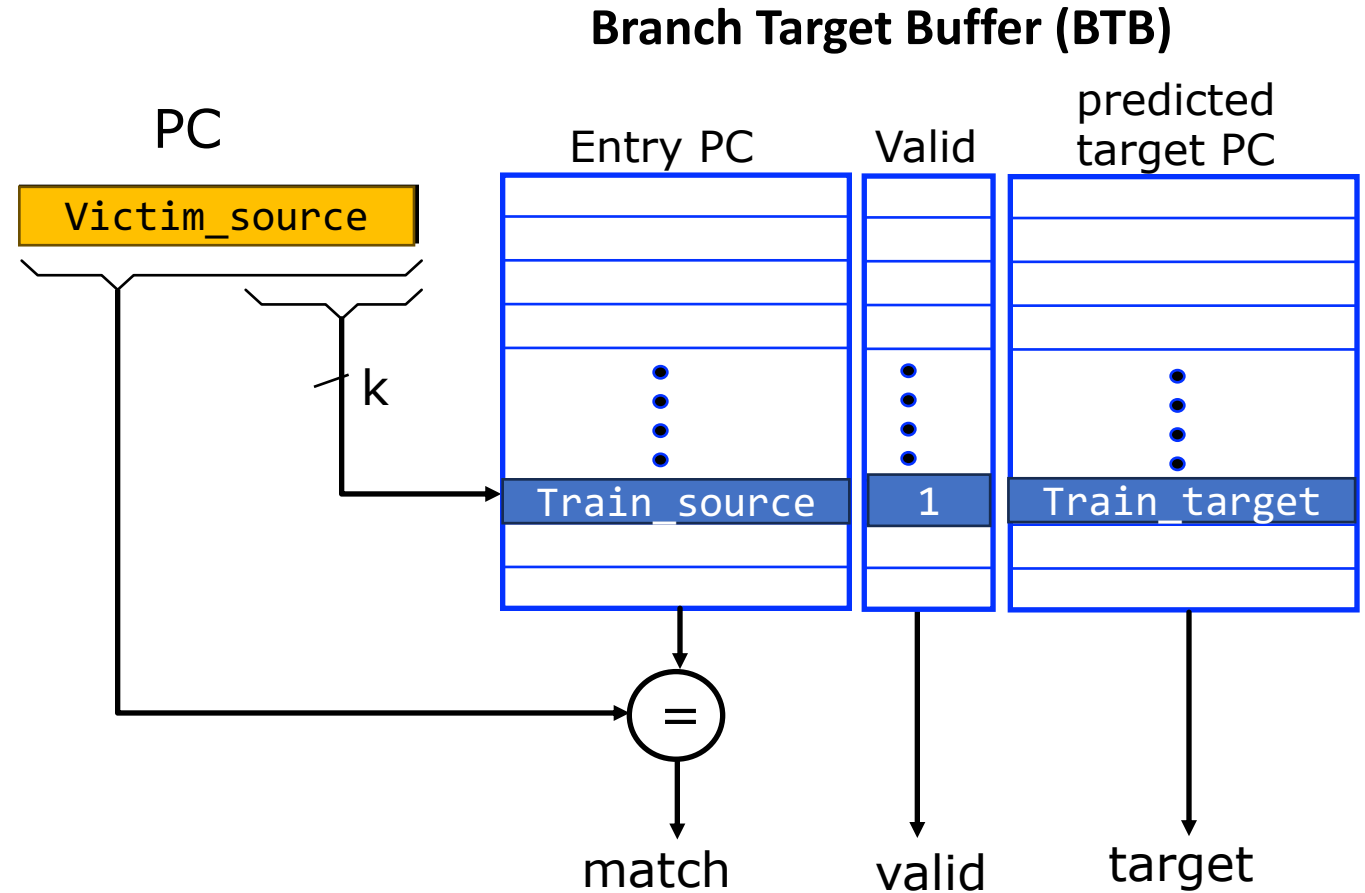
Mengjia Yan

Spring 2024



# Recall Spectre v2 (BTB Injection)

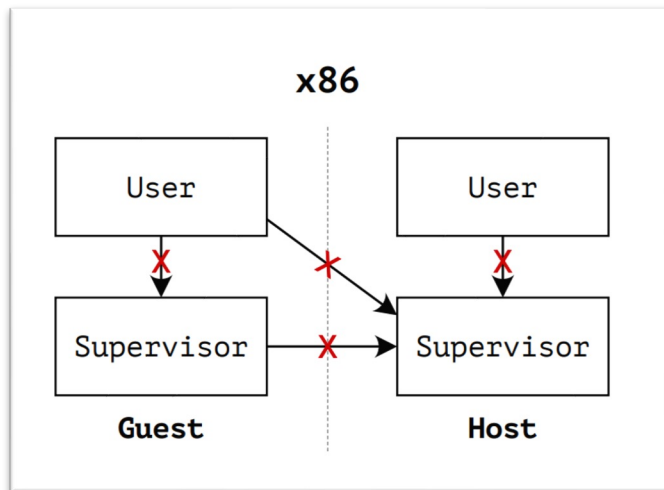
```
; Attacker code
Train_source:
    jmp train_target
...
Train_target:
    secret = array1[x]
    y = array2[secret*4096]
...
; ----CONTEXT SWITCH----
; Victim code
Victim_source:
    jmp rax
...
```



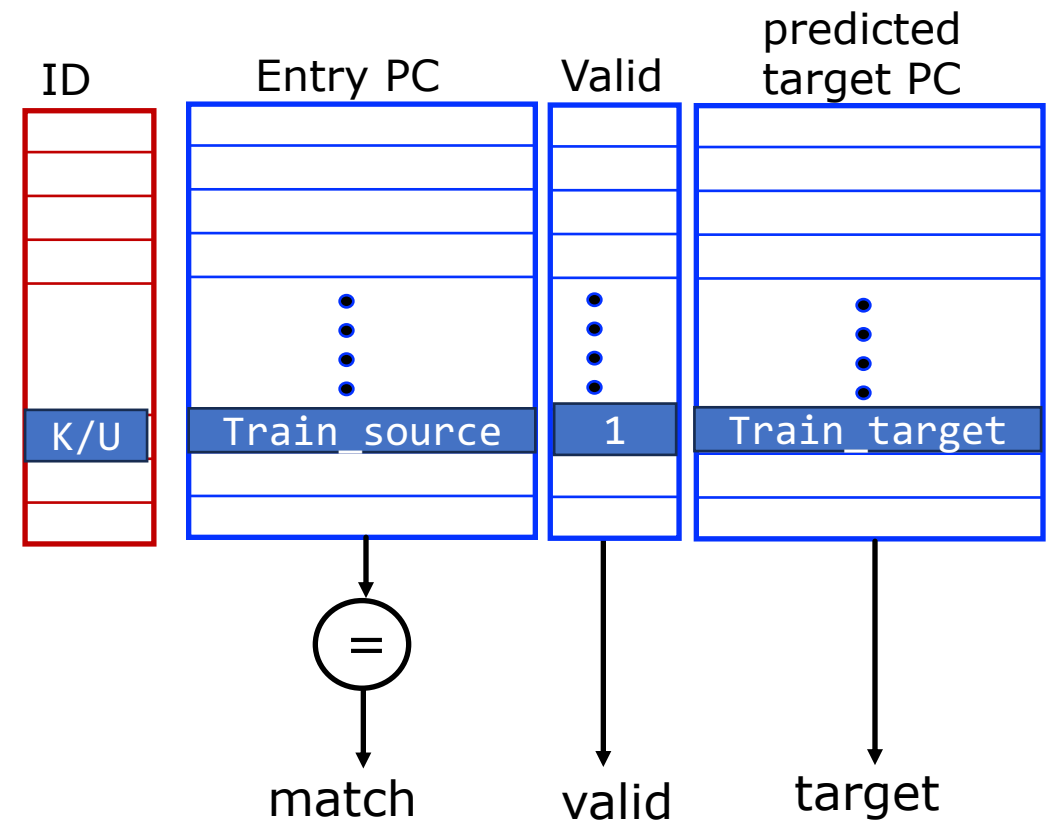
# Deployed Hardware Fixes: eIBRS

**eIBRS** stands for Enhanced Indirect Branch Restricted Speculation => Isolate BTB entries across privilege levels.

“x” indicates which branch injection attack vectors should be prevented.



## Branch Target Buffer (BTB)



# Examine the Security Property

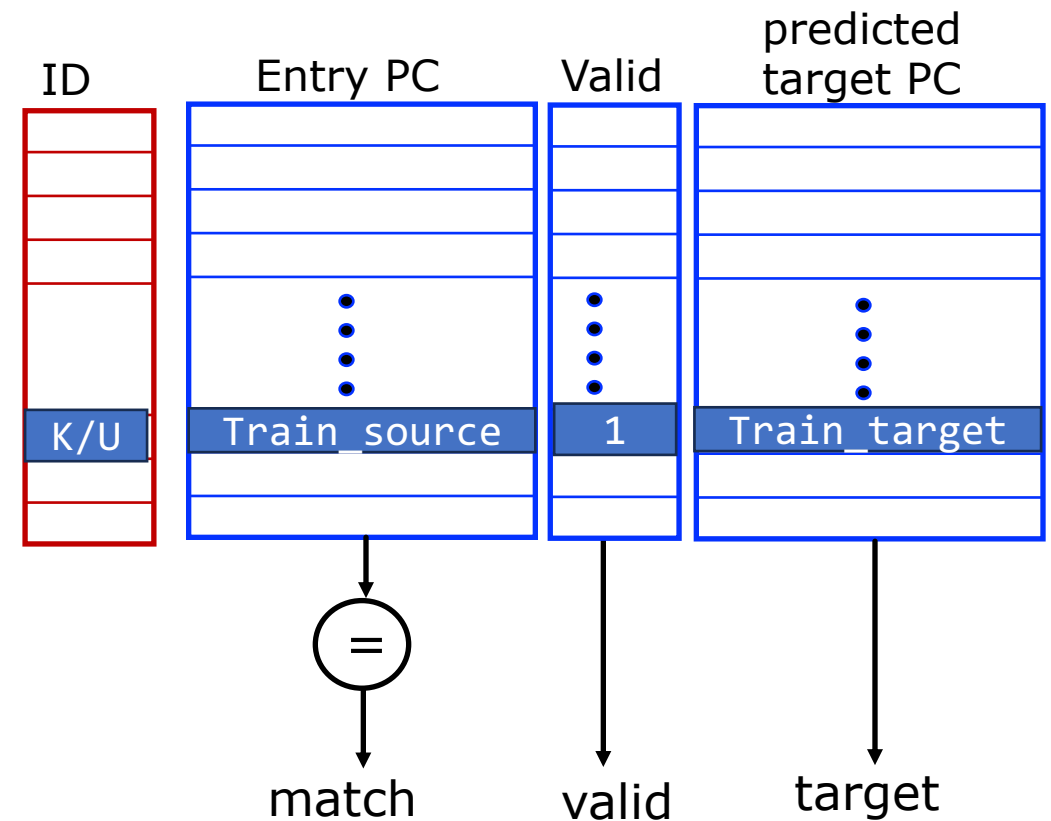
What do we mean by isolation?



- Property #1:
  - Kernel-space indirect branches **do not use** branch target inserted by userspace code.
- Property #2 (non-interference):
  - Userspace code **does not interfere** with Kernel-space indirect branch predictions.

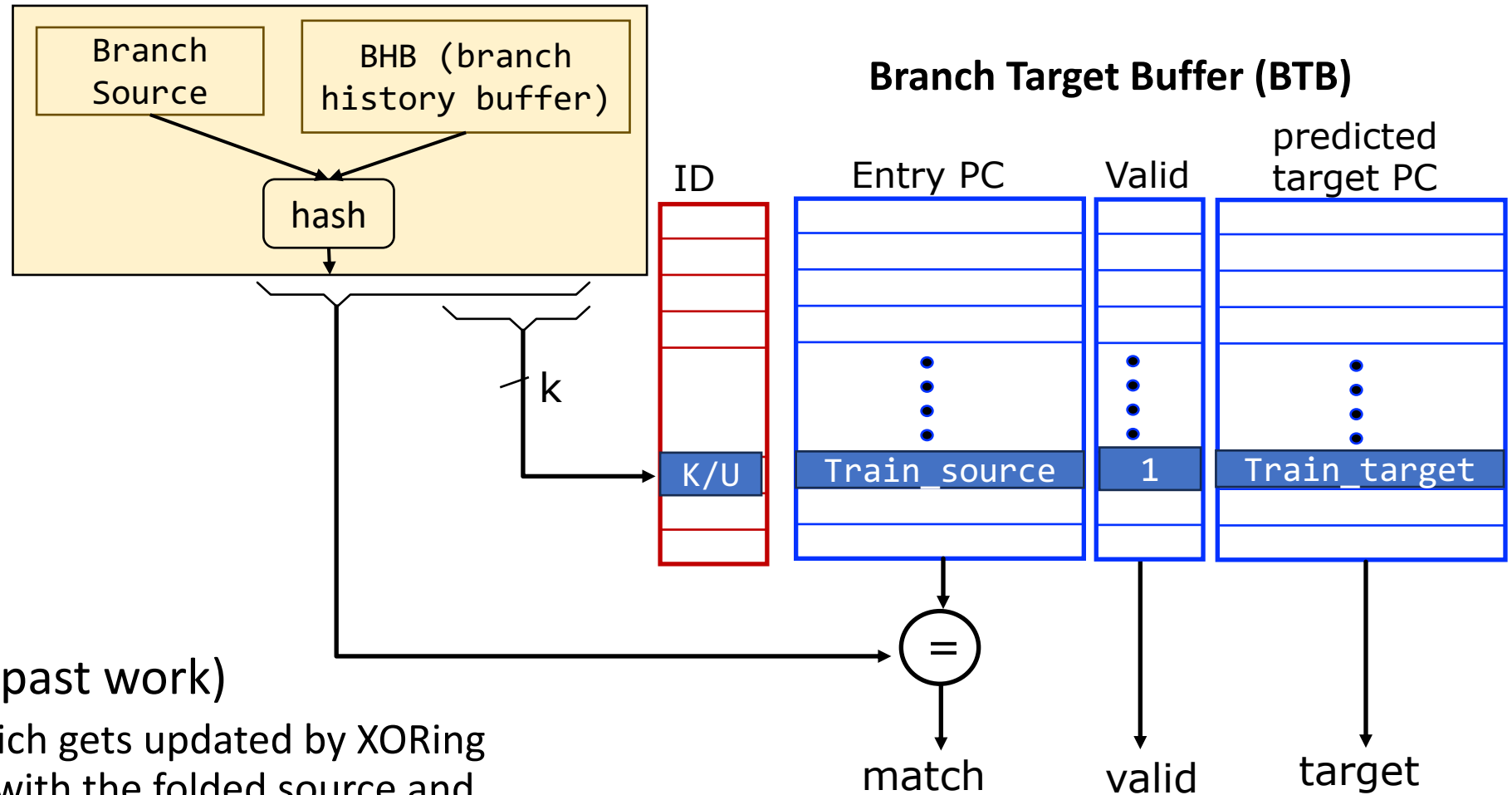
Does eBRS achieve property #2? If not, counterexamples?

## Branch Target Buffer (BTB)



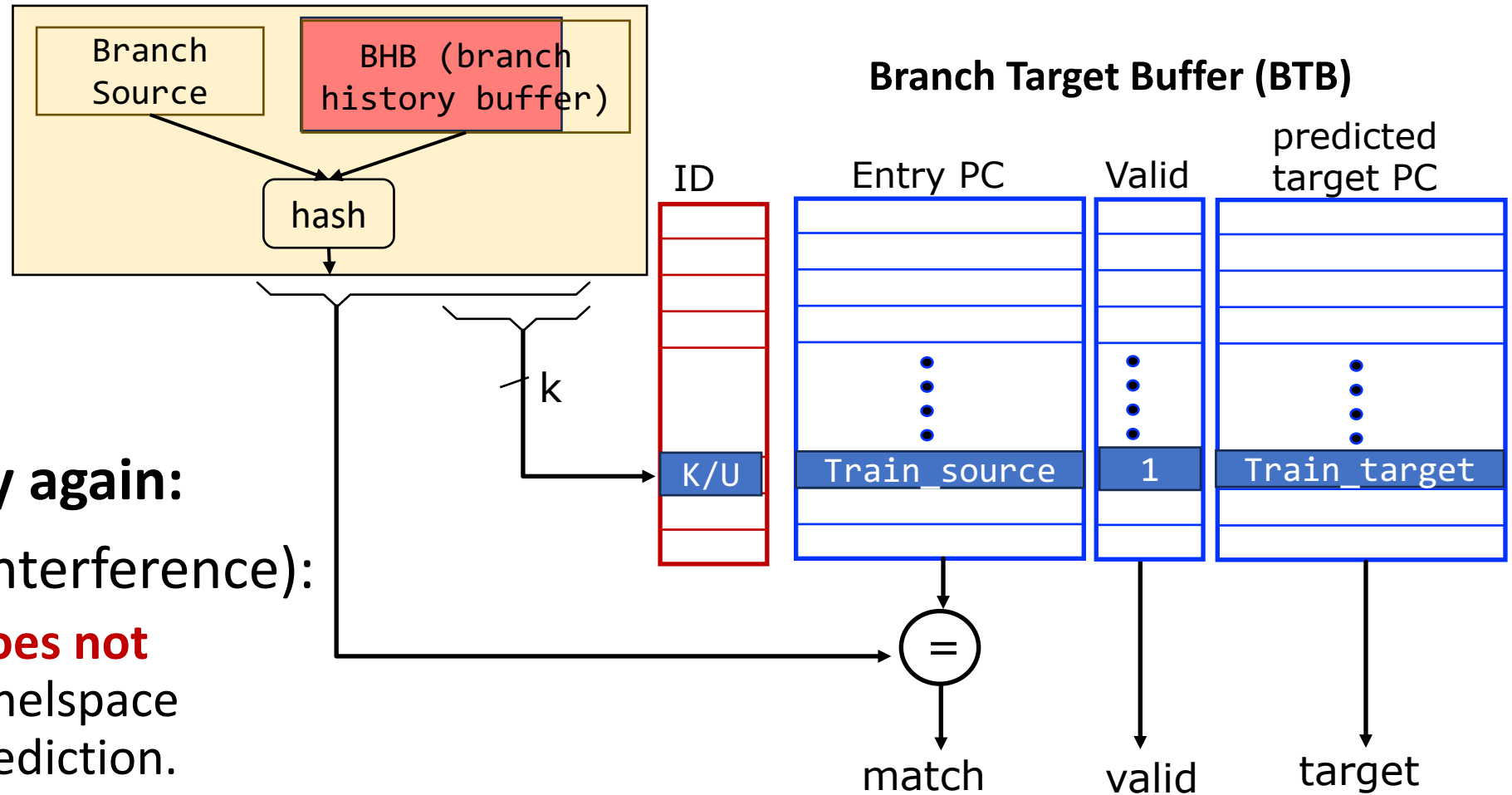
Same-mode misprediction

# How Does BTB Actually Work?



- BHB: (according to past work)
  - A shift register which gets updated by XORing its right most bits with the folded source and destination address of a taken branch

# Branch History Injection



## Look at the property again:

- Property #2 (non-interference):
  - Userspace code **does not interfere** with Kernel-space indirect branch prediction.

# A Detour: Consequences due to Retpoline

<b>Before retpoline</b>	<code>jmp *%rax</code>
<b>After retpoline</b>	<pre>call set_up_target (1) capture_spec: (4)     pause     lfence     jmp capture_spec set_up_target:     mov %rax, (%rsp) (2)     ret (3)</pre>

**Listing 3** Linux implementation for the Spectre v2 mitigation before version 5.14 on Intel processors depending on eIBRS hardware support. The shown example is taken from the indirect jump in charge to execute the correct syscall handler stored in the `sys_call_table`.

```
1 do_syscall_64:
2     ;...
3     mov     rax, [sys_call_table + rax*8]
4     call   __x86_indirect_thunk_rax
```

```
1 ;with eIBRS support
2 __x86_indirect_thunk_rax:
3     jmp     rax
```

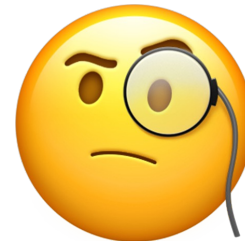
Perfect victim branch for in-place BTB attack

```
1 ;without eIBRS support (retpoline)
2 __x86_indirect_thunk_rax:
3     call   B
4 A:     pause
5     lfence
6     jmp     A
7 B:     mov     [rsp], rax
8     ret
```

<https://support.google.com/faqs/answer/7625886>

# Takeaway Messages

- Goal: communicate security property achieved by hardware defenses
  - The bad example: eIBRS -> unclear what exactly isolation mean...
- Alternative approaches:
  - Approach 1: Show SW people all the HW implementation details
  - Approach 2: define new SW-HW contracts





# SW-HW Contracts for Secure Speculation



# Attempt #1: Make Speculation Invisible

- Idea: make speculative executed instructions' microarchitecture effects invisible by the attacker
- Examine program examples

```
if (false)
  sec = ld x
  dummy = ld sec
```

```
sec = ld x
if (false)
  dummy = ld sec
```

```
sec = ld x
dummy = ld sec
if (false)
  .....
```

Secure if using  
invisible speculation?

Do they follow  
constant-time programming?

# Speculative Non-interference

- Some notations
  - $P$ : a deterministic program
  - $M_{pub}$ : public memory and inputs
  - $M_{sec}$ : secret memory and inputs
  - $O$ : microarchitecture observation (traces)
- Property:
  - **if** the SW does not leak under the constant-time programming model
  - **then** the HW should ensure no more secrets leaked under speculation

$\forall P, M_{pub}, M_{sec}, M'_{sec},$

IF  $O_{seq}(P, M_{pub}, M_{sec}) = O_{seq}(P, M_{pub}, M'_{sec})$

THEN  $O_{spec}(P, M_{pub}, M_{sec}) = O_{spec}(P, M_{pub}, M'_{sec})$

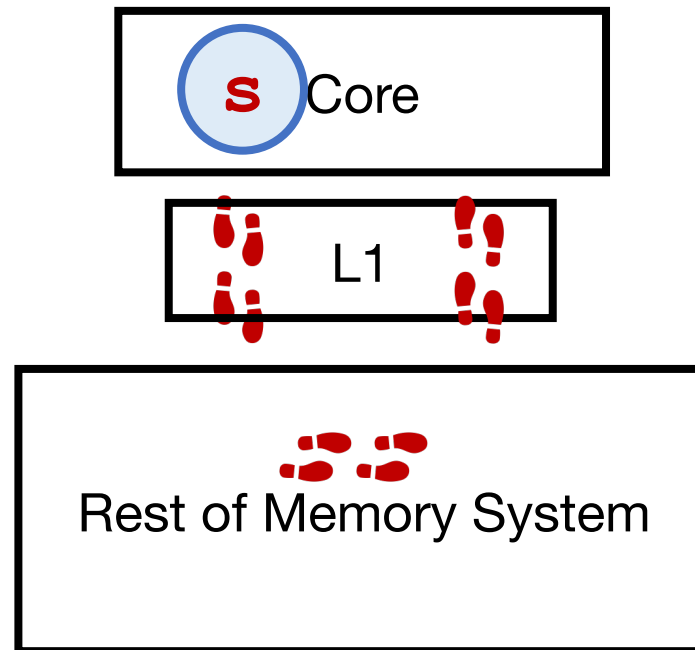
Execute program **sequentially**,  
monitor memory addresses.

Execute program **speculatively**,  
monitor memory addresses.

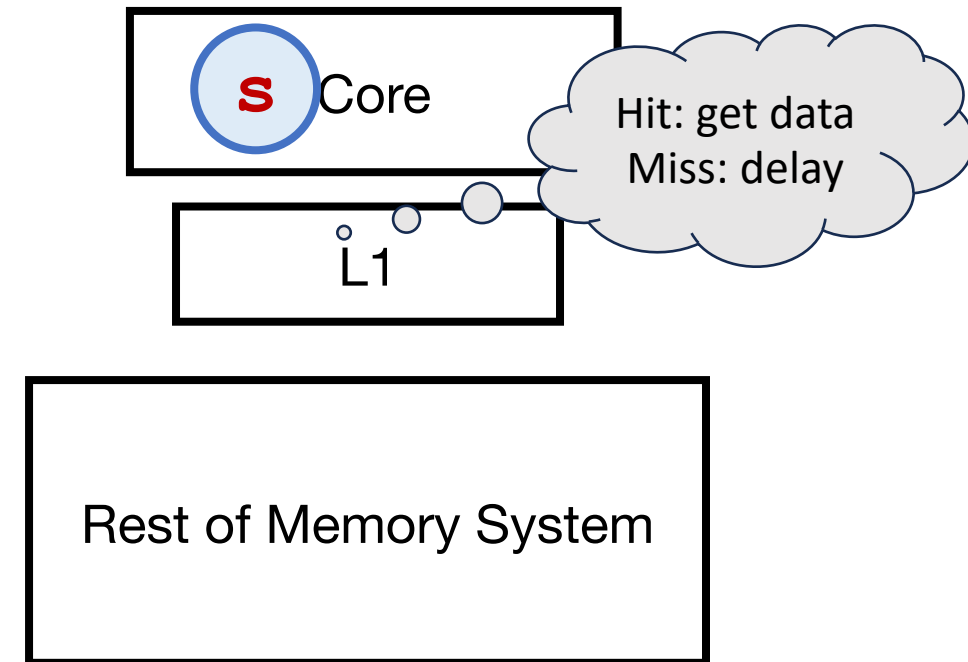
# Scheme #1: DoM

```
sec = ld x  
if (false)  
  dummy = ld sec
```

Insecure Baseline



Delay-on-Miss

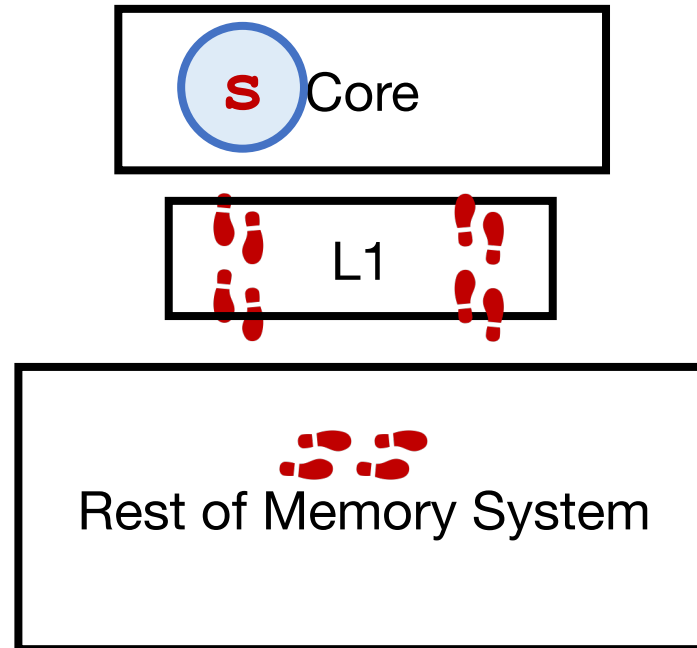


# Scheme #2: Invisible Speculation

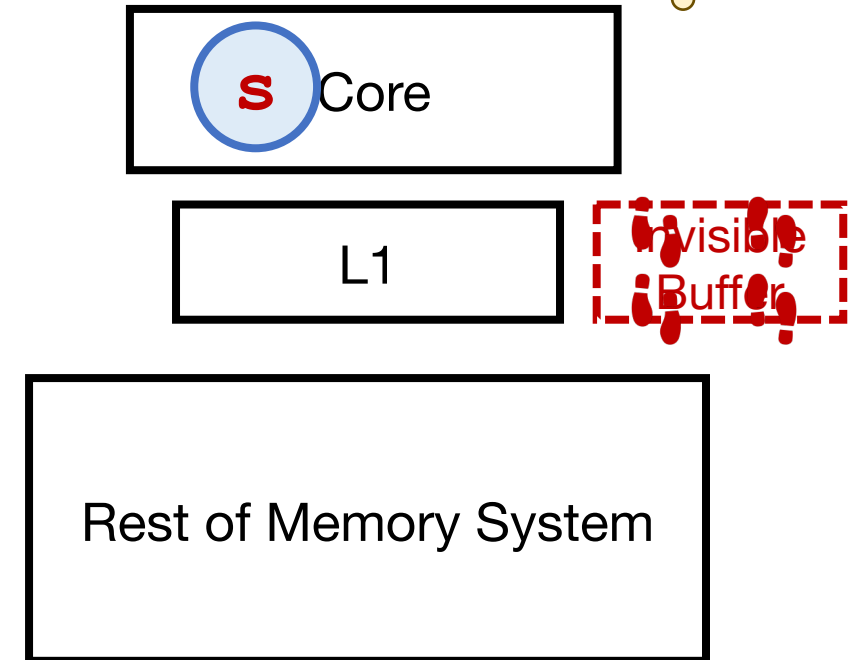
Any problem?  
Do we really achieve  
speculative non-  
interference here?

```
sec = ld x  
if (false)  
  dummy = ld sec
```

Insecure Baseline



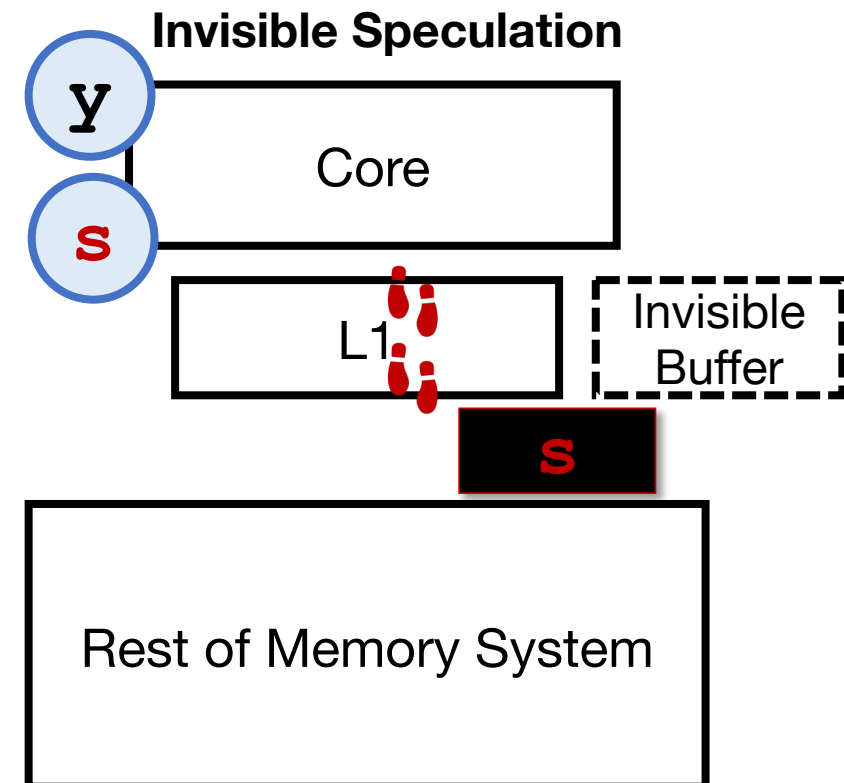
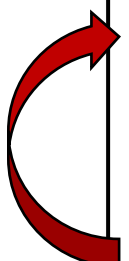
Invisible Speculation



# Speculative Interference Attack

- **Younger** speculative loads interfere with **older** bound-to-commit loads.
- Many other contention structures: non-pipelined ALU, cache port, bank contention, network-on-chip, etc.

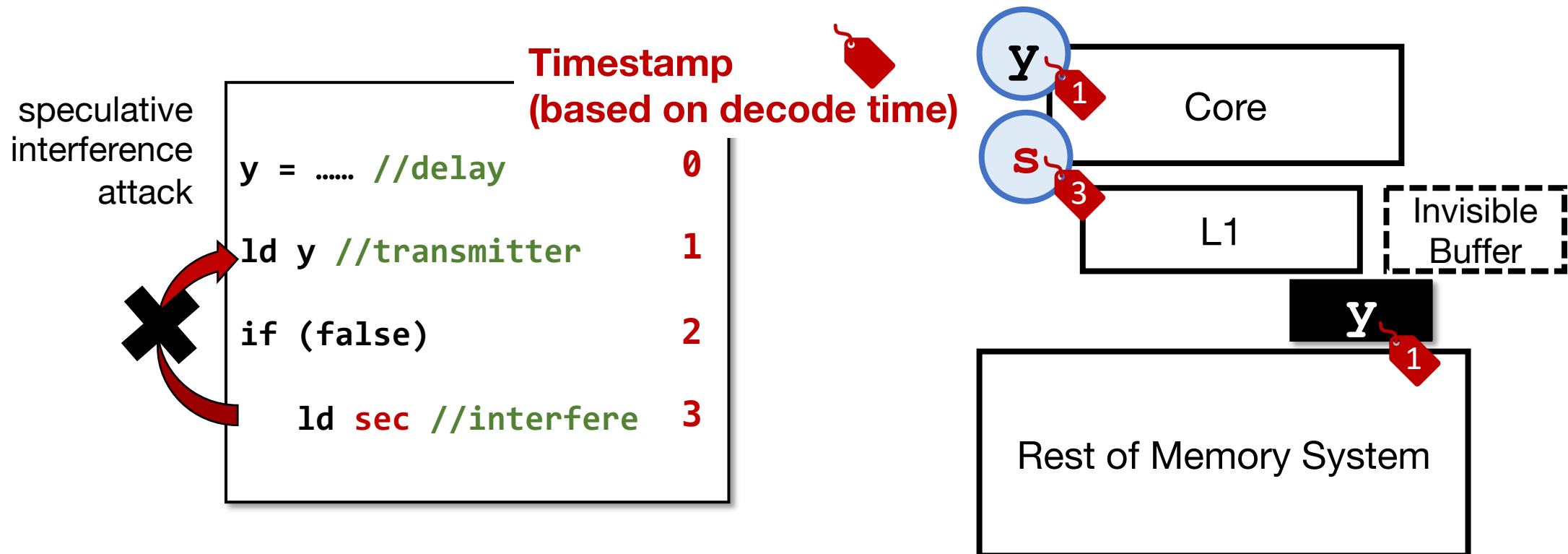
```
y = ..... //delay
ld y //transmitter
if (false)
    ld sec //interfere
```



# GhostMinion

#1: Invisible Speculation

#2: Prioritize Older Instructions through Timestamps



# New Attack Variant

GhostMinion prioritizes **smaller** timestamps

Timestamp  
(based on decode time)

<code>y = ..... // delay</code>	<code>0</code>
<code>ld y // bound-to-commit</code>	<code>1</code>
<code>if (false)</code>	<code>2</code>
<code>ld sec // transient</code>	<code>3</code>

**Older** (pointing to line 1)

**Younger** (pointing to line 3)

**X** (crossing out the if statement)

Original speculative interference attack

Timestamp  
(based on decode time)

<code>if (true)</code>	<code>0</code>
<code>ld y // bound-to-commit</code>	<code>2</code>
<code>else</code>	<code>No Program Order</code>
<code>ld sec // transient</code>	<code>1</code>

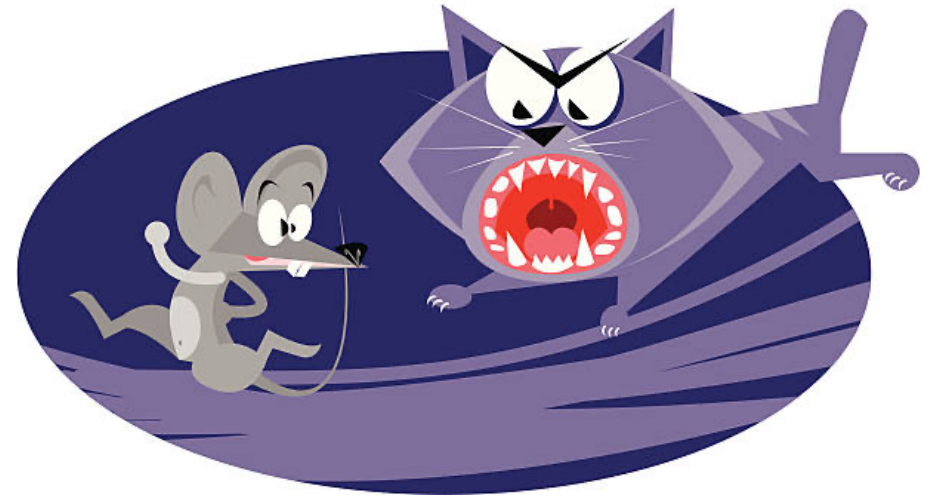
Black arrow pointing to line 0

Red arrow pointing from line 2 to line 1

New attack variant



# Summary: The Cat-and-Mouse Game



Need tools for automatically discovering vulnerabilities

# More Contracts



# Attempt #2: Relax the Security Property

- Idea: only protect speculatively loaded data

Spectre v1

```
if (false)
  sec = ld x
  dummy = ld sec
```

```
sec = ld x
if (false)
  dummy = ld sec
```

```
sec = ld x
dummy = ld sec
if (false)
  .....
```

Secure if using invisible speculation?



SW needs to follow constant-time programming

Secure if only protecting speculatively loaded data?



Software sandboxing



# STT and NDA Designs

- Draw on the board

# Understand the Property/Contract

**Speculative non-interference:** HW that can protect constant-time programs.

$$\forall P, M_{pub}, M_{sec}, M'_{sec},$$
$$\text{IF } O_{seq}(P, M_{pub}, M_{sec}) = O_{seq}(P, M_{pub}, M'_{sec})$$
$$\text{THEN } O_{spec}(P, M_{pub}, M_{sec}) = O_{spec}(P, M_{pub}, M'_{sec})$$

Execute program **sequentially**,

**Monitor architecture registers**

Execute program **speculatively**,  
monitor memory addresses.

Can also be used to describe the case for protecting software sandboxing...



# Summary of SW-HW Contracts

$\forall P, M_{pub}, M_{sec}, M'_{sec},$

IF  $O_{seq}(P, M_{pub}, M_{sec}) = O_{seq}(P, M_{pub}, M'_{sec})$

THEN  $O_{spec}(P, M_{pub}, M_{sec}) = O_{spec}(P, M_{pub}, M'_{sec})$

Describe what SW needs to achieve

Describe what HW needs to achieve for **only** the SW that satisfies the IF statement

- The payoff: we can check security properties for SW and HW independently
- Ongoing research: How to check and design according to these properties?

# Next: Paper Discussion

For presenters: 12 min per presentation. **If you run out of time, you will be interrupted and end up not finishing your presentation.**

For the rest: please come to the class on time and participate in the Q&A.

