Side Channel Mitigations

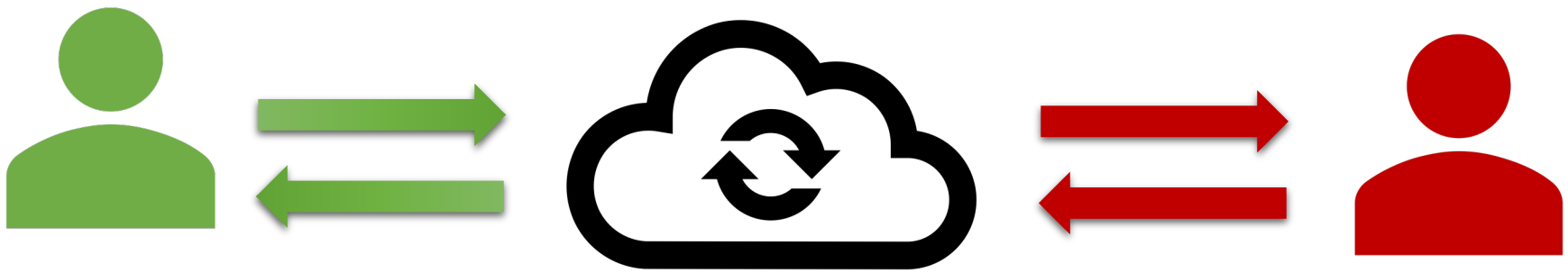
Mengjia Yan

Spring 2022

Outline

- **Non-interference**: a general security property
- Verify Non-interference for Side Channels and Transient Execution
- Hardware and Software Contract

Non-Interference Example

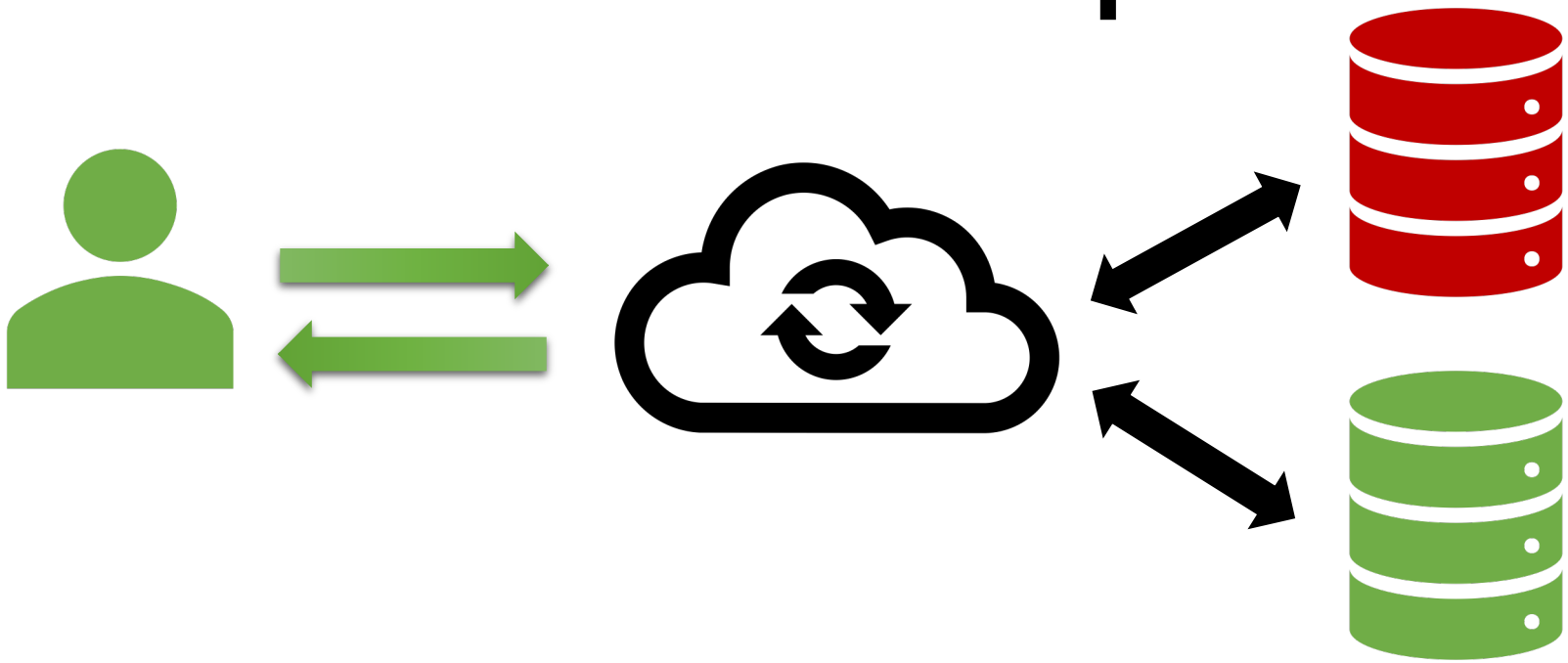


Low
(low sensitivity,
not highly classified)

High
(high sensitivity,
not seen by uncleared
users)

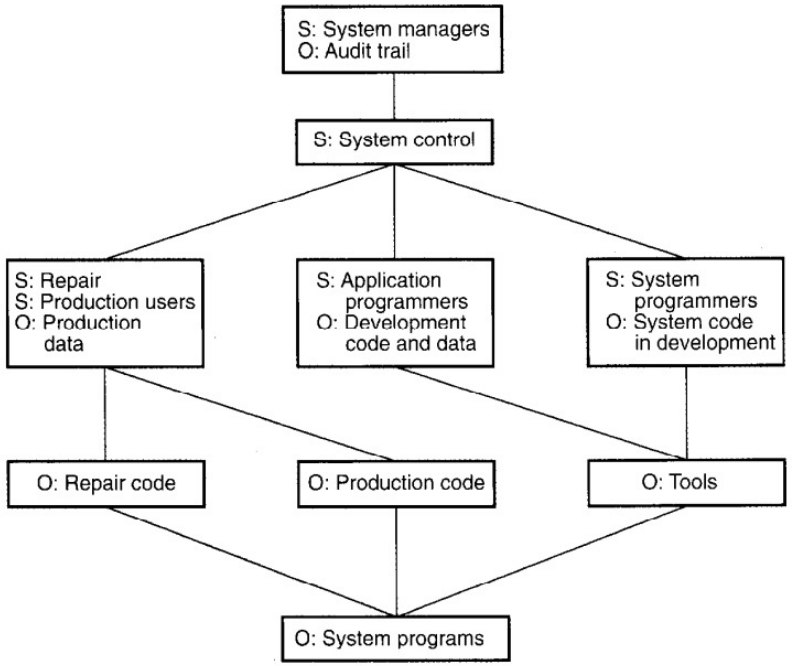
- Intuitively: not affecting
- Any sequence of **low** inputs will produce the same **low** outputs, regardless of what the **high** level inputs are.

Non-Interference Example



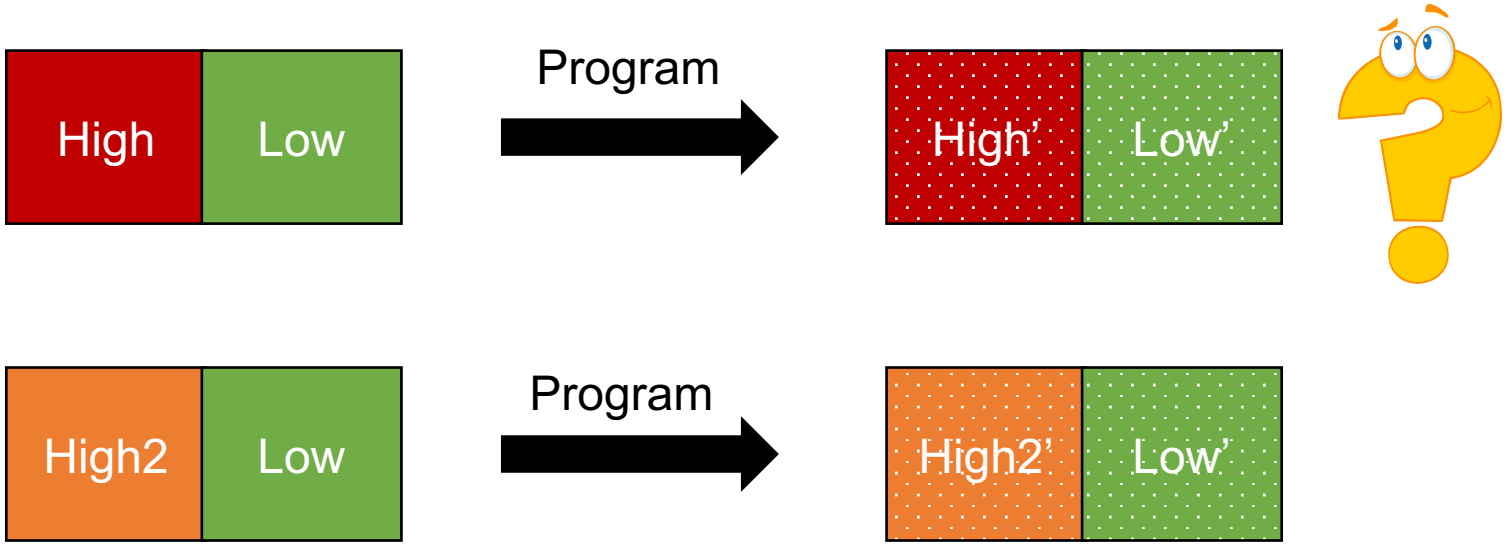
Use Cases of Non-interference

- Confidentiality: e.g., process isolation
  - My memory -> Confidentiality of High state
  - Other programs' memory -> Low
- Integrity: e.g., control-flow hijacking
  - My memory -> Integrity of Low State
  - Attacker controlled input -> High
- Swirl example
- Expand High-Low to Lattice



Lattice-Based Access Control Models; Ravi S. Sandhu; 1993

Non-Interference Formulation



$$\forall\, S1, S2, \qquad (S, P) \rightarrow S'$$

$$\text{if} \quad S1_L = S2_L$$

$$\text{then} \quad S1'_L = S2_L{}'$$

- Formulate the property as state-machine transition.
- Looking at a single-trace is ineffective

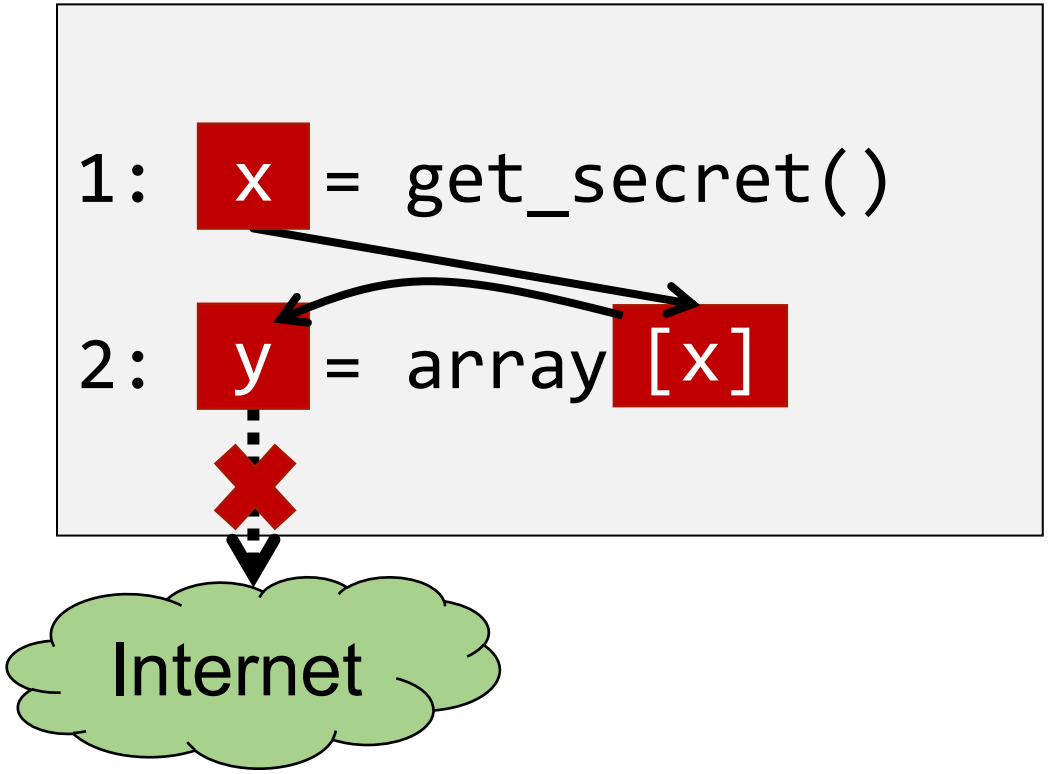
Generality of Non-interference

- Conventionally: ISA emulation for software analysis/testing

- Can also be used for hardware security design
  - Micro-architecture: state includes caches, buffers, buses, etc.
  - Circuit level: flip-flop

Taint Analysis (also Taint Tracking)

- Goal: verify non-interference property
- Analogy
- Components:
  - Source of taint (high state)
  - Taint propagation
  - Taint check (no taint on low state)



Explicit and Implicit Information Flow

```
1: x = secret;
2: y = x;
3: z = array[y];
```

Control-Flow

```
1: x = secret;
2: if (x == 0) {
3:     y = 1;
4: } else {
5:     z = 2;
6: }
```

Address Alias

```
1: x = secret;
2: array[0] = x;
3: z = array[y];
```

Taint Analysis Methods

- Dynamic: run-time check
  - Detect non-interference violation on-the-fly for a given input

- Static: compiler-time check
  - Verify whether a given program is secure/bug-free for arbitrary input

Dynamic Taint Analysis

Sub-State	H/L	Tainted
a	H	1
b	L	0
c	L	0

Taint Monitor
(SW/HW)

- Problems:
  - Granularity
  - Run-time overhead
  - How to handle implicit flow?

Dynamic Taint Analysis

```
1: x = secret;
2: if (x == 0) {
3:     y = 1;
4: } else {
5:     z = 2;
6: }
```

```
1: BREQ x 4
2: y <- 1
3: JMP 5
4: z <- 2
5: …
```

- Problems:
  - Granularity
  - Run-time overhead
  - How to handle implicit flow?

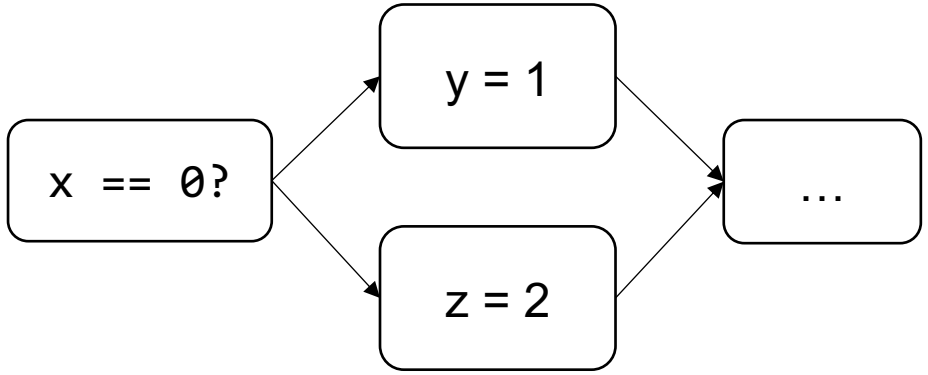
Taint PC.
Taint Explosion.

How to deal with it?

Static Taint Analysis

- Key differences
  - Verify whether a given program is secure for arbitrary inputs
  - Can leverage high-level program information

```
1: x = secret;
2: if (x == 0) {
3:     y = 1;
4: } else {
5:     z = 2;
6: }
```



Control Flow Graph (CFG)

Static Taint Analysis

- Problems
  - Scalability (check all possible inputs)
  - How to handle implicit flow?

How to deal with it?

```
1: x = secret;
2: array[0] = x;
3: z = array[y];
```

```
1: x = secret;
2: *ptr1 = x;
3: z = *ptr2;
```

If conservative,
Taint Explosion.

Takeaways

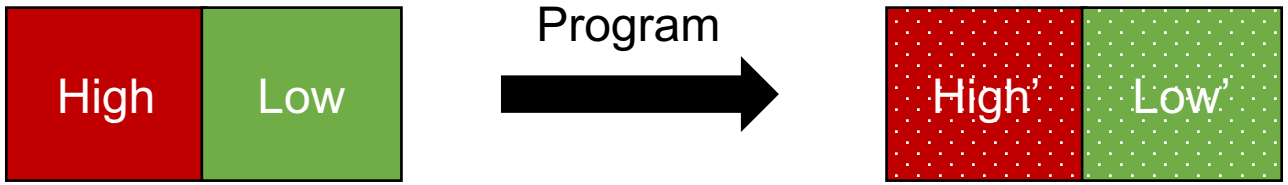
- Non-interference property: general security property for both confidentiality and integrity

- Taint Analysis
  - Useful techniques for checking non-interference
    - Static: verification tool
    - Dynamic: online monitoring
  - Both have taint explosion problems

Non-interference for Timing Side Channels

- How to define non-interference for timing side channels?
- How to check whether a given mitigation achieves non-interference or not?
- How to coordinate software and hardware mitigations? How to reason security about software-hardware co-design?
  - Given SW x, running on HW y can protect all data containing secret z? `{SW x, HW y, sec z}`

Non-interference at Micro-arch Level



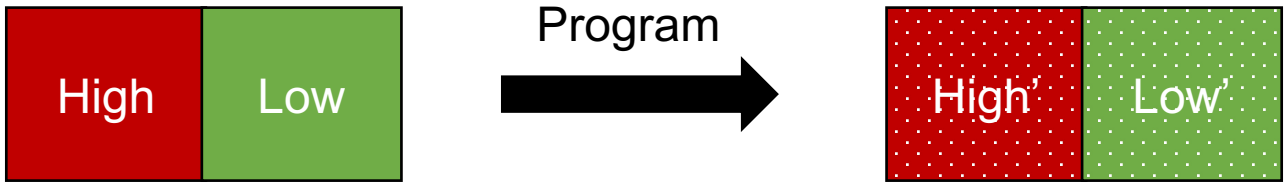
$$\forall\, S1, S2, \qquad (S, P) \rightarrow S'$$

$$\text{if} \quad S1_L = S2_L$$

$$\text{then} \quad S1'_L = S2_L{}'$$

	State	State Transition (Program Execution)
Software Analysis		
Micro-arch Side Channel		

Non-interference at Micro-arch Level



$$\forall\, S1, S2, \qquad (S, P) \rightarrow S'$$

$$\text{if} \quad S1_L = S2_L$$

$$\text{then} \quad S1'_L = S2_L{}'$$

	State	State Transition (Program Execution)
Software Analysis	Register, Memory (virtual)	ISA Emulation
Micro-arch Side Channel	Register, Memory (Physical) Cache, BTB, Bus Busy Bits, Pipeline ROB status etc	Detailed Instruction Execution

Verify HW Design Using Static IFT

Annotate variables (registers and wires) with security labels.

```
reg {L} v, {L} l, {H} h;
// LH (0) = L, LH (1) = H
wire {LH(v)} shared;

// l=h is forbidden
if (v == 0) l = shared;
else h = shared;

// implicit flow, not allowed
if (h == 0) l = 0;
else l = 1;
```

NS: a bit to indicate normal world or secure world

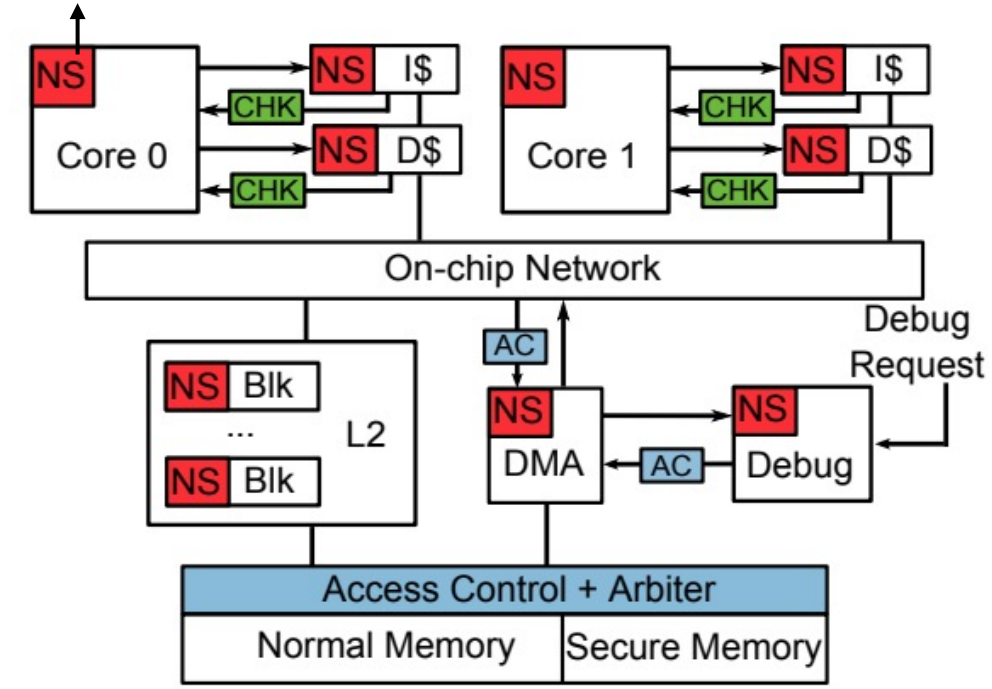
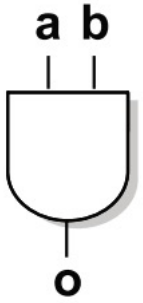


Figure 1. TrustZone prototype implementation.

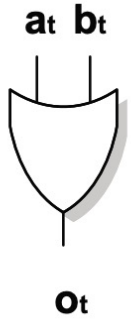
Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis; Ferraiuolo et al; ASPLOS'17
HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security; Ferraiuolo et al; CCS'18

Non-Interference at Gate Level

- Dynamic taint tracking



2-input
AND gate



Shadow Taint Logic
OR gate

Sound, yet Conservative

Logic Truth Table

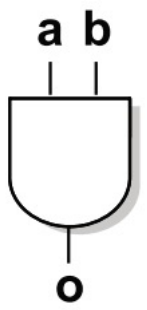
a	b	out
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

a	b	a_t	b_t	out_t
0	0	0	1	**0**
0	1	0	1	**0**
1	0	0	1	**1**
1	1	0	1	**1**

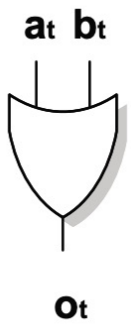
Complete Information Flow Tracking from the Gates Up; Tiwari et al; ASPLOS'09

Non-Interference at Gate Level

- Dynamic taint tracking

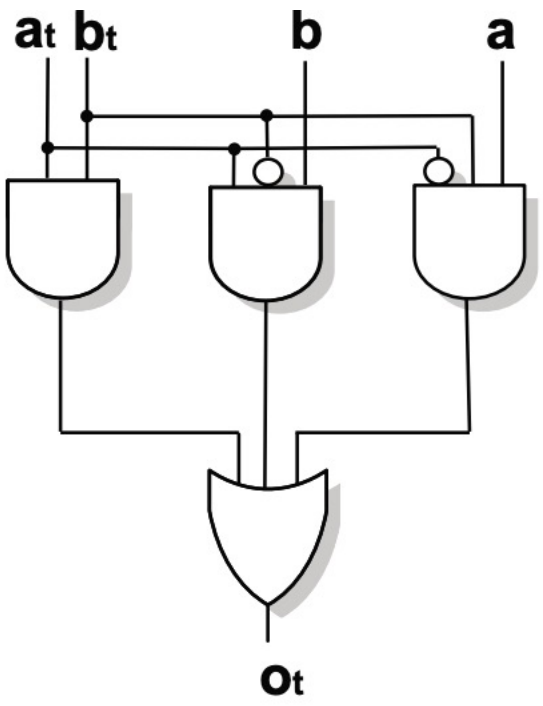


2-input
AND gate



Shadow Taint Logic
OR gate

Sound, yet Conservative



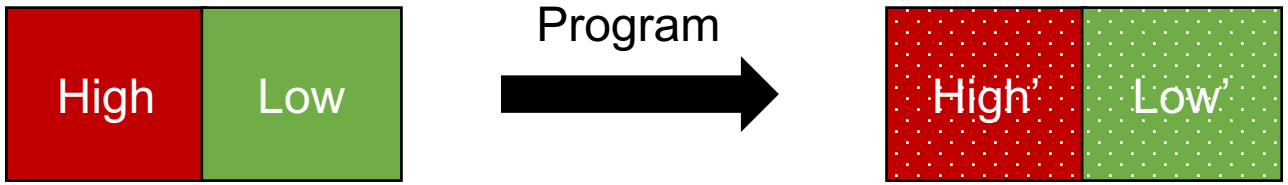
Precise Taint Logic

compose
large
functions

Overhead?

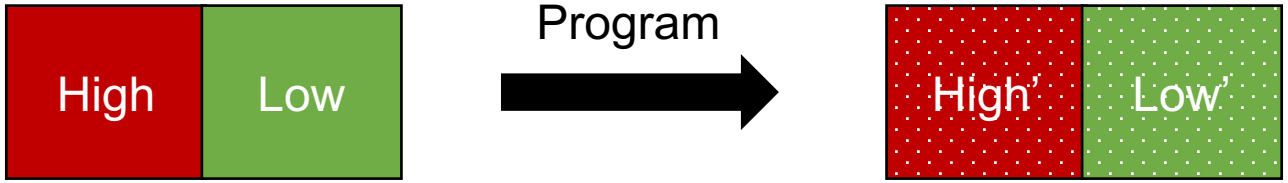
Complete Information Flow Tracking from the Gates Up; Tiwari et al; ASPLOS'09

Non-interference at Micro-arch Level



	State	State Transition (Program Execution)
Software Analysis	Register, Memory (virtual)	ISA Emulation
Micro-arch Side Channel	Register, Memory (Physical) Cache, BTB, Bus Busy Bits, Pipeline ROB status etc	Detailed Execution

Non-interference at Micro-arch Level



Yes, we can. But ...

Can we use the same definition to reason about software mitigations?

	State	
Software Analysis	Register, Memory (virtual)	ISA Emulation
Micro-arch Side Channel	Register, Memory (Physical) Cache, BTB, Bus Busy Bits, Pipeline ROB status etc	Detailed Execution

“Constant-time” Programming

- Write program w/o data-dependent behavior
- Verify non-interference of timing side channels by simulating micro-arch state machine.
- Problems?

Original:

```
bool secret;
x <- pub[secret*64];
```

Data Oblivious:

```
bool secret;
a <- pub[0];
b <- pub[64];
cmov x <- (secret) ? b : a;
```

Observation Model

- Motivation:
  - Avoid verifying SW against specific implementations
- Observations:
  - *Program counters, Memory access addresses, Memory access data, Register data*
  - **Dependent on hardware implementation**

$$\forall\, S1, S2, \qquad (S, P) \rightarrow S', \mathbf{O}$$

$$\text{if} \quad S1_L = S2_L$$

$$\text{then} \quad S1'_L = S2'_L \text{ and } \boldsymbol{O1 = O2}$$

Verify "Constant-time" Programming

- Using Observations

Original:

```
bool secret;
x <- pub[secret*64];
```

Memory access sequence:
H

Data Oblivious:

```
bool secret;
a <- pub[0];
b <- pub[64];
cmov x <- (secret) ? b : a;
```

Memory access sequence:
0 (L), 64 (L)

Takeaways

- How to verify non-interference of timing side channels?
  - To check HW: state transition at micro-arch and gate level
  - To check SW: define observation model
    - Observation model can be served as a contract between HW and SW

Shall we always assume memory access sequence as the observation?

No. It is hardware dependent. Think about Silent Store and Cache compression.

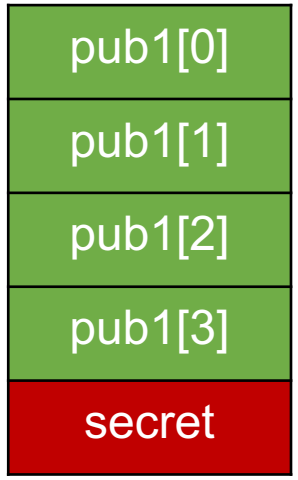
“Constant-time Programming” Fails in the Spectre Era

Original:

```
if (x < limit){ //limit=4
    y <- pub1[x];
    z <- pub2[y*64];
}
```

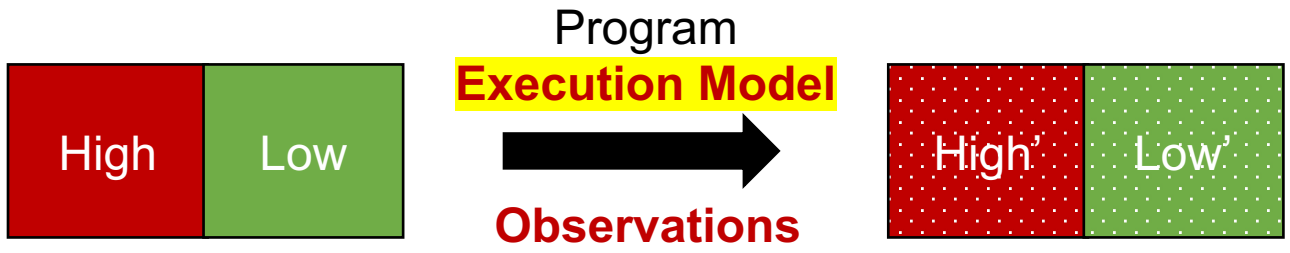
Memory access sequence:
x (L), y (L)

Memory Layout



Execution Model

- Motivation:
  - Incorporate speculative execution in an execution model
- Add Execution Model:
  - Sequential, Branch mis-speculation, etc.



$$\forall\, S1, S2, \qquad (S, P) \rightarrow S', O$$

$$\text{if} \quad S1_L = S2_L$$

$$\text{then} \quad S1'_L = S2'_L \text{ and } O1 = O2$$

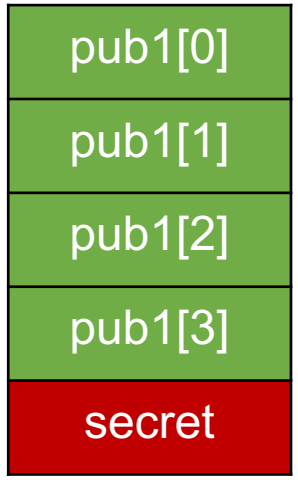
Hardware-Software Contracts for Secure Speculation; Guarnieri et al; S&P'20

“Constant-time Programming” Fails in the Spectre Era

Original:

```
if (x < limit){ //limit=4
    y <- pub1[x];
    z <- pub2[y*64];
}
```

Memory Layout



Memory access sequence (no mispredict):
x (L), y (L)

Memory access sequence (with mispredict):
x (L), y (**H**)

SW Mitigations Against Spectre

fence:

```
if (x < limit){
    lfence();
    y <- pub1[x];
    z <- pub2[y*64];
}
```

Memory access sequence (with mispredict):
∅

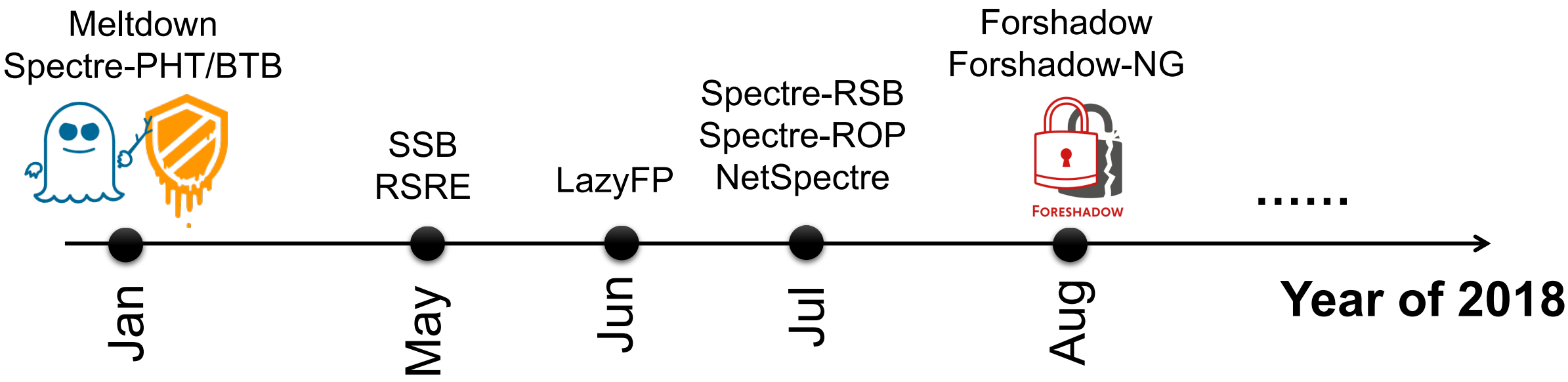
SLH:

```
if (x < limit){
    cmov mask <- (i < limit) 0xFFFF:0
    y <- pub1[x] & mask;
    z <- pub2[x*64];
}
```

Memory access sequence (with mispredict):
0 (L), y (L)

Chandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html

HW Solutions Targeting Many Transient Execution Attacks



Generalization of Transient Execution

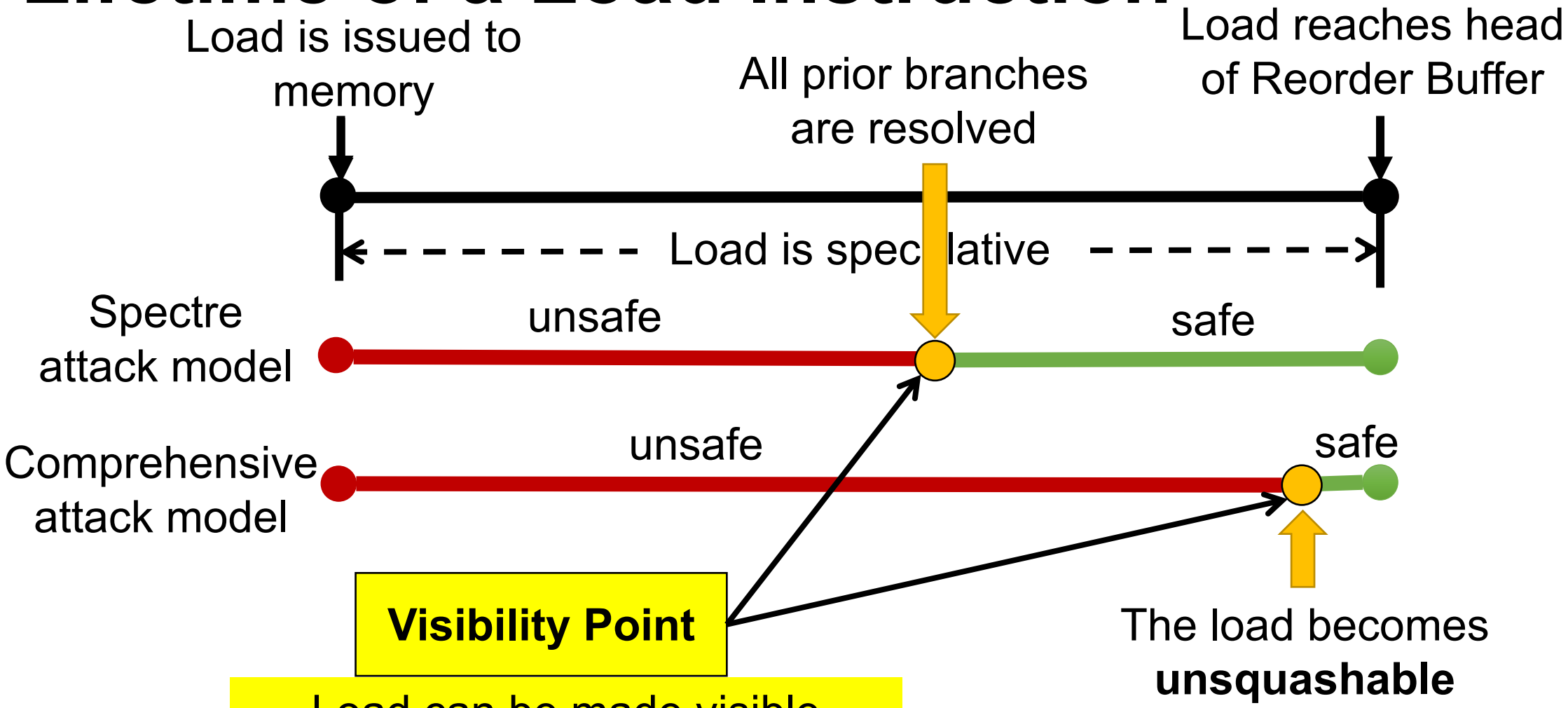
- Different transient execution attacks create transient instructions in different ways
- Speculative attack model: an attacker can exploit ***any*** speculative insts

Speculative Attack Model

Various events, such as:

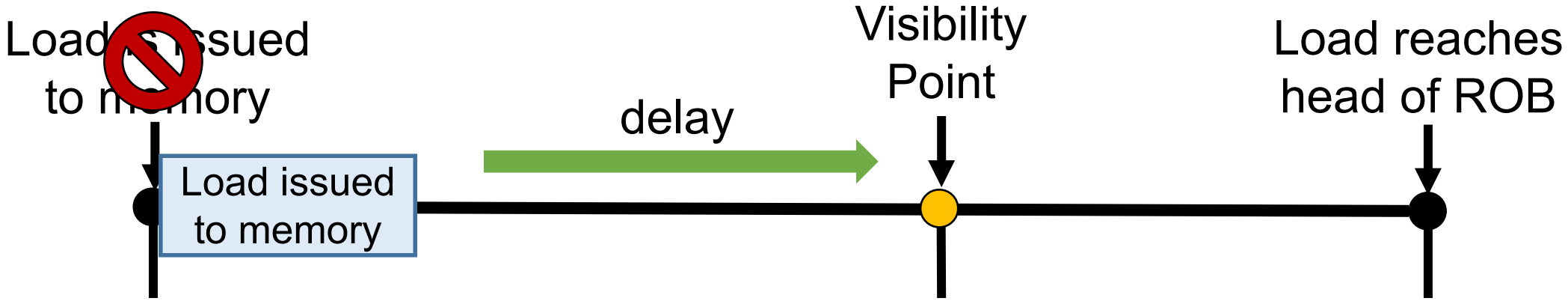
- Control-flow mispredictions → Spectre
- Virtual memory exceptions → Meltdown
- Address alias between a load and an earlier store
- Interrupts
- etc.

Lifetime of a Load Instruction

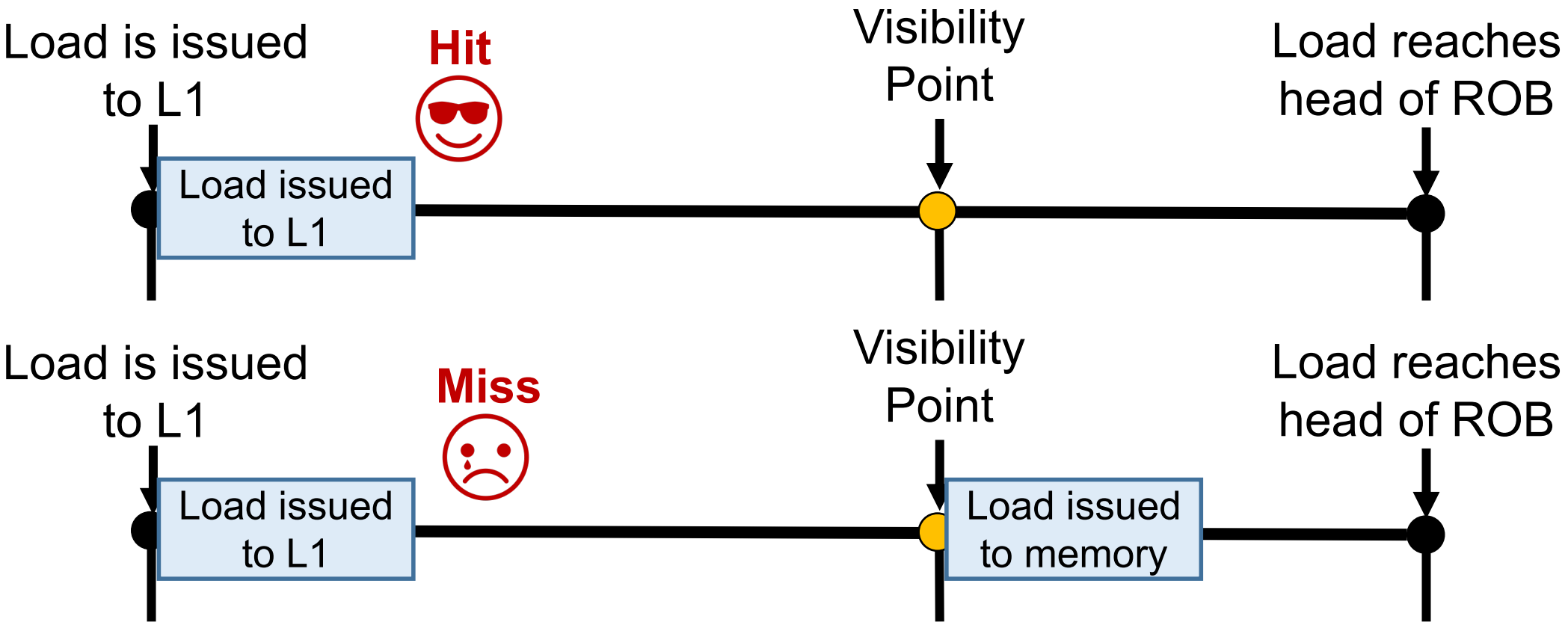


Load can be made visible without compromising security

Naïve Solution: Delay all spec Loads



Delay-on-Miss (DoM)



Christos Sakalis, et al. Efficient invisible speculative execution through selective delay and value prediction. ISCA'19

Performance Optimizations

```
// x is committed
Br:  if (x < size){
// speculation starts here
Ld1:     y = array1[x]
Ld2:     z = array2[y]
     }
```

x: Committed register state (exists in legal execution)

y: Transient register state (does not exist in legal execution)

Insight: Only need to protect the instructions that <u>**use**</u> transient states

STT, NDA, etc

```
// x is committed
Br:  if (x < size){
// speculation starts here
Ld1:     y = array1[x]
Ld2:     z = array2[y]
     }
```

Access Instruction
(brings transient state
into pipeline)

Taint Tracking of
Speculative Data

Transmit Instruction
(uses transient state)

Only Protect Transmit
Instructions

Jiyong Yu, Mengjia Yan, et al. Speculative Taint Tracking (STT): A Comprehensive [illegible] for Speculatively Accessed Data. MICRO'20

Comparing Two Approaches

Speculative

Committed

```
// x is committed
Br:  if (x < size){
// speculation starts here
Ld1:    y = array1[x]
Ld2:    z = array2[y]
      }
```

	DoM/InvisiSpec/…	STT/NDA/…
Ld1	Need protection	**NO** protection
Ld2	Need protection	Need protection
Ld2 is secure:	**Ld2** reaches visibility point	When **Ld1** reaches visibility point

Comparing Two Approaches

Speculative

Committed

```
// x is committed
Ld1:    y = array1[x]
Br:   if (x < size){
// speculation starts here
Ld2:     z = array2[y]
      }
```

Need protection

InvisiSpec/DoM/…

STT/NDA/…

NO protection

Problems

- Different HW mitigations achieve different security properties
- How to communicate this information to SW?
  - ISA?
  - List code patterns?
  - Specify execution model + observation model?

Current state-of-the-art.
May not be the final solution.
An unsolved research problem.

Analyze Security Properties

Disable Speculation

No Protection

		Execution Model	
		Sequential	Speculative (can mispredict)
Observation Model	Program Counter		
	Program Counter + Memory Address		
	Program Counter + Memory Address + Register Content		

Analyze Security Properties

		Execution Model	
		Sequential	Speculative (can mispredict)
Observation Model	Program Counter		
	Program Counter + Memory Address	Disable Speculation	
	Program Counter + Memory Address + Register Content	No Protection	No Protection

Analyze Security Properties

DoM		Execution Model	
		Sequential	Speculative (can mispredict)
Observation Model	Program Counter		
	Program Counter + Memory Address	Disable Speculation	
	Program Counter + Memory Address + Register Content		

Analyze Security Properties

		Execution Model	
		Sequential	Speculative (can mispredict)
Observation Model	Program Counter		DoM
	Program Counter + Memory Address	Disable Speculation	
	Program Counter + Memory Address + Register Content	DoM	

Analyze Security Properties

STT

		Execution Model	
		Sequential	Speculative (can mispredict)
Observation Model	Program Counter		DoM
	Program Counter + Memory Address	Disable Speculation	
	Program Counter + Memory Address + Register Content	DoM	

Analyze Security Properties

		Execution Model	
		Sequential	Speculative (can mispredict)
Observation Model	Program Counter		DoM
	Program Counter + Memory Address	Disable Speculation	STT
	Program Counter + Memory Address + Register Content	DoM, STT	

Summary

- Non-interference
  - A general security property that can be used to reason software security and micro-arch side channels
  - Pros/Cons of static and dynamic taint analysis
- Reason about non-interference for side mitigations
  - Both observation model and execution model are hardware dependent
- Fundamental problem, timing is not defined at the contract between HW and SW (currently ISA)