

# Transient execution attacks and defenses

Formal models and provable security

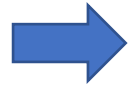
Frank Piessens

MIC-SEC Winter school, Paris, Dec 2022

# Introduction

- Transient execution attacks are a relatively new and a challenging class of microarchitectural attacks
- My objective in this lecture is showing the value of the formal study of these attacks and corresponding defenses
- Formal models can have several objectives:
  - Simplifying a phenomenon to its bare essentials
    - This is great for explaining attacks and defenses!
  - Proving that a defense achieves a well-specified security objective under well-specified assumptions
    - This is great for increasing assurance that a defense works as advertised!
- This talk will illustrate both these things, but with a focus on the explanation part

# Overview

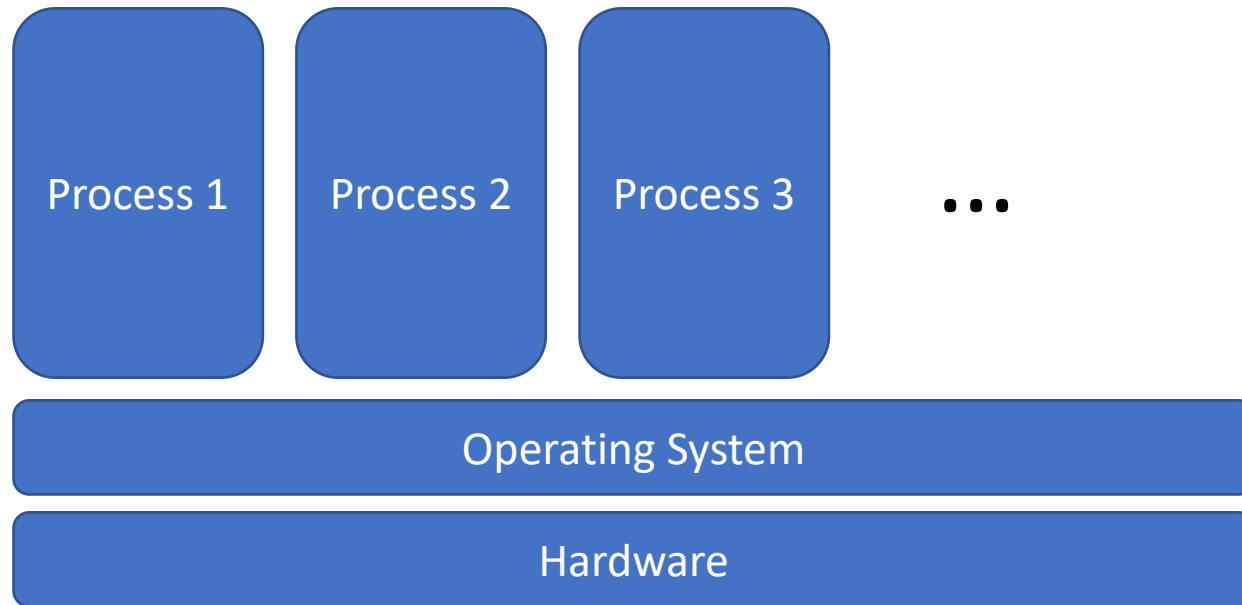


- System model
  - Architectural isolation mechanisms for shared platforms
  - Microarchitectural attacks
  - A simple model
- Microarchitectural side-channel attacks
- Transient execution attacks
- Defenses
- Conclusions

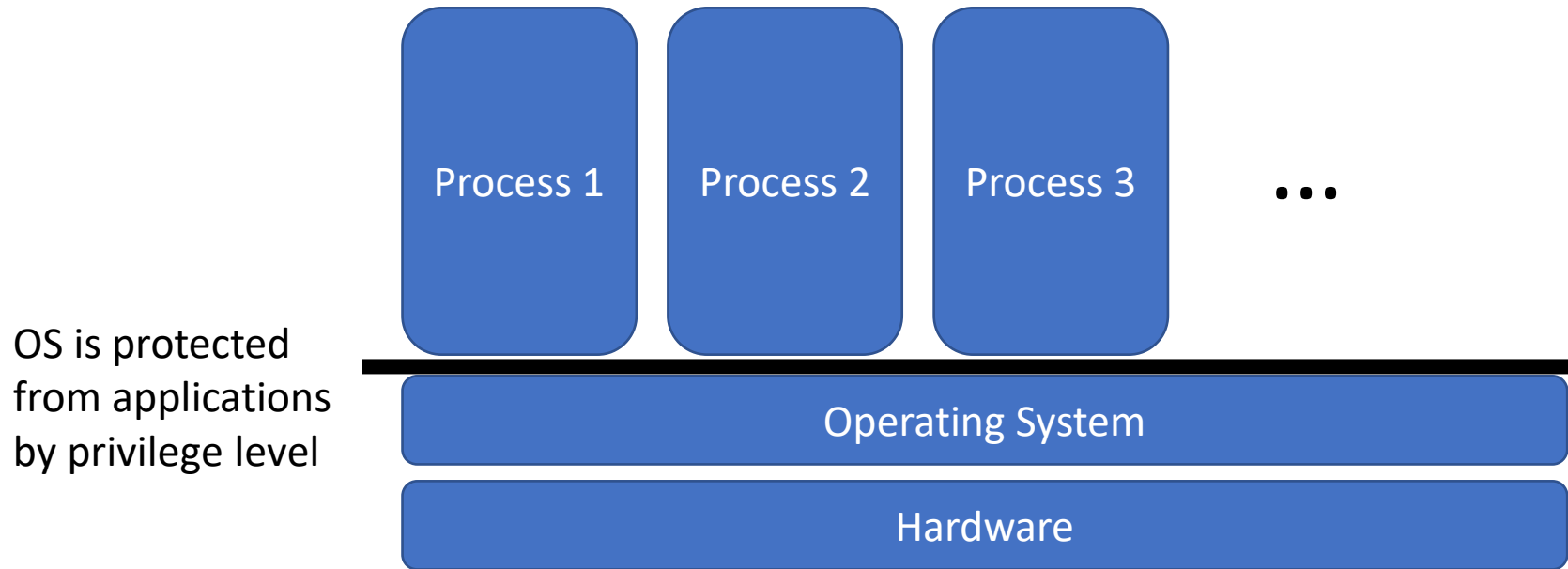
# System model: a shared platform

- A platform runs programs from multiple stakeholders
  - Isolation mechanism isolates these programs
  - The platform supports communication between these programs
- Many systems are such shared platforms:
  - Cloud
  - Mobile
  - Desktop
- A variety of **isolation mechanisms** is used to limit interference between code from different stakeholders

# Classic hierarchical OS protection

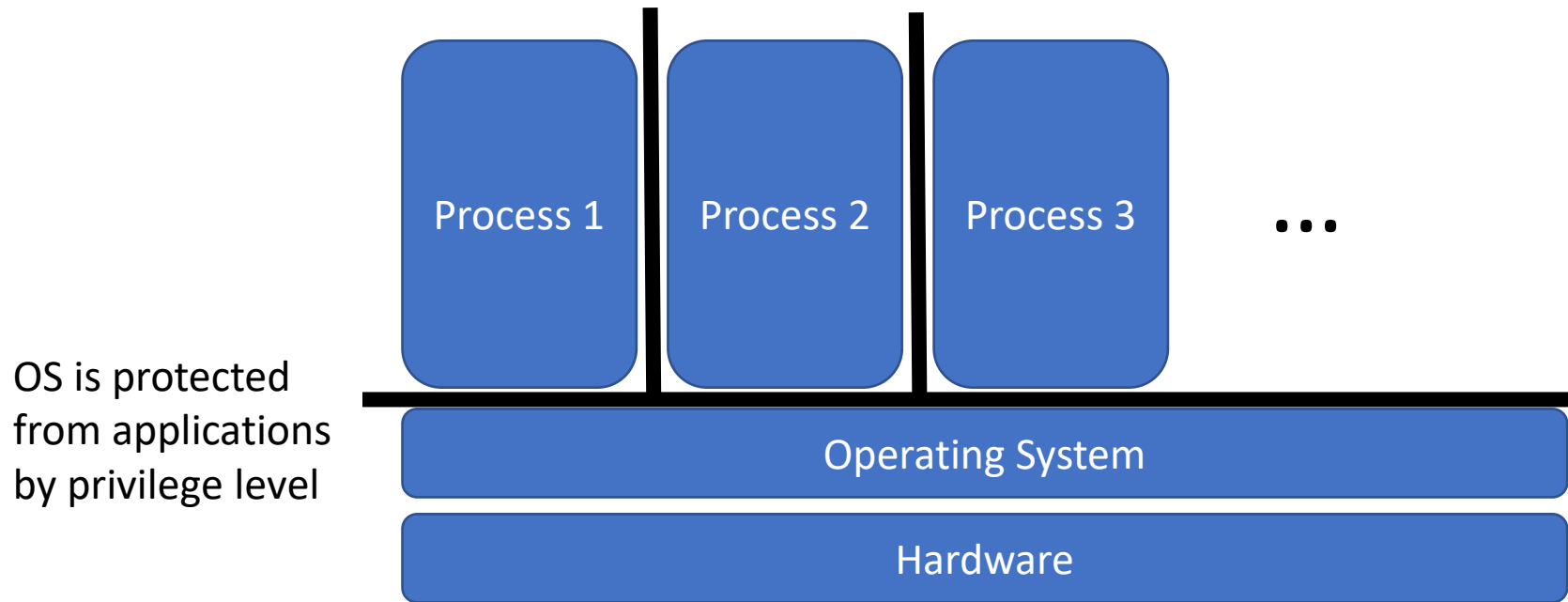


# Protecting the kernel: privilege levels

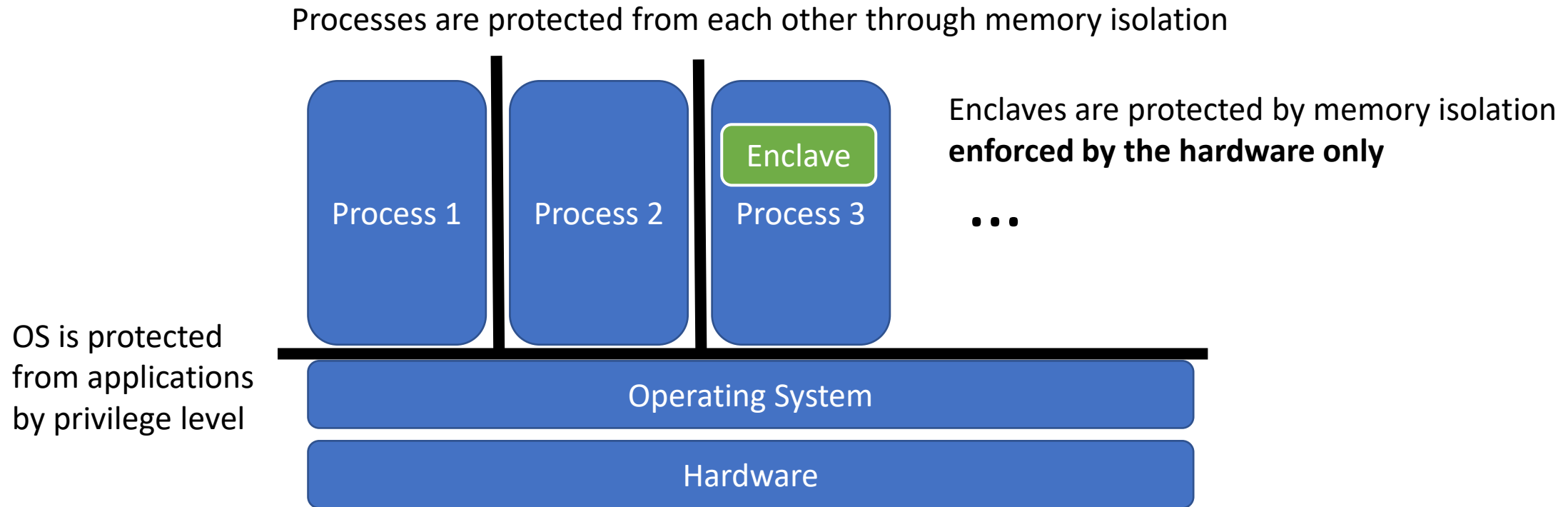


# Protecting processes: virtual memory

Processes are protected from each other through memory isolation

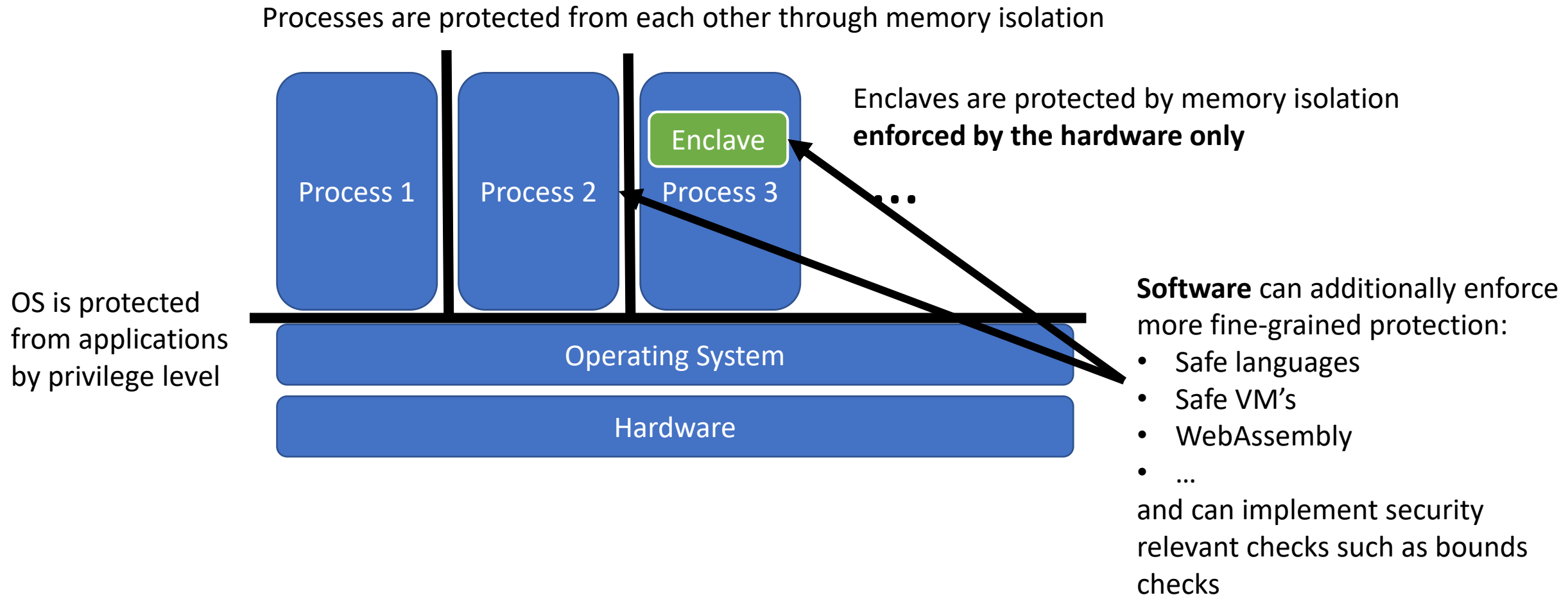


# Protecting critical software: enclaves





# Fine-grained protection: software

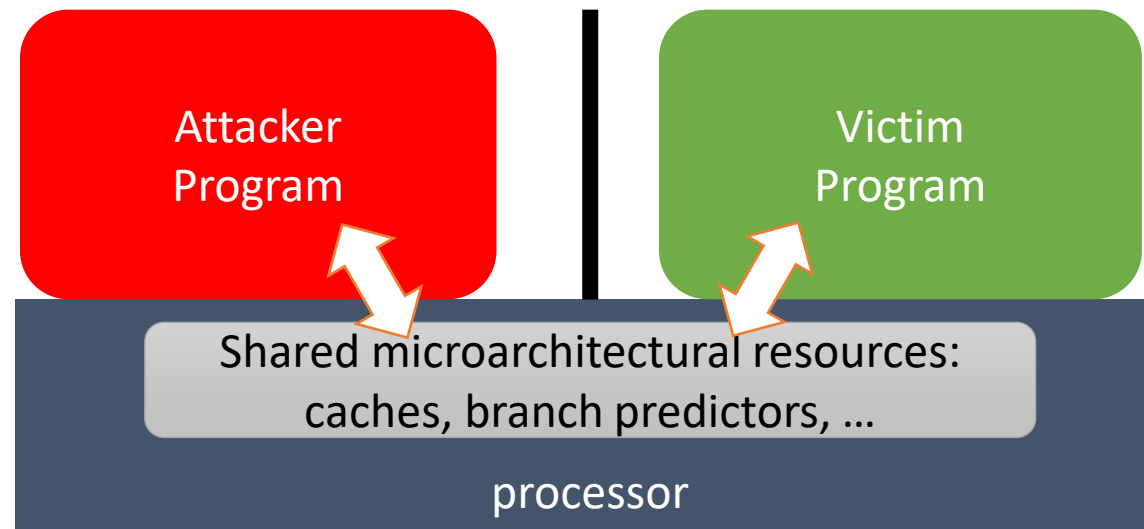


# Architecture versus microarchitecture

- The **Instruction Set Architecture (ISA)** defines how the machine code of a processor behaves
  - Examples: x86, RISC-V, ARM, ...
  - The ISA defines:
    - Architectural state: memory, registers, ...
    - Instruction semantics
- The **microarchitecture** is the way the ISA is implemented in a particular processor
  - Examples: single-cycle versus pipelined, in-order versus out-of-order, ...
  - This can introduce additional state and behavior:
    - Microarchitectural state: e.g., for performance improvements (caches, branch predictor state, various CPU buffers, ...)
    - Behavior: speculative execution, out-of-order execution, ...

# Microarchitectural attacks

- Isolation mechanisms guarantee **architectural isolation**
- Microarchitectural attacks aim to break isolation by exploiting the fact that the microarchitecture shares resources across isolation domains



# A simple Instruction Set Architecture (ISA) model

Register names	$r \in \text{Regs}$	<i>(E.g., <math>r_0, r_1, i, \text{len}</math>. We assume <math>pc \notin \text{Regs}</math>)</i>
Values	$v \in \mathbb{N}$	<i>(also represent addresses)</i>
Expressions	$e ::= v \mid r \mid e + e \mid e < e \mid \dots$	<i>(boolean expressions return 0 or 1)</i>
Instructions	$i ::= r \leftarrow e$	<i>(assign value of <math>e</math> to <math>r</math>)</i>
	$r \leftarrow \text{load}[e]$	<i>(load value at memory address <math>e</math> into <math>r</math>)</i>
	$\text{store}[e] \leftarrow r$	<i>(store <math>r</math> in memory at address <math>e</math>)</i>
	$\text{jmp } e$	<i>(jump to code address <math>e</math>)</i>
	$\text{beqz } r \ v$	<i>(branch to <math>v</math> if <math>r</math> evaluates to 0 )</i>
Programs	$p ::= \vec{i}$	<i>(non-empty list of instructions)</i>

Example program:

```

0  :   $r_0 \leftarrow i < 2$           ; while ( $i < 2$ ) {
1  :  beqz  $r_0$  6                    ;
2  :   $r_0 \leftarrow \text{load}[a + i]$     ;  $\text{sum} = \text{sum} + a[i]$ 
3  :   $\text{sum} \leftarrow \text{sum} + r_0$       ;
4  :   $i \leftarrow i + 1$               ;  $i = i + 1$ 
5  :  jmp 0                          ; }
```

# A simple Instruction Set Architecture (ISA) model

Register names	$r \in \text{Regs}$	(E.g., $r_0, r_1, i, \text{len}$ . We assume $pc \notin \text{Regs}$ )
Values	$v \in \mathbb{N}$	(also represent addresses)
Expressions	$e ::= v \mid r \mid e + e \mid e < e \mid \dots$	(boolean expressions return 0 or 1)
Instructions	$i ::= r \leftarrow e$	(assign value of $e$ to $r$ )
	$r \leftarrow \text{load}[e]$	(load value at memory address $e$ into $r$ )
	$\text{store}[e] \leftarrow r$	(store $r$ in memory at address $e$ )
	$\text{jmp } e$	(jump to code address $e$ )
	$\text{beqz } r \ v$	(branch to $v$ if $r$ evaluates to 0 )
Programs	$p ::= \vec{i}$	(non-empty list of instructions)

Example program:

```

0 :  $r_0 \leftarrow i < 2$  ; while ( $i < 2$ ) {
1 : beqz  $r_0$  6 ;
2 :  $r_0 \leftarrow \text{load}[a + i]$  ;  $sum = sum + a[i]$ 
3 :  $sum \leftarrow sum + r_0$  ;
4 :  $i \leftarrow i + 1$  ;  $i = i + 1$ 
5 : jmp 0 ; }

```

Registers:

pc=0	
a	0
i	0
sum	0
r0	0

Memory:

...	
1:	4
0:	5

# Base semantics

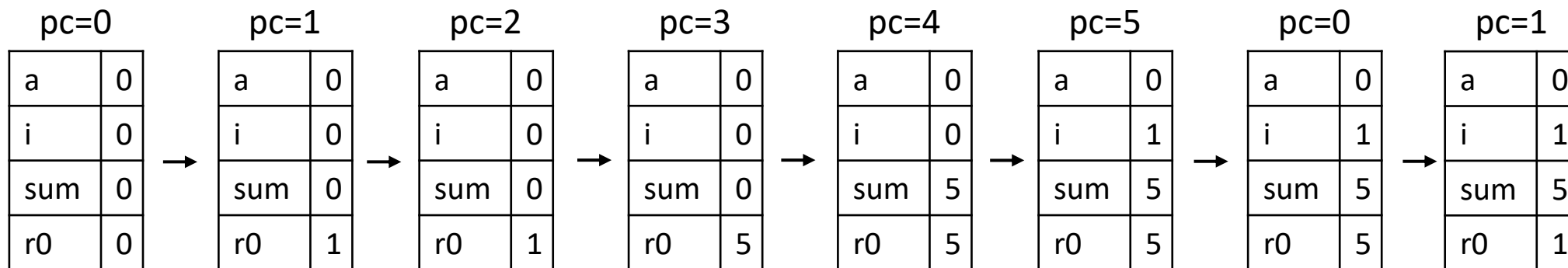
Register state  $\rho \in \text{Regs} \rightarrow \text{Values}$   
 Memory state  $m ::= \vec{v}$   
 Program counter  $pc ::= v$   
 Program state  $\sigma ::= (m, \rho, pc)$

*(mapping from register names to values)*  
*(list of values)*  
*(an index into the program)*

Program:

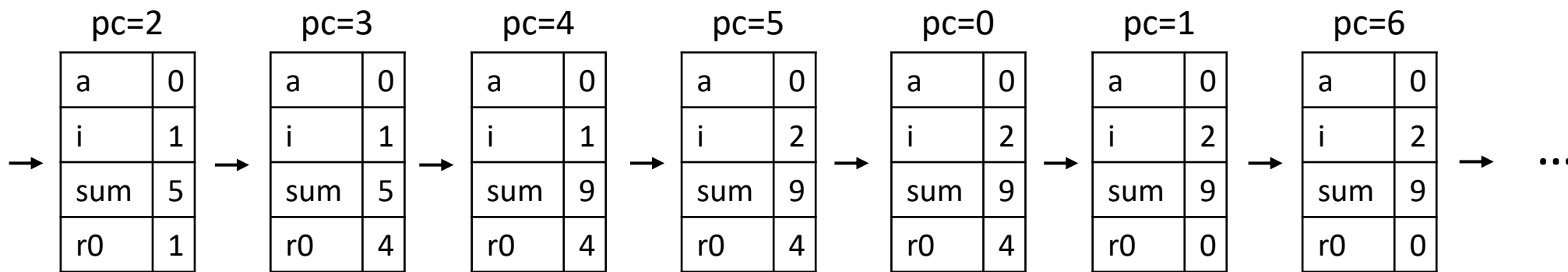
```

0 :  $r_0 \leftarrow i < 2$ 
1 : beqz  $r_0$  6
2 :  $r_0 \leftarrow \text{load}[a + i]$ 
3 :  $sum \leftarrow sum + r_0$ 
4 :  $i \leftarrow i + 1$ 
5 : jmp 0
    
```




Memory:

...	
1:	4
0:	5

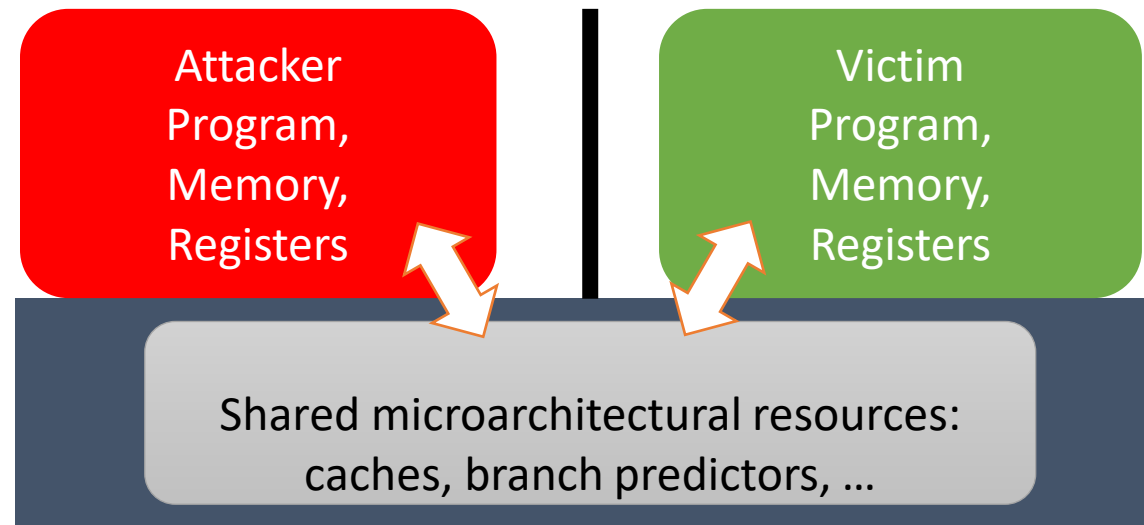


# Overview

- System model
  - Architectural isolation mechanisms for shared platforms
  - Microarchitectural attacks
  - A simple model
-  • Microarchitectural side-channel attacks
- Transient execution attacks
- Defenses
- Conclusions

# Architectural isolation

- We think of architectural state as securely partitioned
  - Programs by different stakeholders are (architecturally) *isolated* from one another
  - At the level of abstraction of the ISA, **no information leaks** between ISA programs of different stakeholders





# Modeling classic microarchitectural side-channels

- Attackers running code on the same platform can learn information about the execution of victim code
  - For instance, attackers can learn:
    - Program control flow, for instance by a cache attack on code memory
    - Addresses of data memory accesses, for instance by a cache attack on data memory
- We model this by assuming a **leakage model** that specifies what can leak to an attacker at each execution step
  - See, for instance: Almeida et al., *Verifying Constant-Time Implementations*, USENIX Security 2016

# The constant time leakage model

- Extend base semantics to specify what leaks at each step:

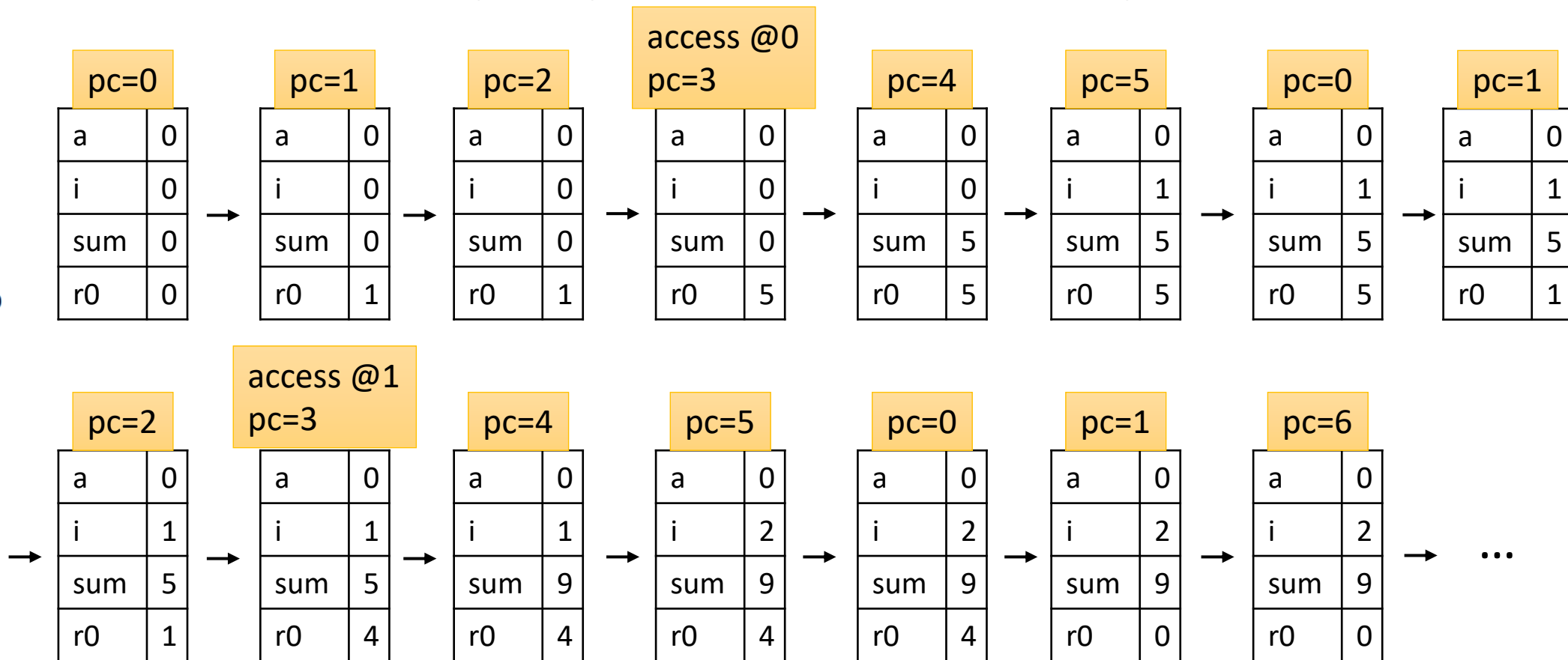
Program:

```

0 :  $r_0 \leftarrow i < 2$ 
1 : beqz  $r_0$  6
2 :  $r_0 \leftarrow \text{load}[a + i]$ 
3 :  $\text{sum} \leftarrow \text{sum} + r_0$ 
4 :  $i \leftarrow i + 1$ 
5 : jmp 0
    
```

Memory:

...	
1:	4
0:	5




# Leak gadgets

- In later attacks, we rely on code snippets that leak values through a microarchitectural side-channel
  - A wide variety of such snippets exist
  - In some scenarios, the attacker can construct them, in other scenarios the attacker must find them in victim code
- To make it obvious that some information leaks, we use the macro:

$$\text{leak } secret \quad ::= \quad dummy \leftarrow \text{load}[secret]$$

(where *secret* is the name of a register containing a value to be leaked, and *dummy* is an otherwise unused register)

# Overview

- System model
- Microarchitectural side-channel attacks
-  • Transient execution attacks
  - Out-of-order and speculative execution
  - Spectre attacks
- Defenses
- Conclusions

# Out-of-order and speculative execution

- Transient execution attacks exploit processor features called *out-of-order and speculative execution*
- The basic idea is:
  - Rather than executing one instruction at a time, **fetch** many instructions into a buffer of *in-flight instructions*
  - **Execute** instructions from this buffer, possibly out-of-order
    - This avoids having to wait while, for instance a slow memory load is happening
  - **Commit** the effect of the instructions to the architectural state in order
- Prediction and speculation are used to speed things up
  - For instance, fetching instructions beyond a branch requires prediction

# Out-of-order and speculative execution

In-flight instructions  $f ::=$

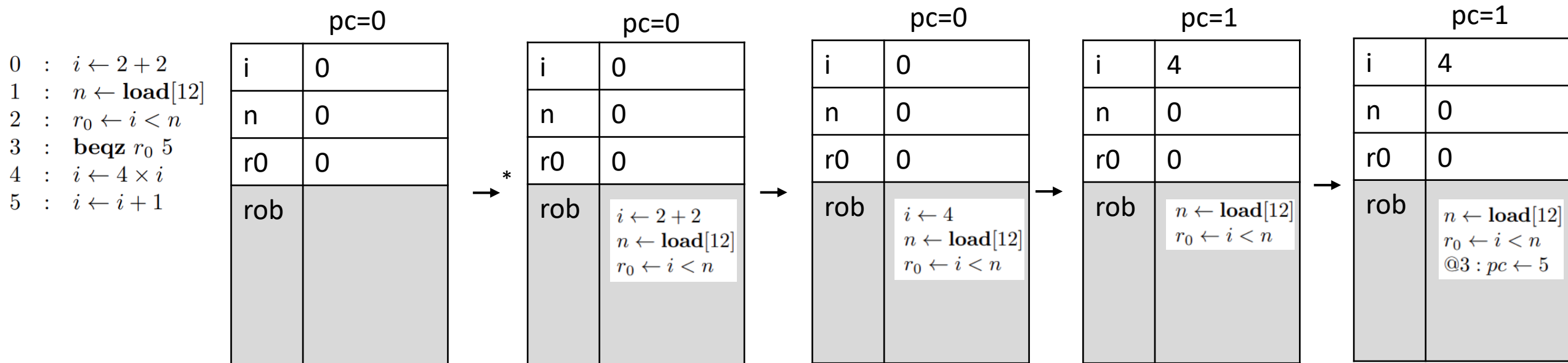
$r \leftarrow e$	
$r \leftarrow \text{load}[e]$	
$\text{store}[e] \leftarrow r$	
$pc \leftarrow v$	
$@v : pc \leftarrow v$	
$@v : r \leftarrow v$	

*(non-speculated jump becomes pc assignment)*

*(speculated jump,  $v$  is address of original instruction)*

*(speculated load,  $v$  is address of original instruction)*

Reorder buffer  $rob ::= \overrightarrow{f}$   
 Program state  $\sigma ::= (m, \rho, pc, rob)$

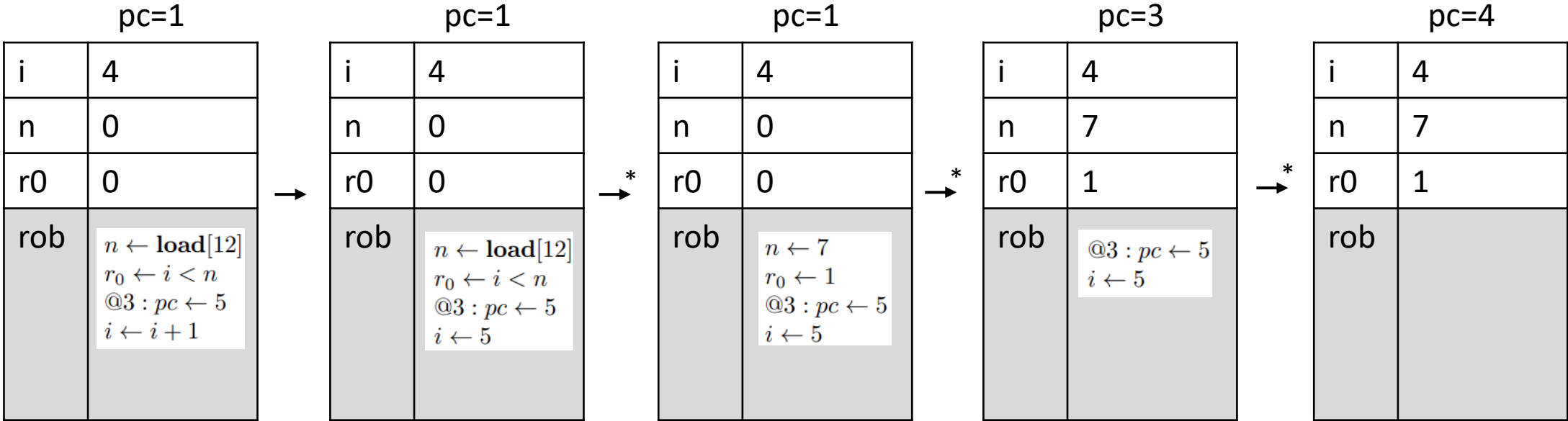


```

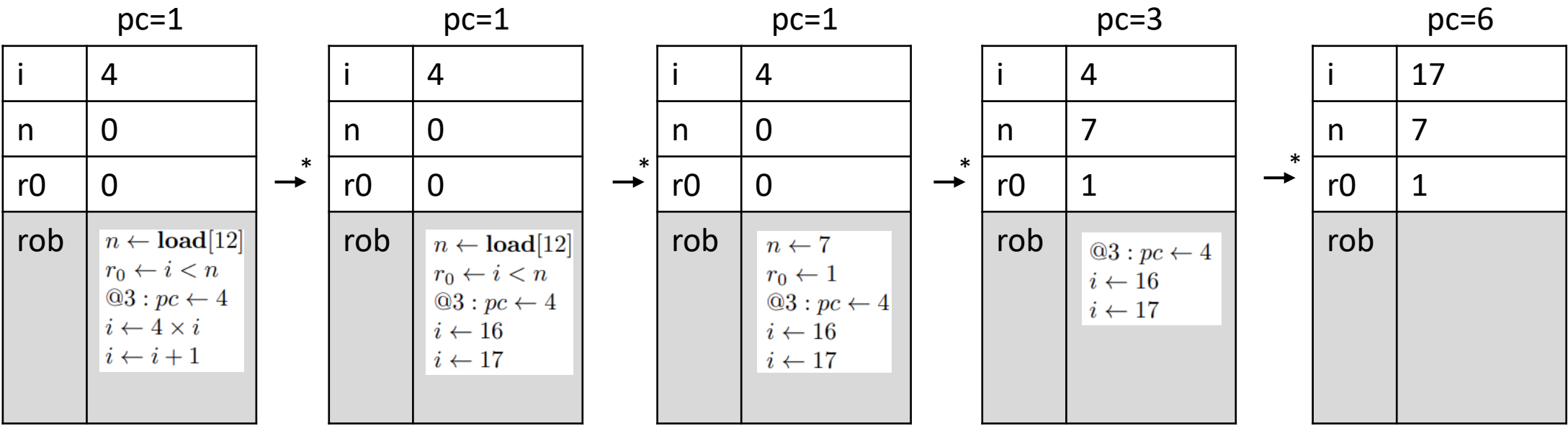
0 : i ← 2 + 2
1 : n ← load[12]
2 : r0 ← i < n
3 : beqz r0 5
4 : i ← 4 × i
5 : i ← i + 1

```

incorrect  
prediction



correct  
prediction



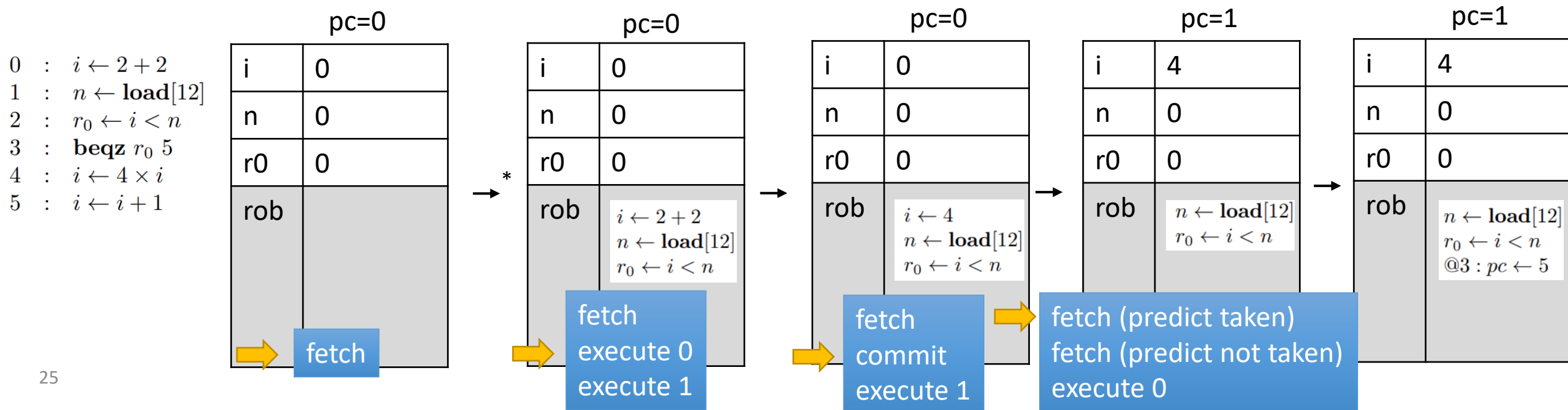
# Predictions and scheduling

- The semantics requires the processor to make choices, for instance for predicted values
  - These happen based on heuristics and observing past behavior
  - Hence, they can also be influenced by an attacker
    - E.g., “training the branch-predictor”
- How should we model this influence of the attacker?



# Attacker influence on the execution

- Prediction and scheduling choices can be done by the attacker within constraints defined in the semantics, e.g.:
  - Fetch is only possible if the reorder buffer has room
  - Executing an instruction in the reorder buffer is only possible if its dependencies are satisfied
  - Commit is only possible for the oldest instruction in the reorder buffer, and only after it has fully executed



# Transient execution attacks

- We have seen that instructions can execute transiently
- This impacts security in two ways:
  - Transiently executed instructions can also leak information to the attacker
    - On rollback, architectural effects are discarded, but microarchitectural effects remain
  - Transiently executed instructions can **access** information expected to be inaccessible
    - Because the information is protected by software -> “Spectre”-style attacks
    - Because it is in another hardware protection domain -> “Meltdown”-style attacks
- For this talk, we focus just on Spectre-style attacks
  - These are the hardest to defend against efficiently

# Spectre examples

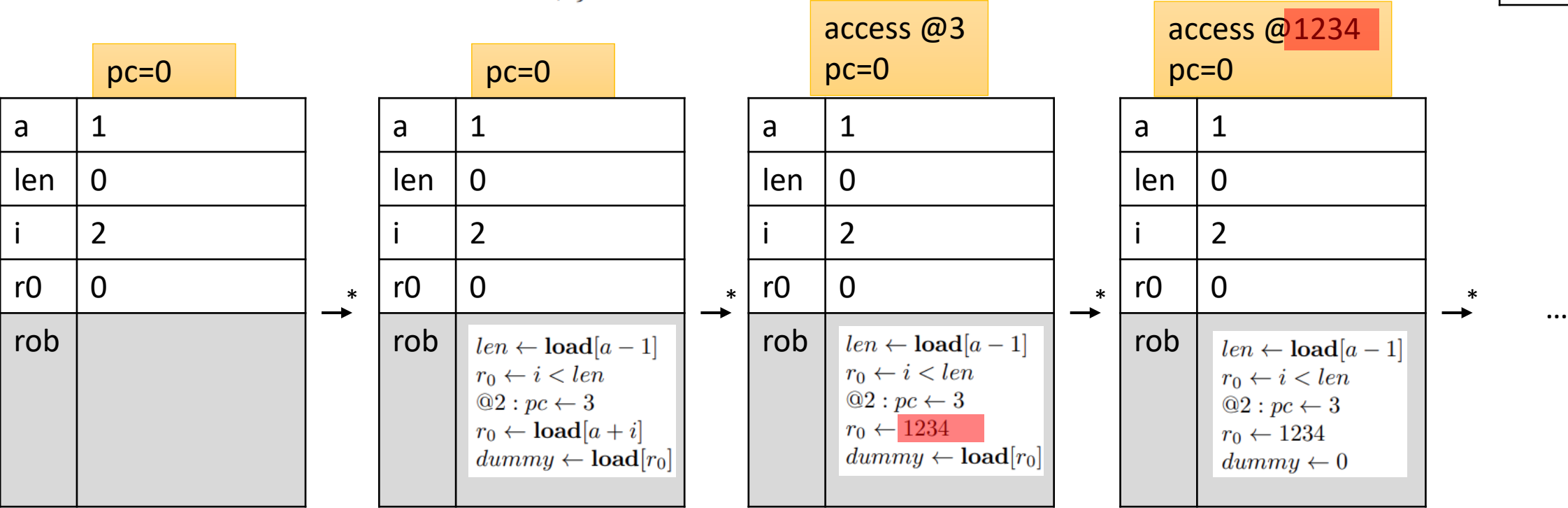
- We will discuss a couple of Spectre examples
- In each example:
  - There is code operating in a program state containing secrets
  - According to the base ISA semantics, the code does not leak these secrets
    - Even taking into account the constant time leakage model
  - Yet, because of speculation and out-of-order execution, the secrets **do** leak

# Example 1: Spectre v1 (Spectre-PHT)

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 5 ; if(i < len){
3 : r0 ← load[a + i] ; r0 = a[i]
4 : leak r0 ; leak(r0)
5 : ... ; }
```

Memory:

	1234:	0
	...	...
	3:	1234
a:	2:	5
	1:	3
len:	0:	2

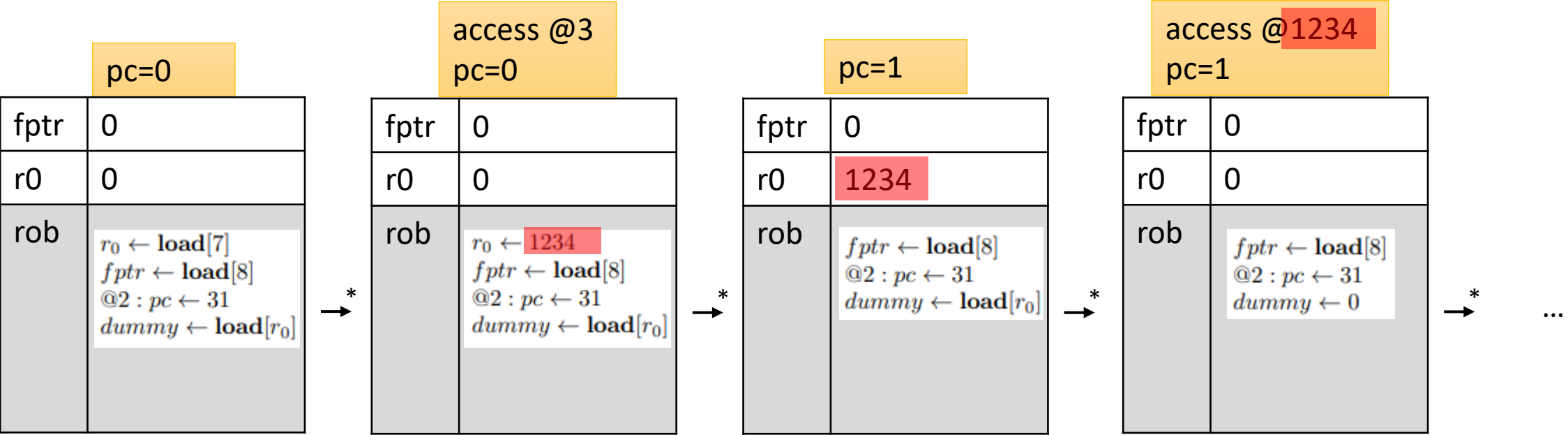


# Example 2: Spectre v2 (Spectre-BTB)

```
0 : r0 ← load[7] ; load a secret into r0
1 : fptr ← load[8] ; load a "function pointer" to a trusted function
2 : jmp fptr ; call trusted function that safely accesses secret
... : ...
20 : r0 ← 0 ; trusted function just clears secret
21 : jmp 3
... : ...
31 : leak r0
... : ...
```

Memory:

1234:	0
...	...
8:	20
7:	1234
...	...
0:	0



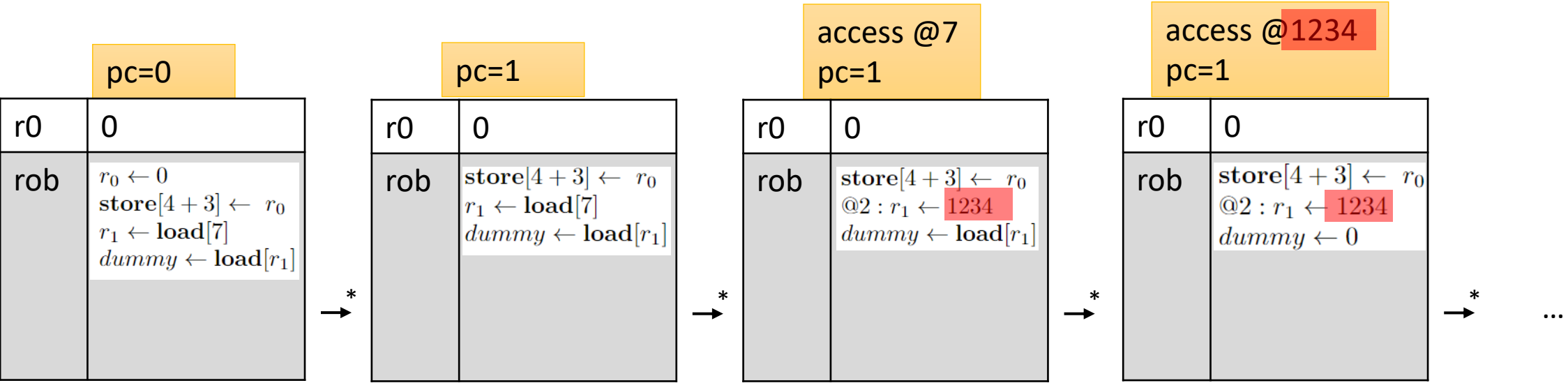
# Example 3: Spectre v4 (Spectre-STL)

Memory:

1234:	0
...	...
7:	1234
...	...
0:	0

*; Suppose memory address 7 contains the secret 1234, that is currently cached*

```
0 : r0 ← 0
1 : store[4 + 3] ← r0 ; overwrite the secret with 0
2 : r1 ← load[7]      ; load address 7 into r1, should read 0
3 : leak r1
... : ...
```



# Transient execution attacks

- These were a couple of **simplified** Spectre attacks
  - See <https://transient.fail/> for more variants and more details
- Note that our simplifications make the attacks look simpler than they are.
  - Doing these attacks on real system requires a significant amount of expertise

## Spectre-PHT (*aka Spectre v1*)




Kocher et al. first introduced Spectre-PHT, an attack that poisons the Pattern History Table (PHT) to mispredict the direction (taken or not-taken) of conditional branches. Depending on the underlying microarchitecture, the PHT is accessed based on a combination of virtual address bits of the branch instruction plus a hidden Branch History Buffer (BHB) that accumulates global behavior for the last N branches on the same physical core.

### References

- [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 Evtvushkin, Daniel Gruss (*USENIX Security 2019*)
- [Spectre Attacks: Exploiting Speculative Execution](#)  
Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom (*IEEE S&P 2019*)
- [BranchScope: A New Side-Channel Attack on Directional Branch Predictor](#)  
Dmitry Evtvushkin, Ryan Riley, Nael Abu-Ghazaleh, Dmitry Ponomarev (*ASPLOS 2018*)
- [The microarchitecture of Intel, AMD and VIA CPUs](#)  
Agner Fog

# Overview

- System model
- Microarchitectural side-channel attacks
- Transient execution attacks
  - Out-of-order and speculative execution
  - Spectre attacks
-  • Defenses
- Conclusions



# Defenses

- Defenses against transient execution attacks are being investigated at multiple levels:
  - Hardware fixes
    - For instance, do not forward values from faulting loads to subsequent instructions
  - Operating system level fixes
    - For instance, do not place the kernel in the same virtual address space as user code
  - Compiler level fixes
    - For instance, insert instructions to stop out-of-order execution, or rewrite code to remove the vulnerability
- We focus on defenses against Spectre attacks

# Security objective of defenses

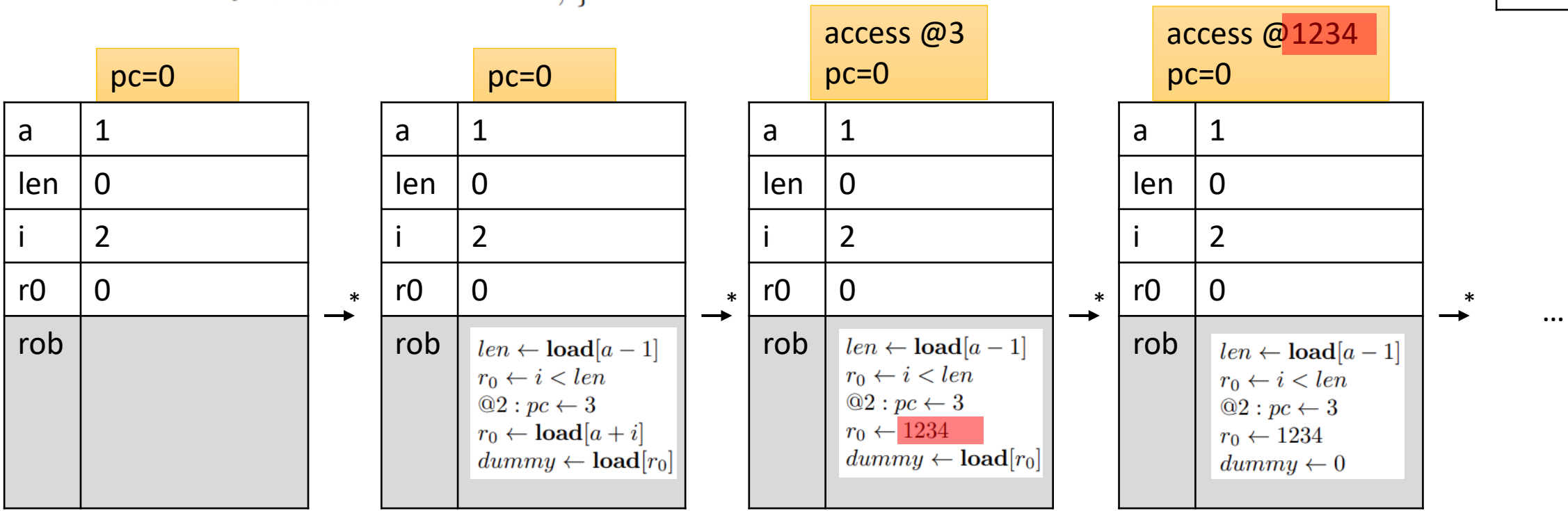
- Microarchitectural side-channel attacks and transient execution attacks cause unexpected information leaks
- The security objective of a defense depends on how much program state we actually want to keep secret
- We define a **policy** as an equivalence relation over program states
  - The intuition is that the policy relates states that should be *indistinguishable* to an attacker. Typically, one defines a policy by marking secrets, and two states are equivalent if they only differ in secrets.
- A program P is secure on hardware H if executing P on H starting from any two equivalent initial states will produce identical observations for the attacker
  - Security can be achieved by software mitigations, or by hardware mitigations, or by a combination of both

# Reconsider the Spectre v1 example:

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 5 ; if(i < len){
3 : r0 ← load[a + i] ; r0 = a[i]
4 : leak r0 ; leak(r0)
5 : ... ; }
```

Memory:

	1234:	0
	...	...
	3:	1234
a:	2:	5
	1:	3
len:	0:	2



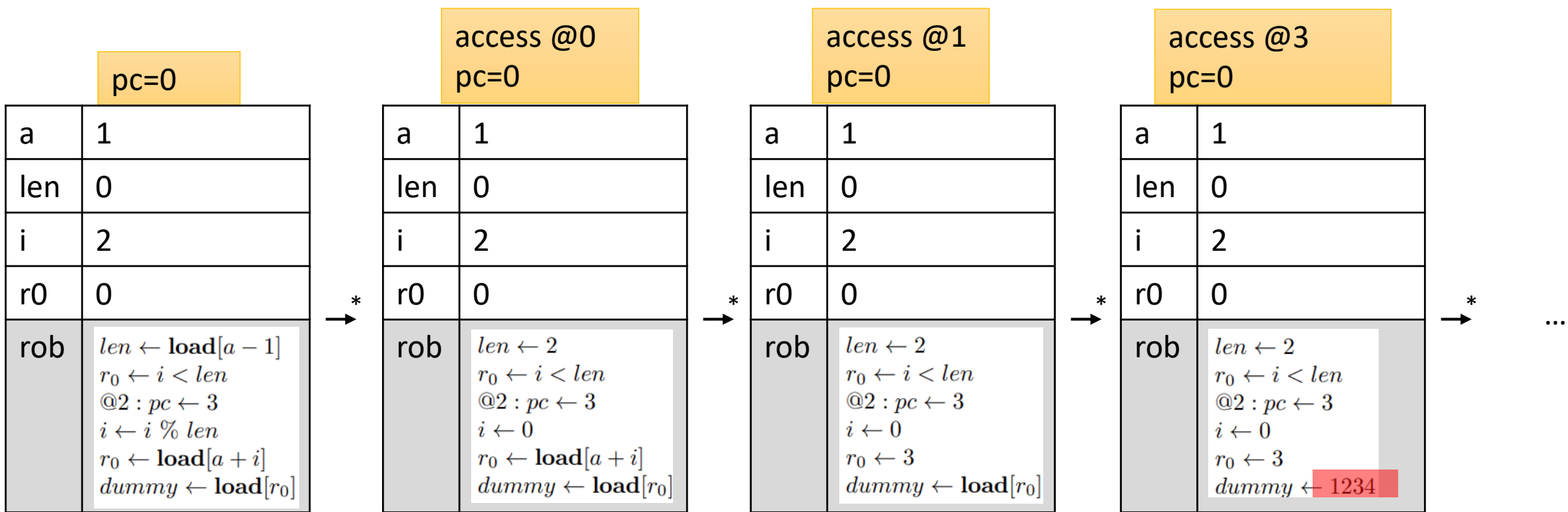
# A hardened version of the program is secure:

```

0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 6          ; if(i < len){
3 : i ← i % len        ; i = i % len
4 : r0 ← load[a + i]   ; r0 = a[i]
5 : leak r0            ; leak(r0)
6 : ...                ; }
    
```

Memory:

	1234:	0
	...	...
	3:	1234
a:	2:	5
	1:	3
len:	0:	2

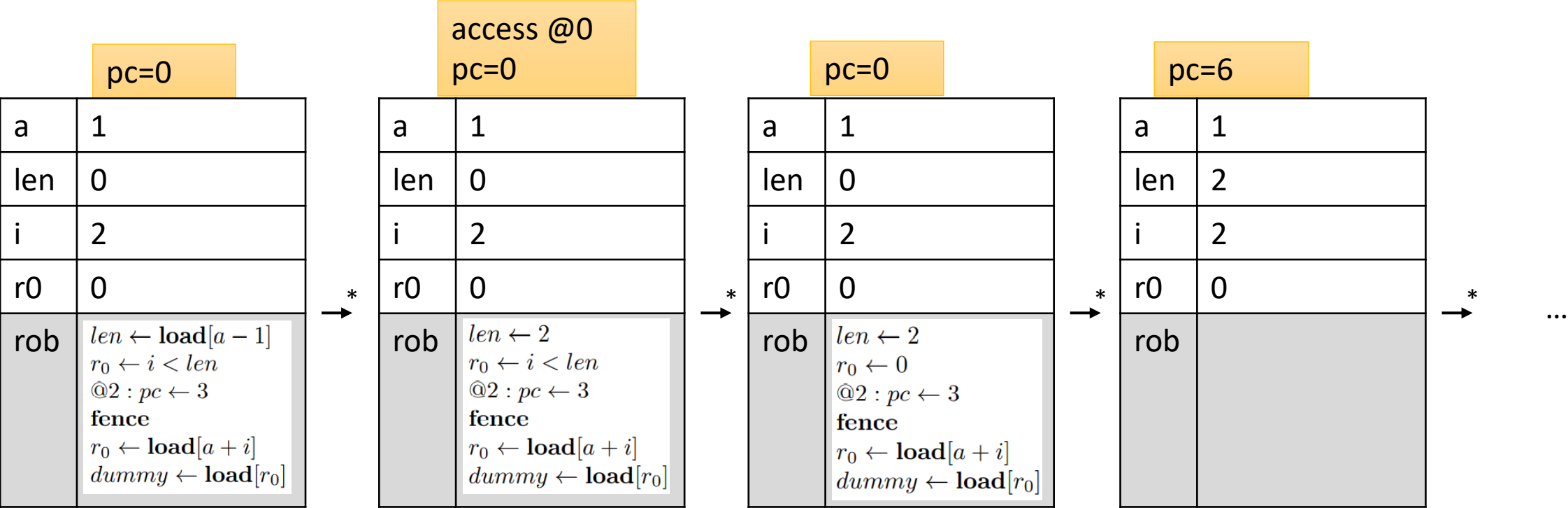


# Hardening with speculation barriers

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 6          ; if(i < len){
3 : fence              ;
4 : r0 ← load[a + i]   ; r0 = a[i]
5 : leak r0            ; leak(r0)
6 : ...                ; }
```

Memory:

	1234:	0
	...	...
	3:	1234
a:	2:	5
	1:	3
len:	0:	2



# A provable defense based on our models

- Let us now show the value of the models by designing a provably secure system
  - Joint work (under submission) with L. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk
- The basic idea is:
  - Software will have to be secure under the constant-time model
  - Hardware will track secrets and stall speculative execution only when secrets could leak
  - Together, these two mitigations lead to an efficient and provably secure system

# Illustrating the defense on Spectre v1

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 5 ; if(i < len){
3 : r0 ← load[a + i] ; r0 = a[i]
4 : leak r0 ; leak(r0)
5 : ... ; }
```

pc=0

a	1
len	0
i	2
r0	0
rob	

→ \*

Memory:

	1234:	0
	...	...
	3:	1234
a:	2:	5
	1:	3
len:	0:	2

# Illustrating the defense on Spectre v1

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 5          ; if(i < len){
3 : r0 ← load[a + i]   ;   r0 = a[i]
4 : leak r0           ;   leak(r0)
5 : ...               ; }
```

pc=0	
a	1
len	0
i	2
r0	0
rob	

→ \*

Code should be constant-time

Memory should be partitioned in public/secret

Memory:

1234:	0
...	...
3:	1234
2:	5
a: 1:	3
len: 0:	2

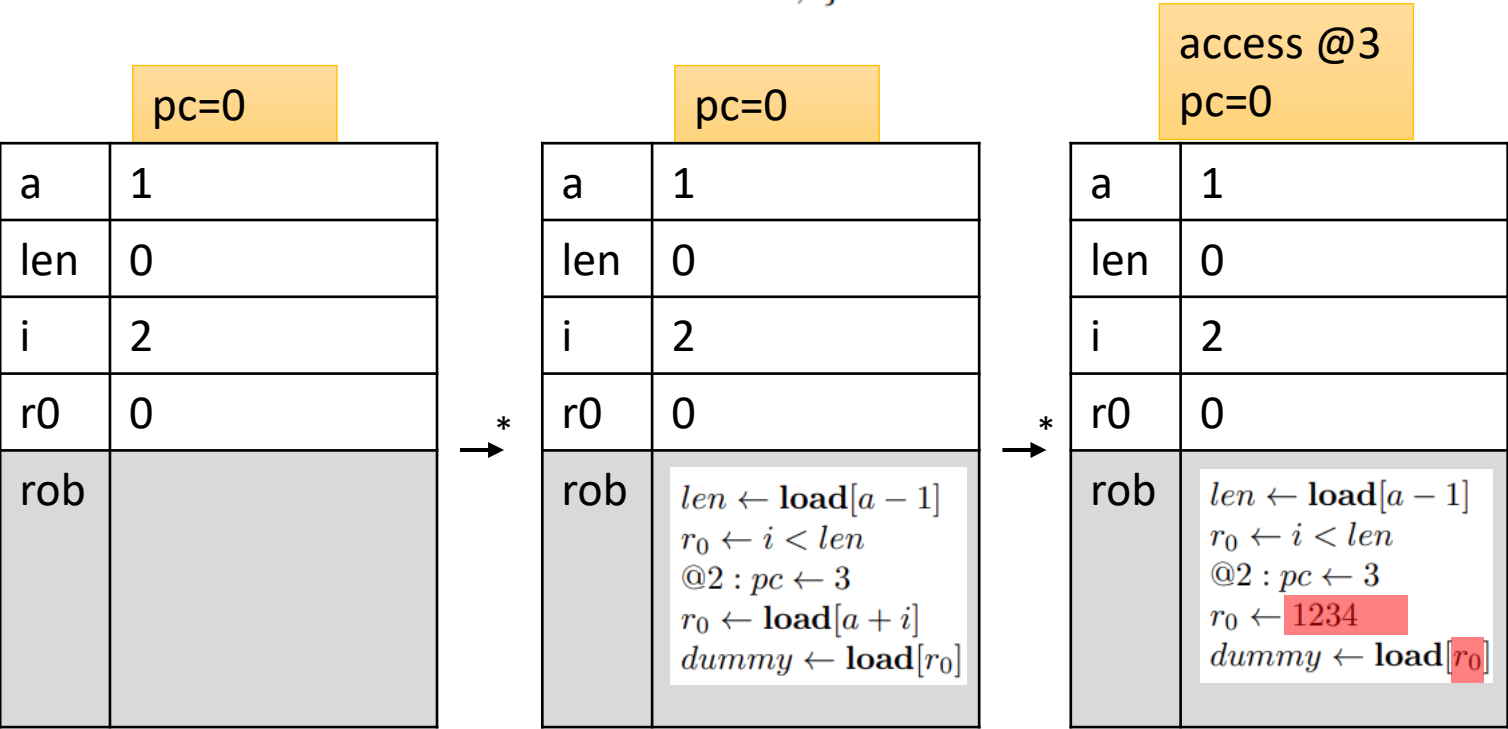


# Illustrating the defense on Spectre v1

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 5          ; if(i < len){
3 : r0 ← load[a + i]   ;   r0 = a[i]
4 : leak r0            ;   leak(r0)
5 : ...                ; }
```

Memory:

	1234:	0
	...	...
	3:	1234
a:	2:	5
	1:	3
len:	0:	2



Microarchitecture should track secret taint, ...

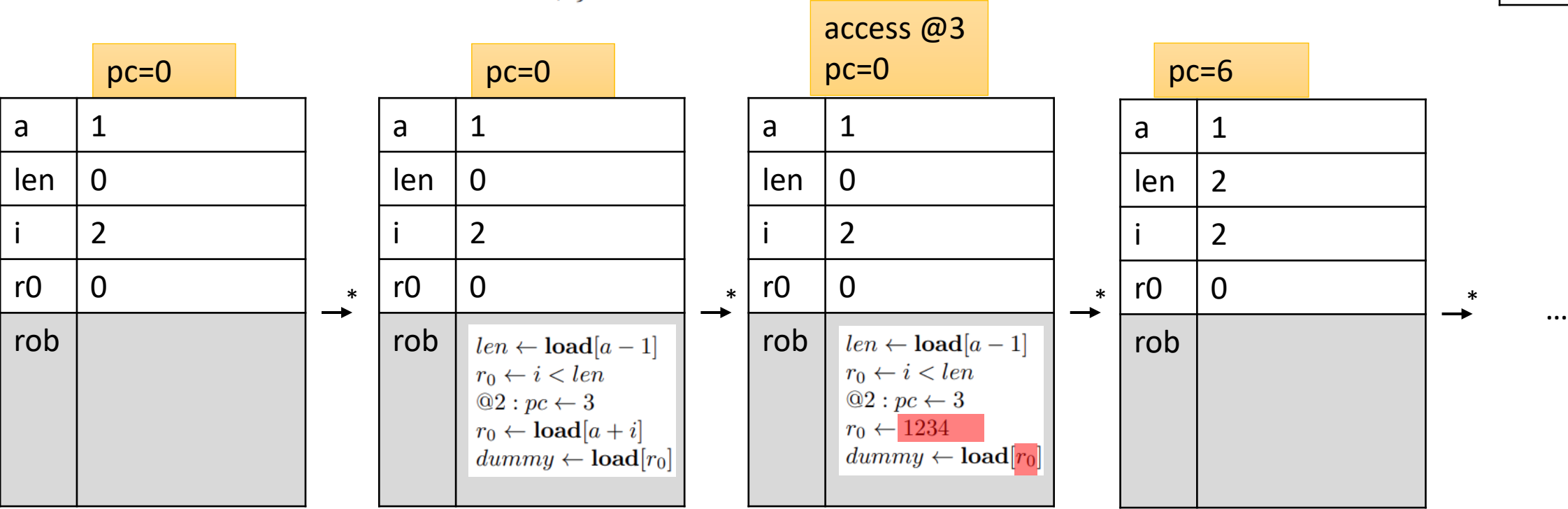
..., and block speculative execution of leaky instructions that depend on secrets

# Illustrating the defense on Spectre v1

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 5          ; if(i < len){
3 : r0 ← load[a + i]   ; r0 = a[i]
4 : leak r0           ; leak(r0)
5 : ...               ; }
```

Memory:

	1234:	0
	...	...
	3:	1234
a:	2:	5
	1:	3
len:	0:	2



# The formal model

We consider two security labels:  $L$  (low, public) and  $H$  (high, secret). Program states are of the form  $\langle m, r, buf, \mu \rangle$  where:

- $m$  is a memory, mapping addresses to values, and addresses are statically labeled as being  $L$  or  $H$ ,
- $r$  is a register map, mapping register names (including  $pc$ ) to a value paired with a security label,
- $buf$  is a reorder buffer, and
- $\mu$  is the microarchitectural *context* that the attacker can observe/influence. The attacker provides an initial  $\mu_0$ , and provides:
  - *update*, which is called with anything the attacker might observe;
  - *predict*, which allows the attacker to control prediction/speculation;
  - *next*, which allows the attacker to choose the next processor step to execute.

STEP

$$\begin{array}{c} \mu' \triangleq \text{update}(\mu, m|_L, r|_L, \lfloor buf \rfloor_L) \\ d \triangleq \text{next}(\mu') \quad \langle m, r, buf, \mu' \rangle \xrightarrow[d]{} \langle m', r', buf', \mu'' \rangle \\ \hline \langle m, r, buf, \mu \rangle \rightarrow \langle m', r', buf', \mu'' \rangle \end{array}$$

FETCH-PREDICT-BRANCH-JMP

$$\begin{array}{c} (1:_) \triangleq \llbracket pc \rrbracket_{apl(buf, r)} \\ P[1] \in \{\text{beqz } e \_, \text{ jmp } e\} \quad 1' \triangleq \text{predict}(\mu) \\ \hline \langle m, r, buf, \mu \rangle \xrightarrow[\text{fetch}]{} \langle m, r, buf \cdot pc \leftarrow (1':L) @ 1, \mu \rangle \end{array}$$

# Example of an end-to-end theorem

- Suppose that the software:
  - Labels memory addresses
  - Is constant-time (does not leak secrets into control flow or addresses)
  - Does not **declassify** secrets
- Then, execution of that software on the modeled processor does not leak secrets to the attacker.

**Theorem 1** (Security for constant-time programs.). *Let  $P$  be a constant-time program that does not declassify secret data. For all number of steps  $n$ , memories  $m_0, m'_0$ , register maps  $r_0, r'_0$ , and microarchitectural state  $\mu_0$ ,*

$$\begin{aligned} m_0|_L = m'_0|_L \wedge r_0|_L = r'_0|_L \wedge \\ \langle m_0, r_0, \epsilon, \mu_0 \rangle \rightarrow^n \langle m_n, r_n, buf_n, \mu_n \rangle \implies \\ \langle m'_0, r'_0, \epsilon, \mu_0 \rangle \rightarrow^n \langle m'_n, r'_n, buf'_n, \mu'_n \rangle \wedge \mu_n = \mu'_n \end{aligned}$$

# Discussion

- Security:
  - This theorem covers all the variants of Spectre we discussed and more
  - The conditions on the software are realistic:
    - Cryptographic software is already complying with the constant-time model today
    - (Compiler will still need to make sure that memory addresses are labeled correctly)
  - The conditions on the hardware are realistic:
    - We have an implementation on an out-of-order RISC-V showing low hardware cost
- Performance:
  - If everything is public, no performance penalty is incurred
  - If software is constant-time, only mis-speculations stall the pipeline
  - Under other conditions, performance cost grows as there are (1) more secrets, (2) used in computations that benefit from speculation

# Overview

- System model
- Microarchitectural side-channel attacks
- Transient execution attacks
- Defenses
- ➔ • Conclusions

# Conclusions

- Transient execution attacks are a fundamentally new class of attacks:
  - That break many important security mechanisms
  - That are not easy to defend against
- Short-term defenses have been useful but ad-hoc
- Long-term defenses are the subject of current research
  - Pure software defenses against Spectre will remain important for the foreseeable future and are the subject of active research.
    - For a recent survey, see:
      - Cauligi et al., SoK: Practical Foundations for Software Spectre Defenses, IEEE S&P 2022
  - Hardware/software co-designs can offer better security/performance trade-offs
    - Excellent starting point for reading more:
      - Guarnieri et al., Hardware/software contracts for secure speculation, IEEE S&P 2021