

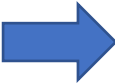
Transient execution attacks and defenses

Frank Piessens

Seminar for the Software Security course

21/04/2022

Overview

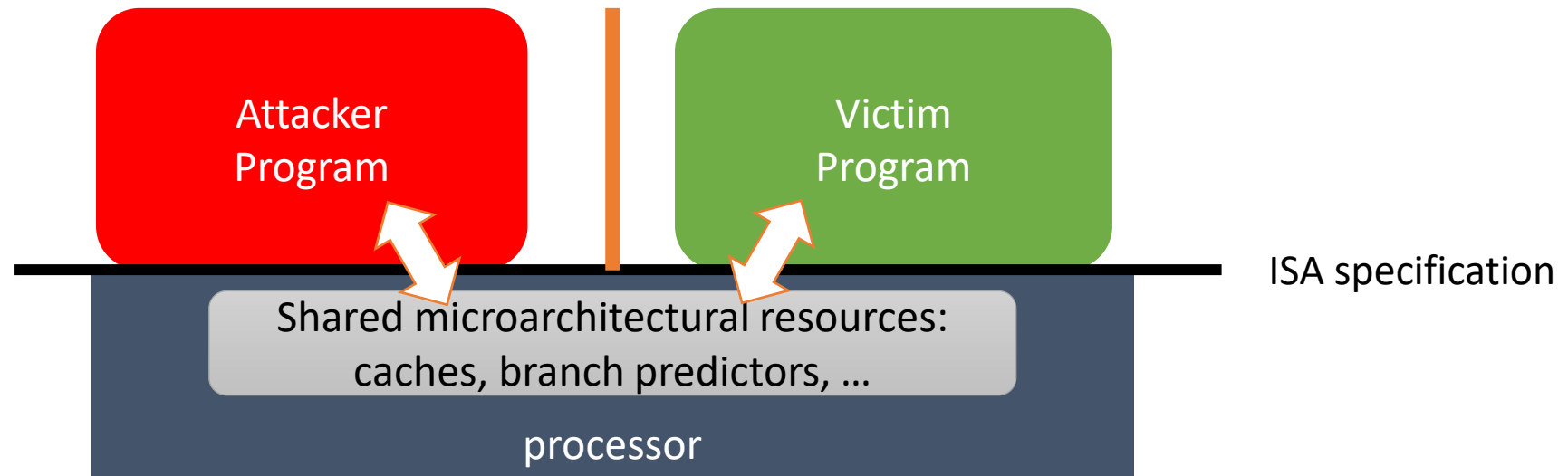
- 
- Introduction
 - A simple Instruction Set Architecture (ISA) model
 - Out-of-order and speculative execution
 - Modeling the attacker
 - Transient execution attacks by example
 - Towards defenses
 - Conclusions

System model: a shared platform

- A platform runs programs from multiple stakeholders
 - Isolation mechanism isolates these programs
 - The platform optionally 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
 - Process isolation, virtual machines, enclaved execution, software-based isolation, ...
 - You have seen examples of such isolation mechanisms in an earlier lecture

Microarchitectural attacks

- Attacker code and victim code run on the same computing platform
 - They are architecturally isolated from each other (e.g., process isolation)
 - But they share microarchitectural resources
 - **Architectural state:** state as defined in the ISA spec (memory, registers, ...)
 - **Microarchitectural state:** additional state in the processor implementation, e.g., for performance improvements (caches, branch predictors, various CPU buffers, ...)



Relevant for security-critical software for a long time

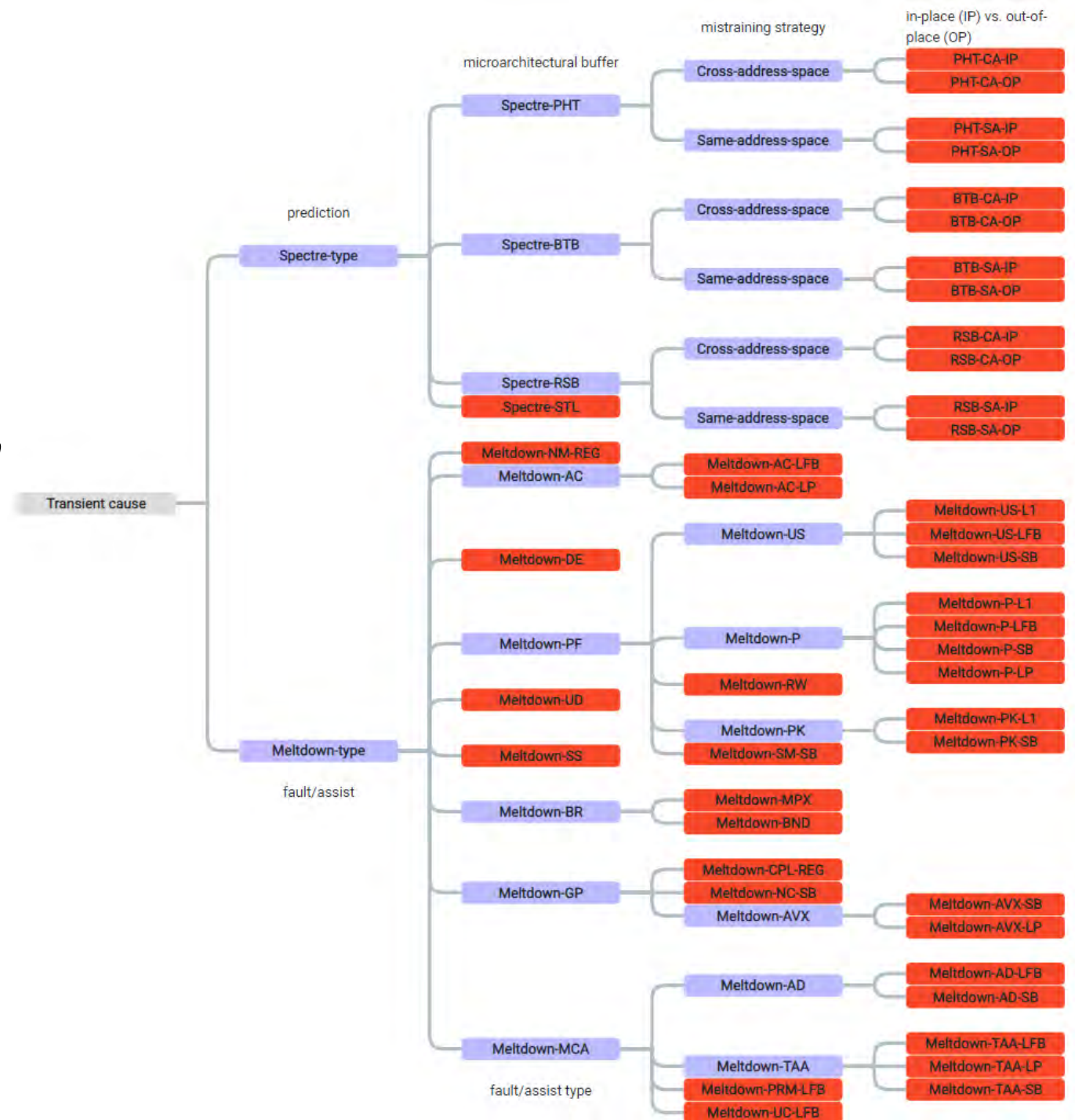
- Instances of microarchitectural side-channel attacks have been known for 15+ years
 - E.g., cache timing attacks that were covered in an earlier lecture
 - Ge et al., *A survey of microarchitectural timing attacks and countermeasures on contemporary hardware*, J. Cryptographic Engineering, 2018
- The crypto community has developed solid countermeasures
 - E.g., constant-time programming
 - Almeida et al., *Verifying Constant-Time Implementations*, USENIX Security 2016

Transient execution attacks changed the game

- Spectre, Meltdown and Foreshadow (all publicly disclosed in 2018) showed how speculative and out-of-order execution significantly amplified the problem of microarchitectural attacks
 - Kocher et al. *Spectre Attacks: Exploiting Speculative Execution*, IEEE S&P 2019
 - Lipp et al. *Meltdown: Reading Kernel Memory from User Space*, USENIX Security 2018
 - Van Bulck et al. *Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution*, USENIX Security 2018
- Many other variants followed: RIDL, ZombieLoad, Fallout, LVI, ...
 - <https://mdsattacks.com/>
 - <https://cpu.fail/>
- Strong academic and industry impact

Many variants

- Classification tree from:
 - Canella et al., *A Systematic Evaluation of Transient Execution Attacks and Defenses*, Usenix Security 2019.
- Further extended and maintained at:
 - <https://transient.fail/>



Academic and industry impact

TITLE	CITED BY	YEAR
Spectre attacks: Exploiting speculative execution P Kocher, J Horn, A Fogh, D Genkin, D Gruss, W Haas, M Hamburg, ... 2019 IEEE Symposium on Security and Privacy (SP), 1-19	1873	2019
Meltdown: Reading Kernel Memory from User Space M Lipp, M Schwarz, D Gruss, T Prescher, W Haas, A Fogh, J Horn, ... 27th USENIX Security Symposium (USENIX Security 18)	1603 *	2018
Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution J Van Bulck, M Minkin, O Weisse, D Genkin, B Kasikci, F Piessens, ... 27th USENIX Security Symposium (USENIX Security 18)	810	2018

- 30+ CVE's in the 2017-2021 timeframe
- Significant efforts by technology giants to mitigate the risks

Towards a principled understanding

- Both the discovery of vulnerabilities as well as the development of countermeasures has been a productive but chaotic process
 - Academics compete for finding issues first
 - Companies protect their customers through embargos and ad-hoc countermeasures
- But this class of vulnerabilities is important enough to deserve a systematic foundational study
 - How should we model processors to reason about microarchitectural vulnerabilities, attacks and countermeasures? What is the exact security objective?
 - This seminar will follow the *language-based* approach, of which many instances exist:
 - Mcilroy et al., Spectre is here to stay: An analysis of side-channels and speculative execution, arXiv 2019.
 - Disselkoen et al., The Code That Never Ran: Modeling Attacks on Speculative Evaluation, IEEE S&P 2019.
 - Cauligi et al., Constant-time foundations for the new Spectre era, PLDI 2020.
 - Guanciale et al., InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis, CCS 2020.
 - Guarnieri et al., Hardware/software contracts for secure speculation, IEEE S&P 2021.
 - (this list is not complete)

Overview

- Introduction
- ➔ • A simple Instruction Set Architecture (ISA) model
- Out-of-order and speculative execution
- Modeling the attacker
- Transient execution attacks by example
- Towards defenses
- Conclusions

A simple Instruction Set Architecture (ISA) model

Register names	$r \in \text{Regs}$	<i>(E.g., r_0, r_1, i, len. We assume $pc \notin \text{Regs}$)</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 e to r)</i>
	$r \leftarrow \mathbf{load}[e]$	<i>(load value at memory address e into r)</i>
	$\mathbf{store}[e] \leftarrow r$	<i>(store r in memory at address e)</i>
	$\mathbf{jmp} \ e$	<i>(jump to code address e)</i>
	$\mathbf{beqz} \ r \ v$	<i>(branch to v if r 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 :  $r_0 \leftarrow \mathbf{load}[a + i]$  ;  $sum = sum + a[i]$   
3 :  $sum \leftarrow sum + r_0$  ;  
4 :  $i \leftarrow i + 1$  ;  $i = i + 1$   
5 : jmp 0 ; }
```

A simple Instruction Set Architecture (ISA) model

Register names	$r \in \text{Regs}$	<i>(E.g., r_0, r_1, i, len. We assume $pc \notin \text{Regs}$)</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 e to r)</i>
	$r \leftarrow \mathbf{load}[e]$	<i>(load value at memory address e into r)</i>
	$\mathbf{store}[e] \leftarrow r$	<i>(store r in memory at address e)</i>
	$\mathbf{jmp} \ e$	<i>(jump to code address e)</i>
	$\mathbf{beqz} \ r \ v$	<i>(branch to v if r 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 :  $r_0 \leftarrow \mathbf{load}[a + i]$  ;  $sum = sum + a[i]$ 
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

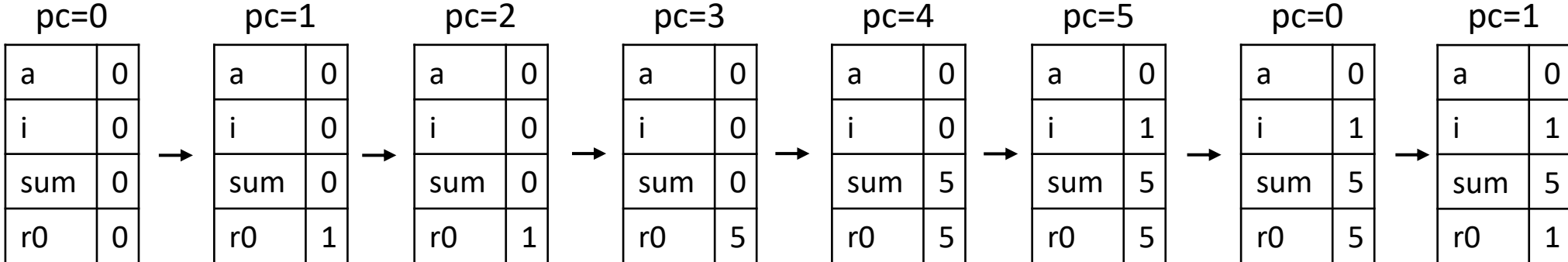
Base semantics

Register state $\rho \in \text{Regs} \rightarrow \text{Values}$ *(mapping from register names to values)*
 Memory state $m ::= \vec{v}$ *(list of values)*
 Program counter $pc ::= v$ *(an index into the program)*
 Program state $\sigma ::= (m, \rho, pc)$

Program:

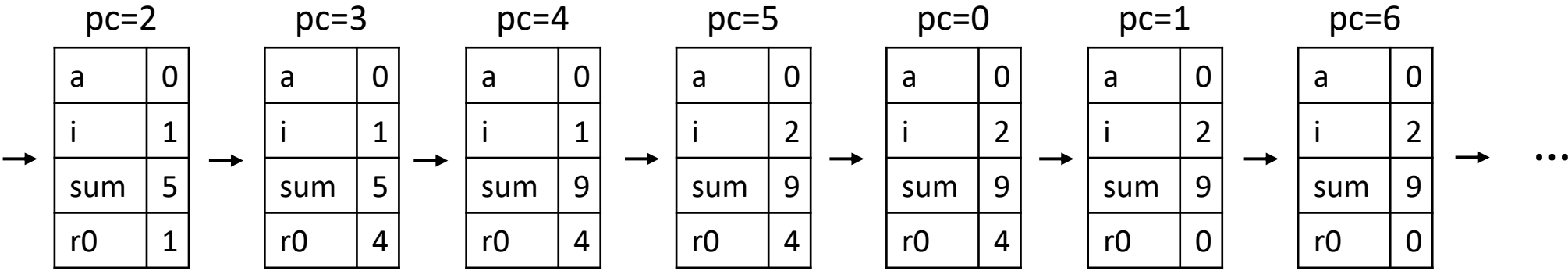
```

0 : r0 ← i < 2
1 : beqz r0 6
2 : r0 ← load[a + i]
3 : sum ← sum + r0
4 : i ← i + 1
5 : jmp 0
    
```



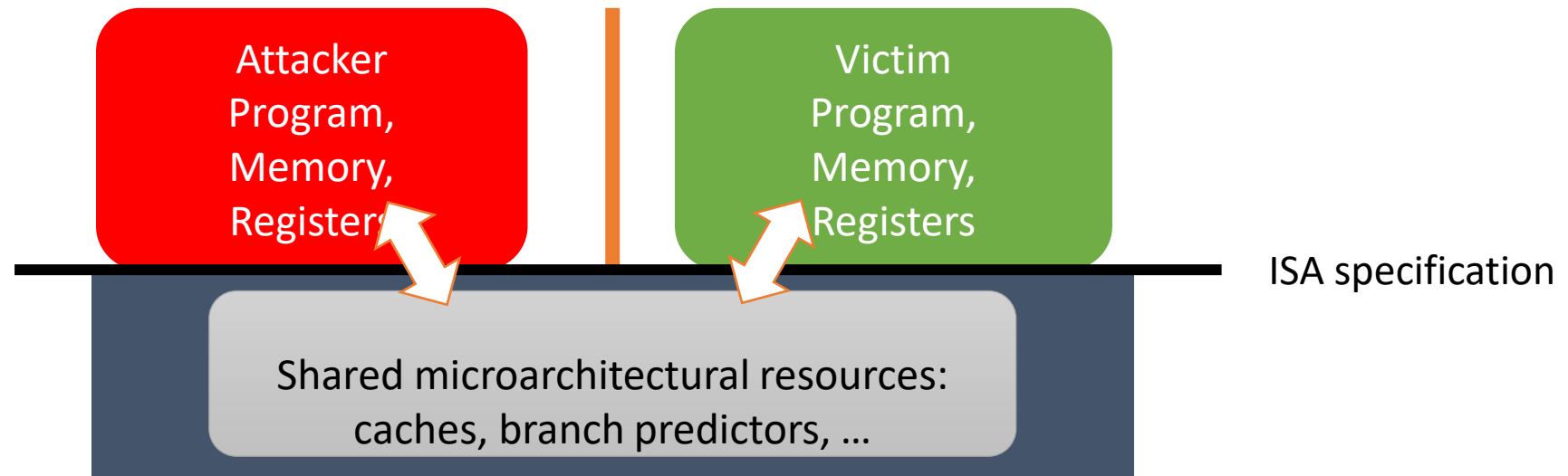
Memory:

...	
1:	4
0:	5




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



Overview

- Introduction
- A simple Instruction Set Architecture (ISA) model
-  • Out-of-order and speculative execution
- Modeling the attacker
- Transient execution attacks by example
- Towards 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

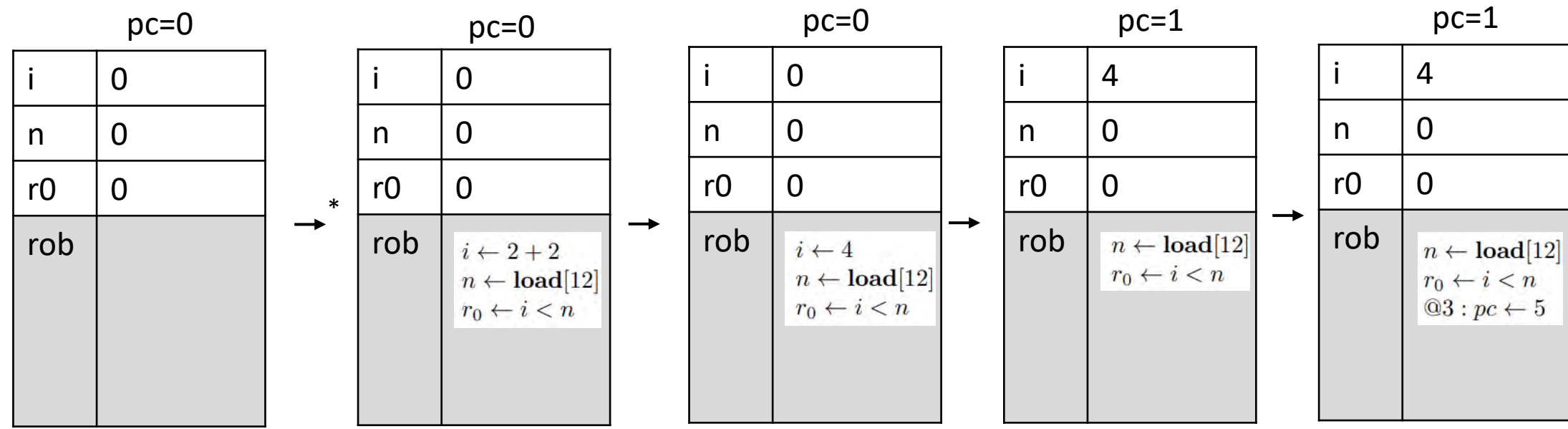
In-flight instructions $f ::= r \leftarrow e$
 $r \leftarrow \mathbf{load}[e]$
 $\mathbf{store}[e] \leftarrow r$
 $pc \leftarrow v$
 $@v : pc \leftarrow v$
 $@v : r \leftarrow v$

Reorder buffer $rob ::= \vec{f}$

Program state $\sigma ::= (m, \rho, pc, rob)$

(non-speculated jump becomes pc assignment)
(speculated jump, v is address of original instruction)
(speculated load, v is address of original instruction)

0 : $i \leftarrow 2 + 2$
 1 : $n \leftarrow \mathbf{load}[12]$
 2 : $r_0 \leftarrow i < n$
 3 : $\mathbf{beqz} r_0 5$
 4 : $i \leftarrow 4 \times i$
 5 : $i \leftarrow i + 1$

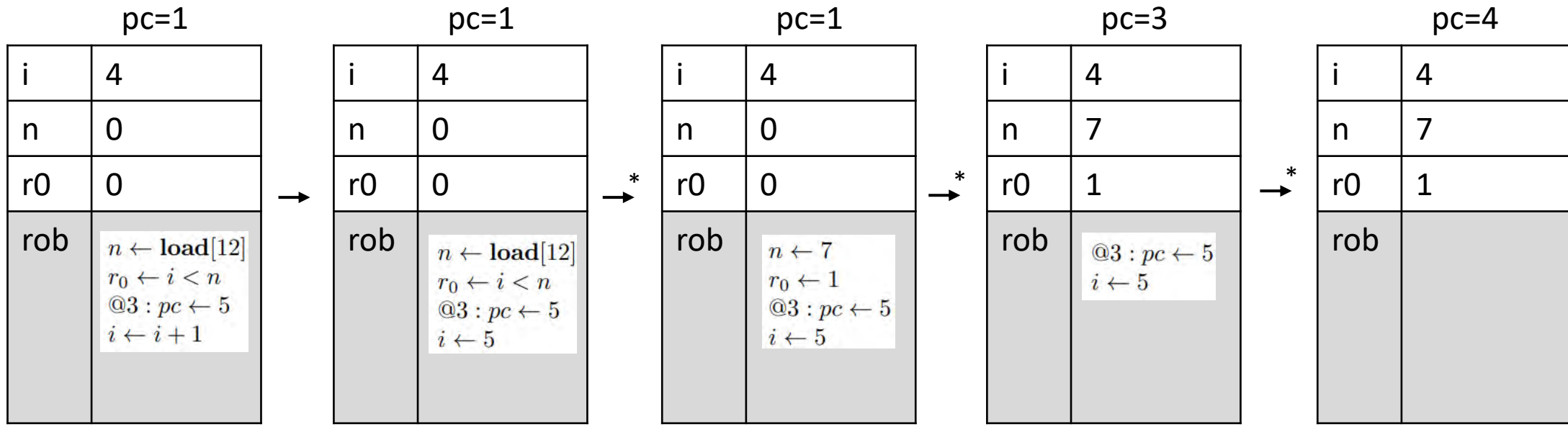


```

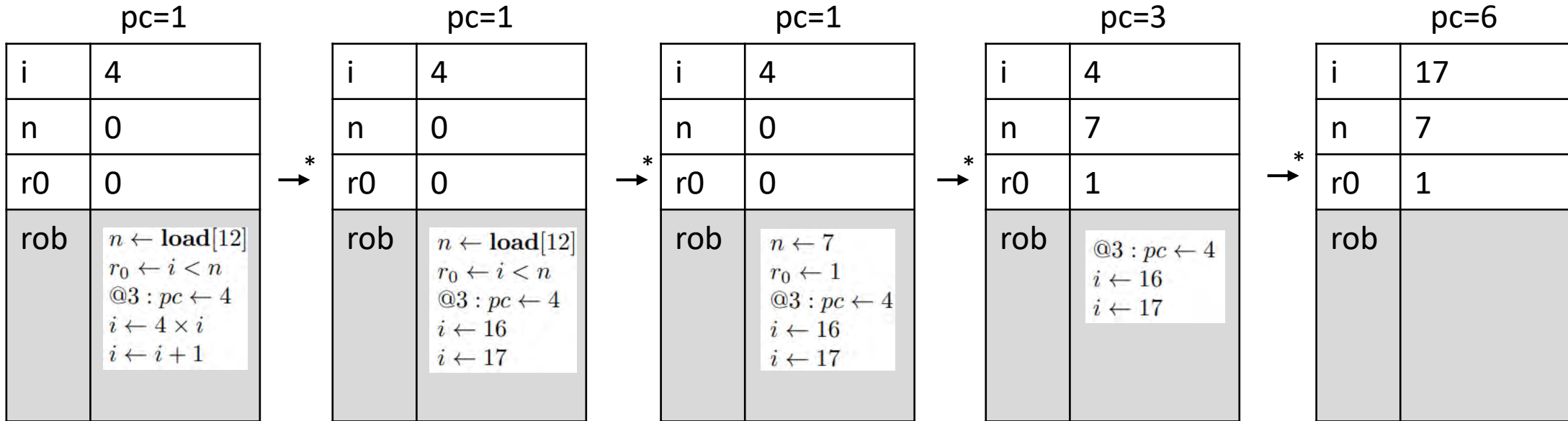
0 :  $i \leftarrow 2 + 2$ 
1 :  $n \leftarrow \text{load}[12]$ 
2 :  $r_0 \leftarrow i < n$ 
3 : beqz  $r_0$  5
4 :  $i \leftarrow 4 \times i$ 
5 :  $i \leftarrow i + 1$ 

```

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?

Overview

- Introduction
- A simple Instruction Set Architecture (ISA) model
- Out-of-order and speculative execution
-  • Modeling the attacker
- Transient execution attacks by example
- Towards defenses
- Conclusions

Attacker model

- Transient execution attacks build on:
 - Classic microarchitectural side-channel and covert-channel attacks,
 - For instance, cache attacks, but there are many more
 - The fact that the attacker can influence the speculative and out-of-order execution
 - For instance, by “training” the branch predictor, but there are many other ways
- To make reasoning about these attacks manageable yet secure, we **overapproximate and simplify**
 - The “constant-time leakage model” models what an attacker can learn through classic side-channels
 - We give the attacker full control over predictions and scheduling
 - Note that this significantly simplifies attack examples!
 - Doing the example attacks we will discuss on a real system can be very labor-intensive

The constant time leakage model

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

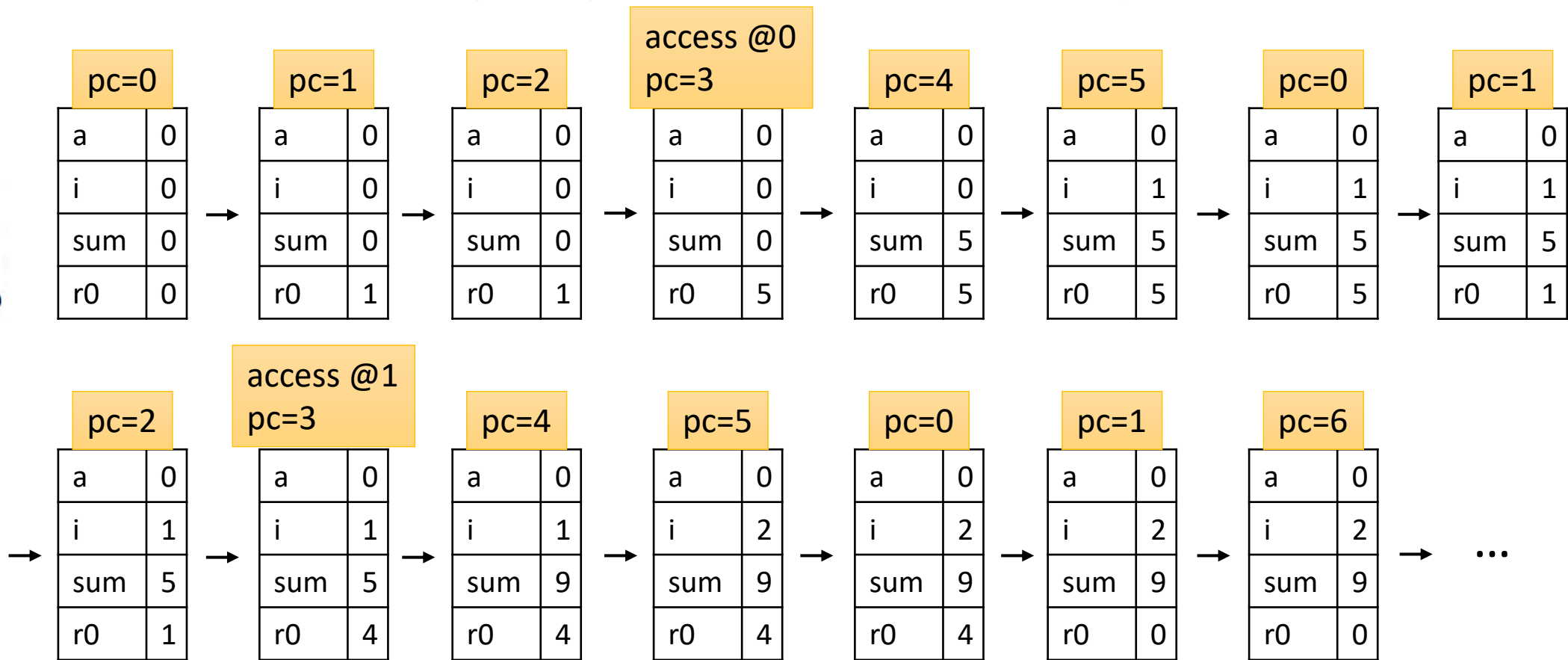
Program:

```

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

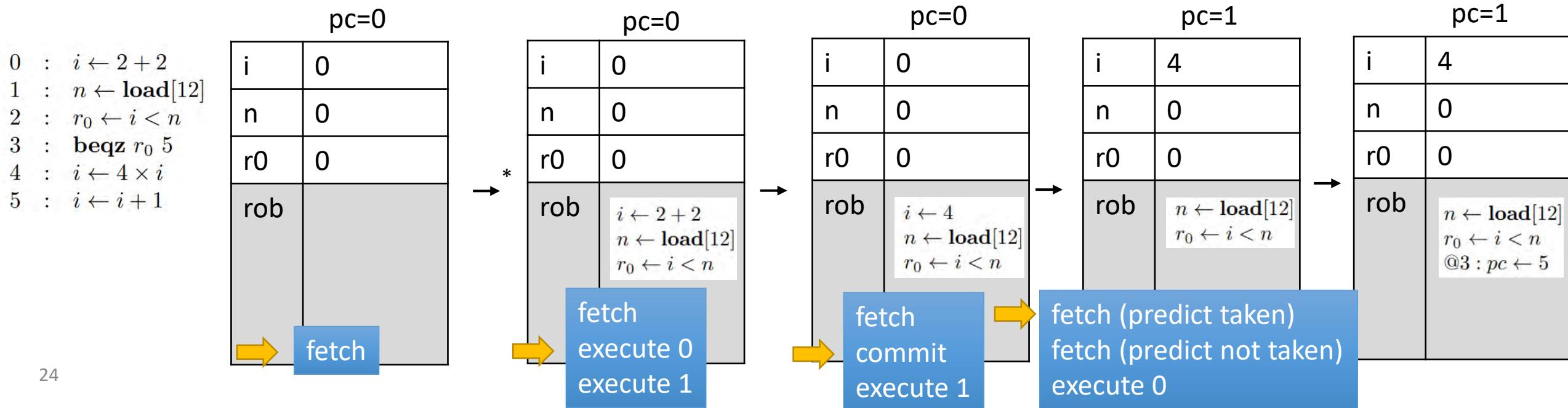
- In later attacks, we rely on code snippets that leak secrets through a microarchitectural side-channel
 - A wide variety of such snippets exist
 - In some scenarios, the attacker can construct them, in other scenarios the attacker has to find them in victim code
- For simplicity, we will define:

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

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

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



Overview

- Introduction
- A simple Instruction Set Architecture (ISA) model
- Out-of-order and speculative execution
- Modeling the attacker
- ➔ • Transient execution attacks by example
- Towards defenses
- Conclusions

Modeling 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

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 “classic” side-channels
 - For instance, all the examples satisfy the constant-time coding discipline
 - Yet, because of speculation and out-of-order execution, the secrets **do** leak

Example 1: Spectre v1 (Spectre-PHT)

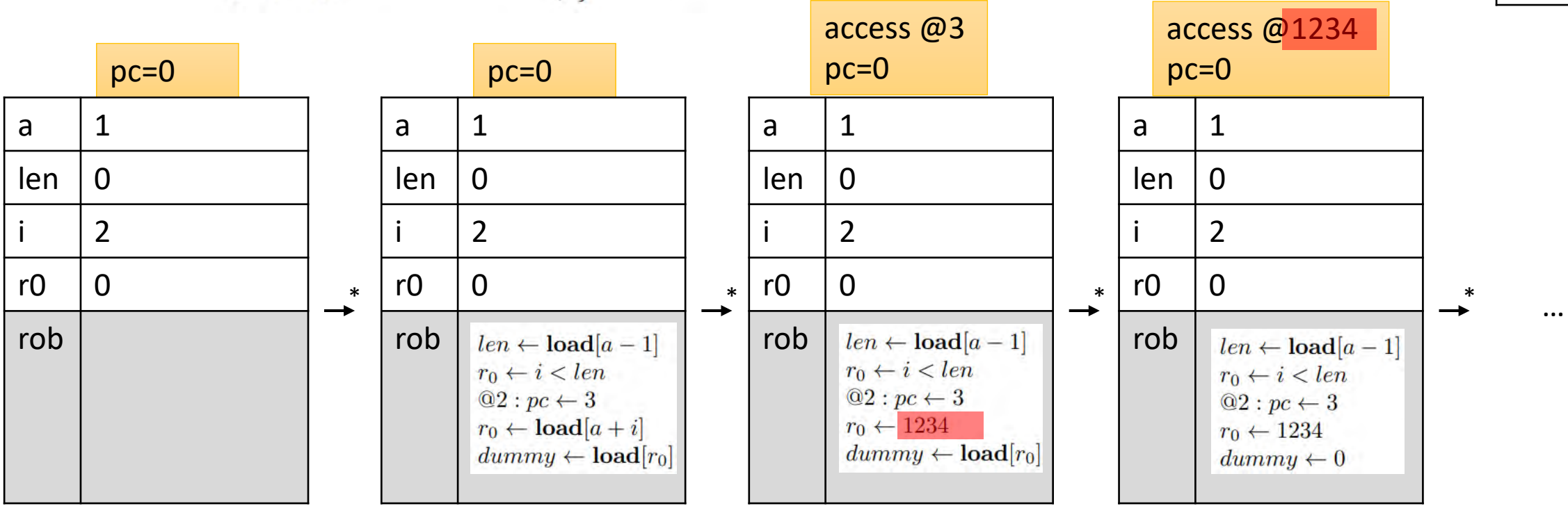
```

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



Example 2: Spectre v2 (Spectre-BTB)

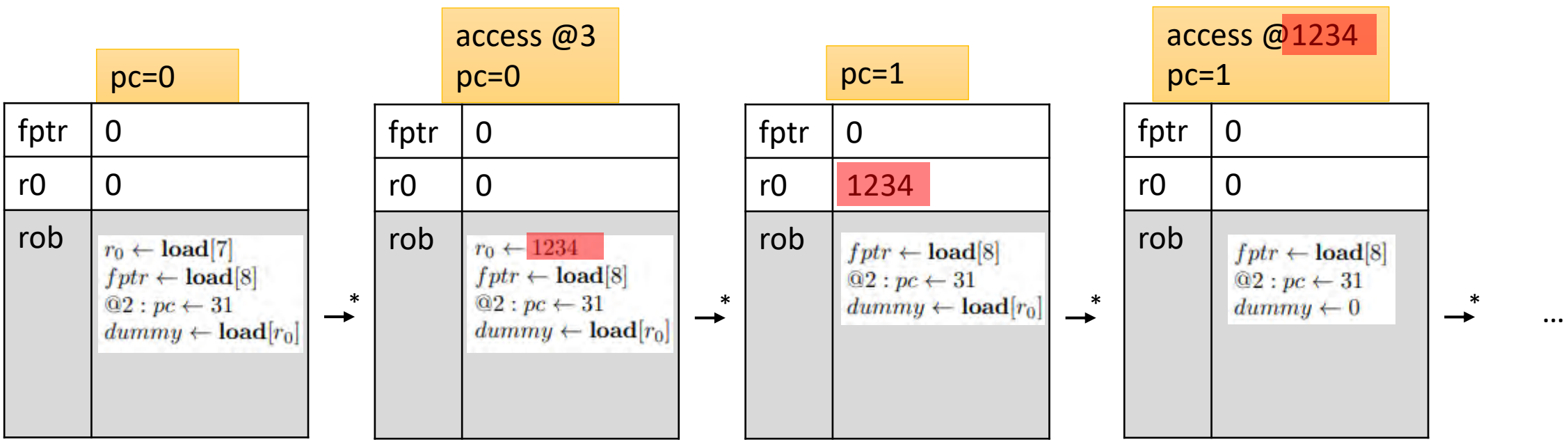
```

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)

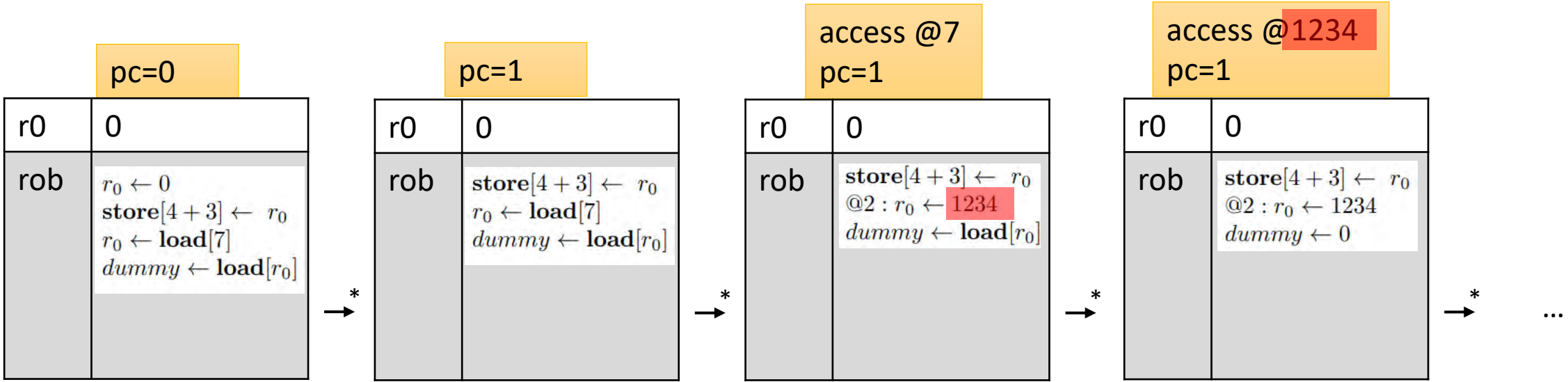
Memory:

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 : r0 ← load[7] ; load address 7 into r0, should read 0
3 : leak r0
... : ...
    
```



Transient execution attacks

- These were a couple of **simplified Spectre attacks**
 - See <https://transient.fail/> for more variants and more details
- Note the **devastating** nature of this kind of attack on software-enforced confidentiality properties

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 Evtushkin, 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 Evtushkin, Ryan Riley, Nael Abu-Ghazaleh, Dmitry Ponomarev (*ASPLOS 2018*)
- [The microarchitecture of Intel, AMD and VIA CPUs](#)
Agner Fog

Meltdown example: faulting loads

- Surprisingly, memory loads that raise a fault (e.g., page fault or protection fault) still execute transiently (on some processors)
- The essence of Meltdown:

```
r0 ← load[kernel_address] ; this load will raise a fault  
leak r0
```

- Later papers have shown that faulting loads (or loads that receive “microcode assists”) compute transiently on all kinds of potentially sensitive data

See again transient.fail for an overview

Meltdown-US-L1 (aka Meltdown)



The original Meltdown attack reads cached kernel memory from user space on CPUs that do not transiently enforce the user/supervisor flag. In the trigger phase an unauthorized kernel address is dereferenced, which eventually causes a page fault. Before the fault becomes architecturally visible, however, the attacker executes a transient instruction sequence that for instance accesses a cache line based on the privileged data read by the trigger instruction. In the final phase, after the exception has been raised, the privileged data is reconstructed at the receiving end of the covert channel (e.g., Flush+Reload).

References

- [Meltdown: Reading Kernel Memory from User Space](#)
Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg (*USENIX Security 2018*)
- [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 Evtushkin, Daniel Gruss (*USENIX Security 2019*)

Known to be affected

- [Intel](#)
- [ARM](#)
- [IBM](#)

Also known as

- [Meltdown](#)
- [Rogue Data Cache Load \(RDCL\)](#)
- [Variant 3](#)

Meltdown-P-L1 (aka Foreshadow)



Meltdown-P-L1 exploits an “L1 Terminal Fault” (L1TF) microarchitectural condition when accessing unmapped pages. A terminal page fault occurs when accessing a page-table entry with either the “present” bit cleared or a “reserved” bit set. In such cases, the CPU immediately aborts address translation. However, since the L1 data cache is indexed in parallel to address translation, the page table entry’s physical address field (i.e., frame number) may still be passed to the L1 cache. Any data present in L1 and tagged with that physical address will now be forwarded to the transient execution, regardless of access permissions.

Foreshadow was initially demonstrated against Intel SGX technology, and a generalized form of the attack allows an attacker to bypass operating system or hypervisor isolation. This variation allows an untrusted virtual machine, controlling guest-physical addresses, to extract the host machine’s entire L1 data cache (including data belonging to the hypervisor or other virtual machines). The underlying problem is that a terminal fault in the guest page-tables early-outs the address translation process, such that guest-physical addresses are erroneously passed to the L1 data cache, without first being translated into a proper host physical address.

References

- [Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution](#)
Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, Raoul Strackx (*USENIX Security 2018*)
- [Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution](#)
Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, Yuval Yarom
- [A Systematic Evaluation of Transient Execution Attacks and Defenses](#)

Meltdown-P-LFB (aka RIDL)



RIDL leaks in-flight data from the line-fill buffer (LFB) by exploiting faulting loads on non-present addresses. If the least-significant 6 bits of the non-present virtual address match a virtual address of data currently stored in the LFB, then this data can be leaked. Any data travelling between the L1 cache and the remaining memory subsystem has to go through the LFB and can be leaked with RIDL.

References

- [RIDL: Rogue In-flight Data Load](#)
Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida (*IEEE S&P 2019*)
- [ZombieLoad: Cross-Privilege-Boundary Data Sampling](#)
Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, Daniel Gruss (*ACM CCS 2019*)
- [Deep Dive: Intel Analysis of Microarchitectural Data Sampling](#)
Intel

Known to be affected

- [Intel](#)

Also known as

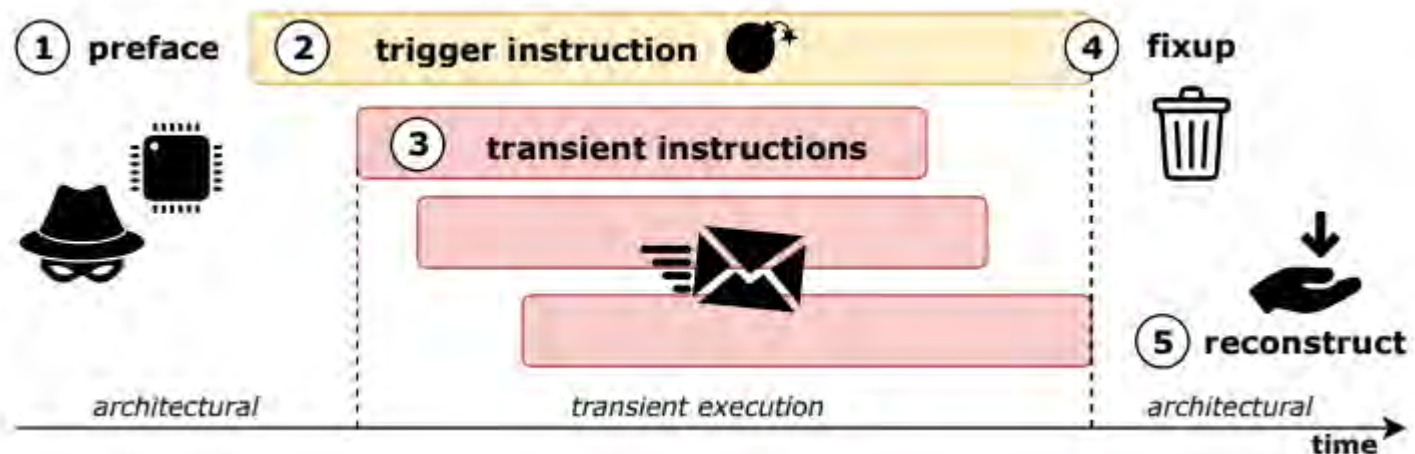
- [Rogue In-flight Data Load \(RIDL\)](#)
- [Microarchitectural Fill Buffer Data Sampling \(MFBDS\)](#)
- [Microarchitectural Data Sampling \(MDS\)](#)

CVE

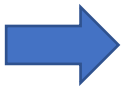
- [CVE-2018-12130](#)

General transient execution attack structure

1. Prime the micro-architectural state
2. Trigger transient execution (misprediction or fault)
3. Send on the covert channel
4. CPU flushes architectural effects of transient execution
5. Read from the covert channel



Overview

- Introduction
- A simple Instruction Set Architecture (ISA) model
- Out-of-order and speculative execution
- Modeling the attacker
- Transient execution attacks by example
-  • Towards defenses
- Conclusions

Defenses

- Defenses are being investigated at multiple levels:
 - Hardware mitigations
 - For instance, do not forward values from faulting loads to subsequent instructions
 - Mitigations in the Operating System
 - For instance, do not place the kernel in the same virtual address space as user code
 - Mitigations in the compiler
 - For instance, insert instructions to stop out-of-order execution, or rewrite code to remove the vulnerability
- Meltdown-style vulnerabilities are being addressed in hardware
- For Spectre-style vulnerabilities, good defenses are still the subject of ongoing research

Security objective of defenses

- Transient execution attacks cause unexpected information flows, and hence the security of a program against these attacks can be defined using techniques from information flow security
- 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:

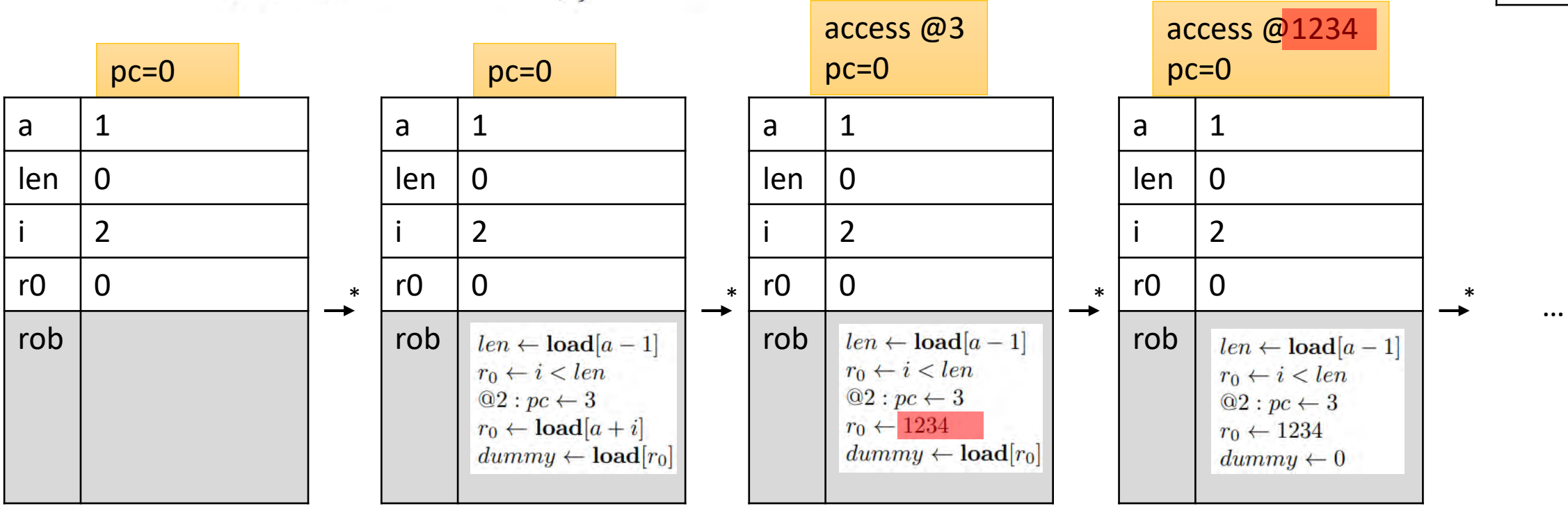
```

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



A hardened version of the program is secure:

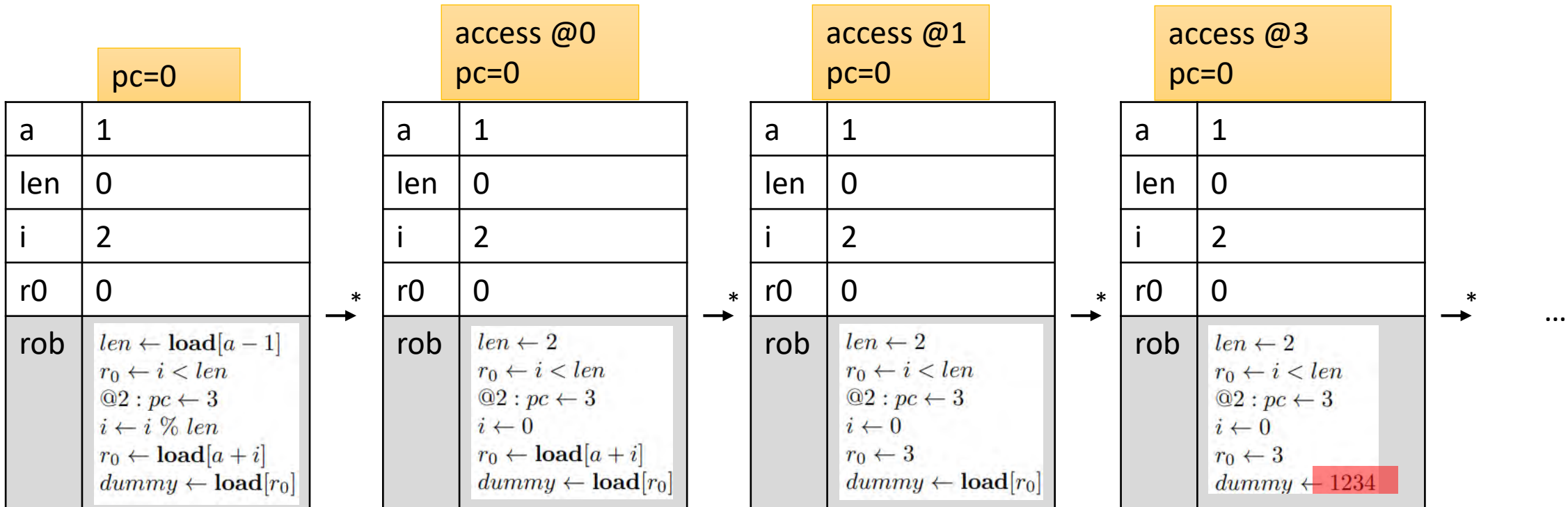
```

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



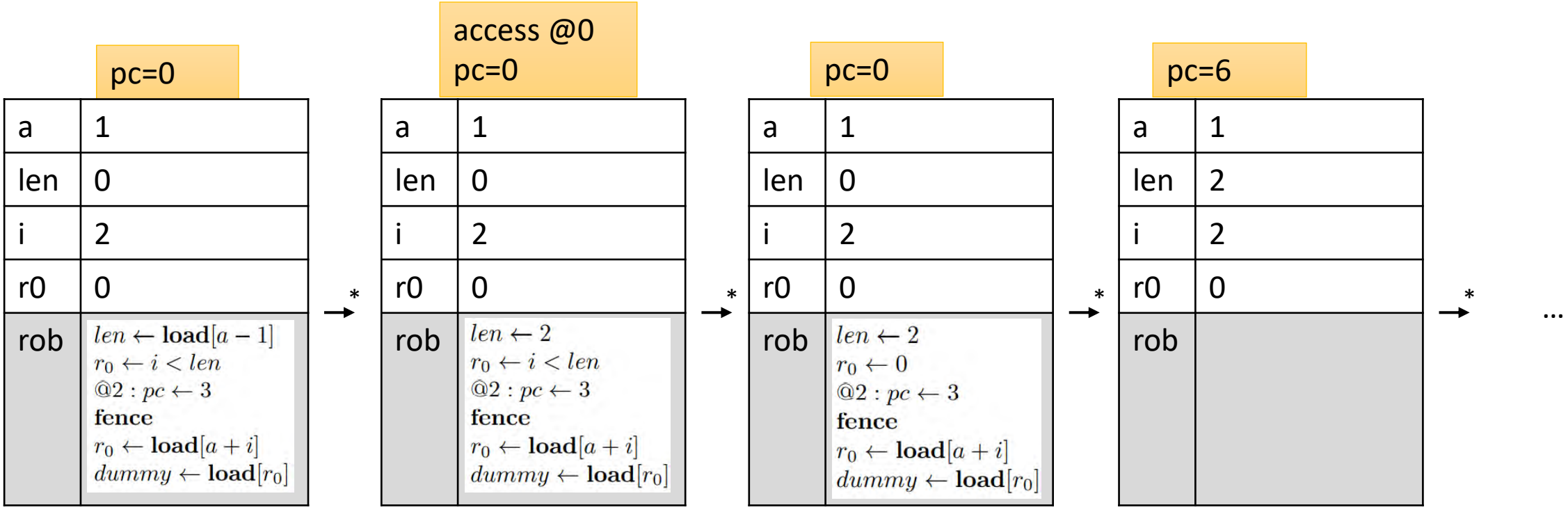
Hardening with speculation barriers

```

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

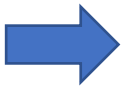


Different scenarios for defenses

- Attacks can be decomposed in two steps:
 1. **Accessing** the secret
 2. **Leaking** it through a microarchitectural channel
- Three scenarios:
 - Access and leak are both non-transient
 - This is a classic side-channel attack, and it can be defended, e.g., with CT programming
 - Access is non-transient, leak is transient
 - The “secure programming” scenario, for instance cryptographic code
 - Can we extend the CT programming discipline to also close these leaks?
 - Both access and leak are transient
 - The “sandboxing” scenario: victim host program executes attacker code using software-based isolation
 - Typical example: a browser running WebAssembly code
 - Sufficient condition for security: hardware ensures that transiently accessed data does not leak

Overview

- Introduction
- A simple Instruction Set Architecture (ISA) model
- Out-of-order and speculative execution
- Modeling the attacker
- Transient execution attacks by example
- Towards 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