Software security across abstraction layers

Frank Piessens
Department of Computer Science
KU Leuven

Marktoberdorf Summer school
August 2024

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

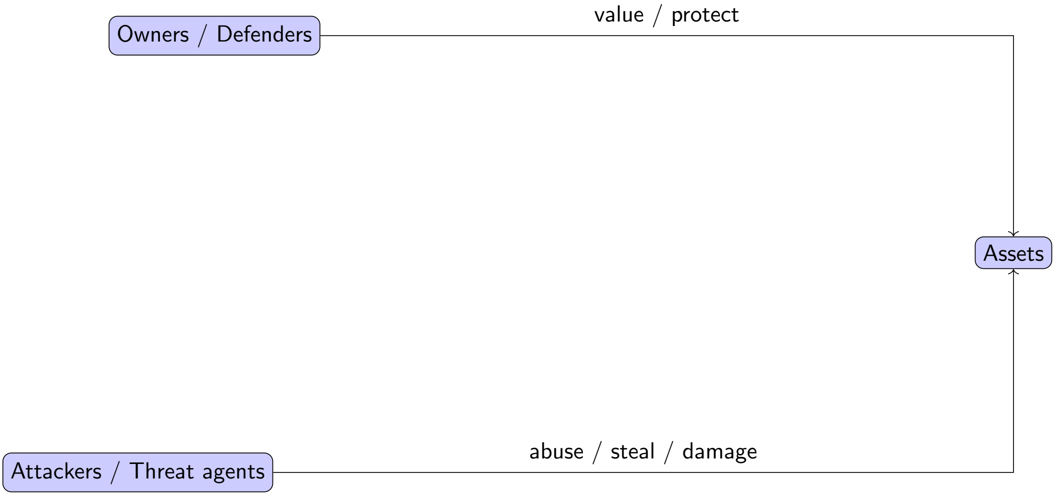
4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Security concepts



Owners / Defenders

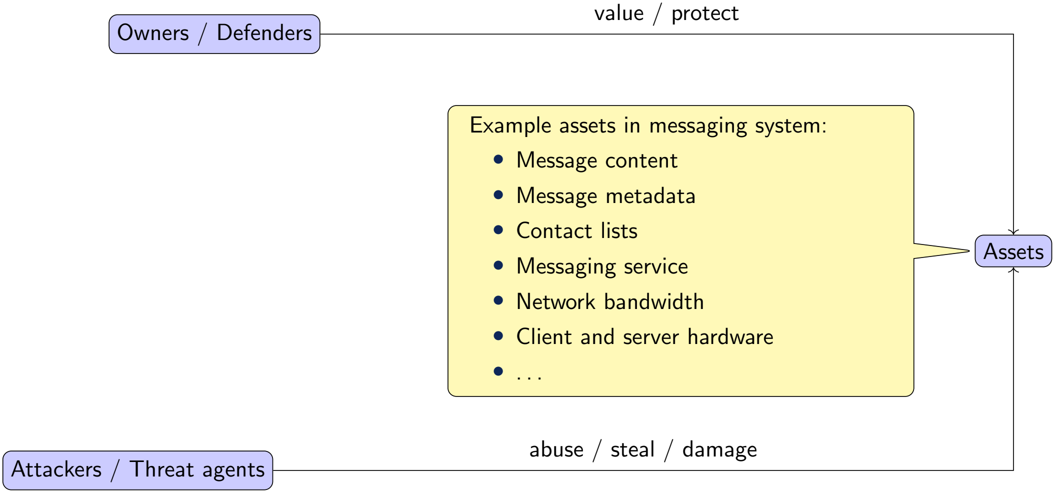
value / protect

Assets

abuse / steal / damage

Attackers / Threat agents

Security concepts



Owners / Defenders

value / protect

Assets

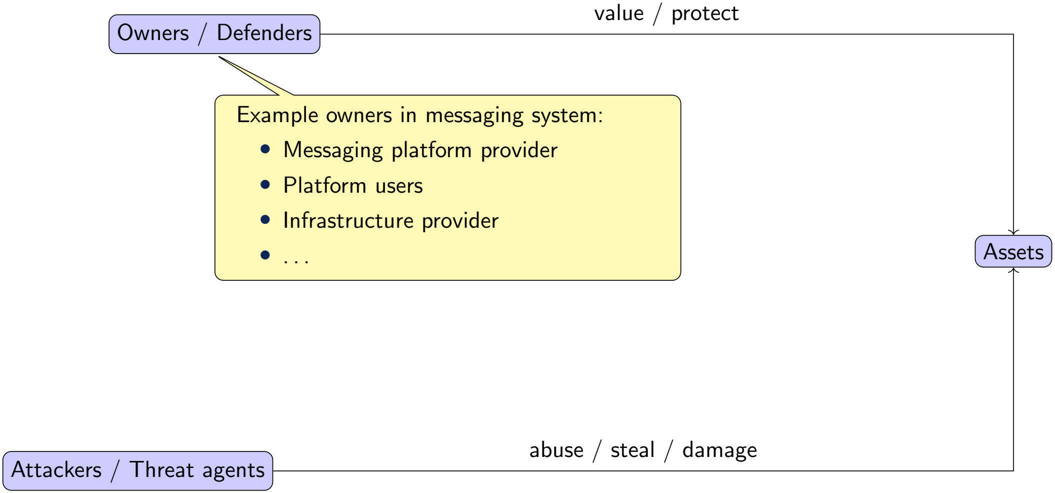
abuse / steal / damage

Attackers / Threat agents

Example assets in messaging system:

- Message content
- Message metadata
- Contact lists
- Messaging service
- Network bandwidth
- Client and server hardware
- . . .

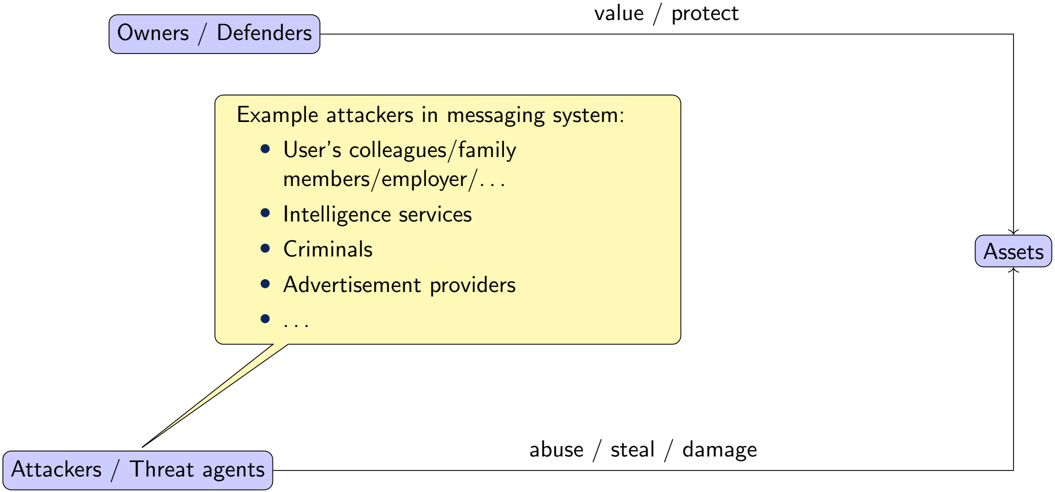
Security concepts



Example owners in messaging system:

- Messaging platform provider
- Platform users
- Infrastructure provider
- . . .

Security concepts



Owners / Defenders

value / protect

Assets

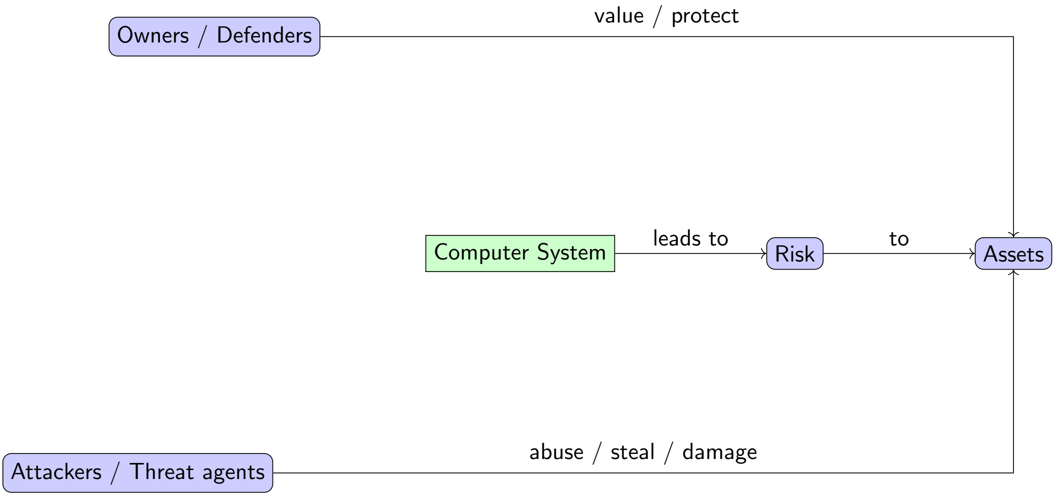
abuse / steal / damage

Attackers / Threat agents

Example attackers in messaging system:

- User's colleagues/family members/employer/...
- Intelligence services
- Criminals
- Advertisement providers
- ...

Security concepts


Owners / Defenders

value / protect

Computer System

leads to

Risk

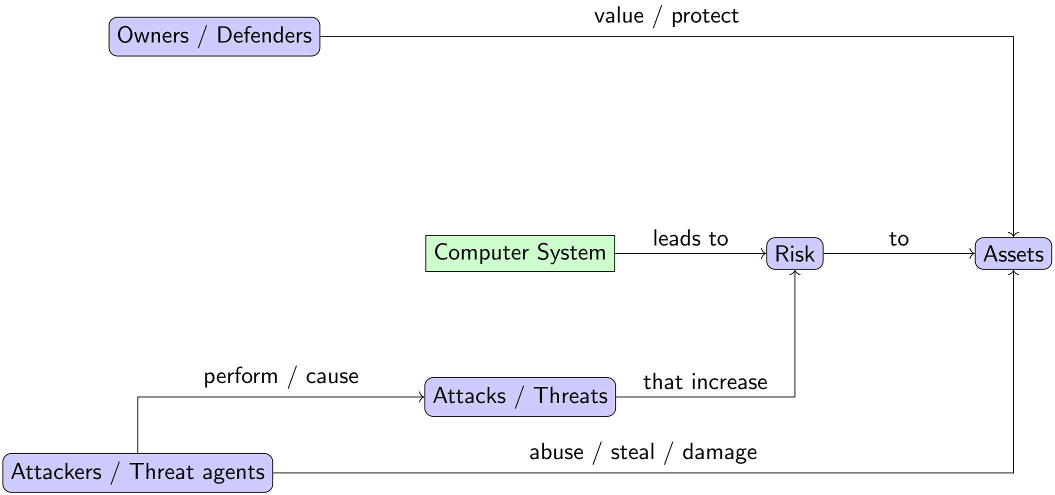
to

Assets

Attackers / Threat agents

abuse / steal / damage

Security concepts



Owners / Defenders

value / protect

Computer System

leads to

Risk

to

Assets

perform / cause

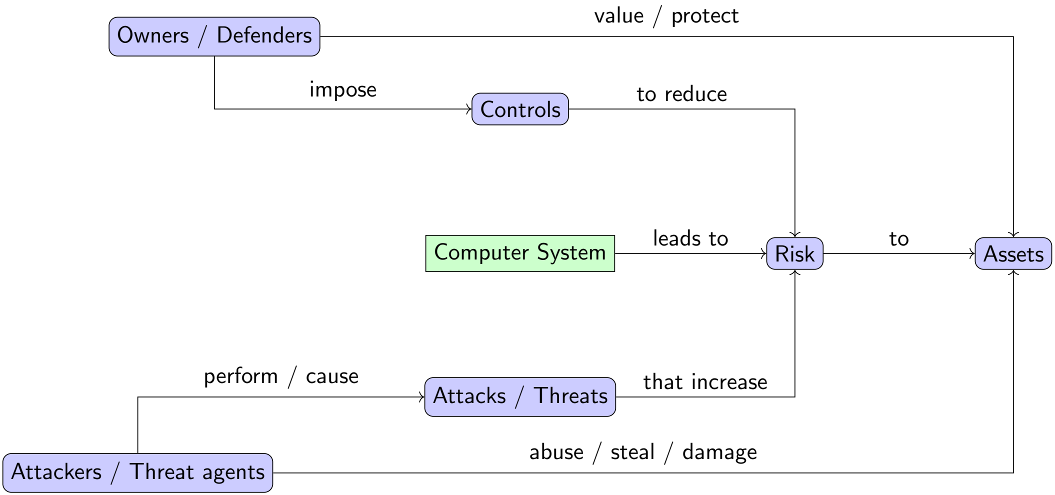
Attacks / Threats

that increase

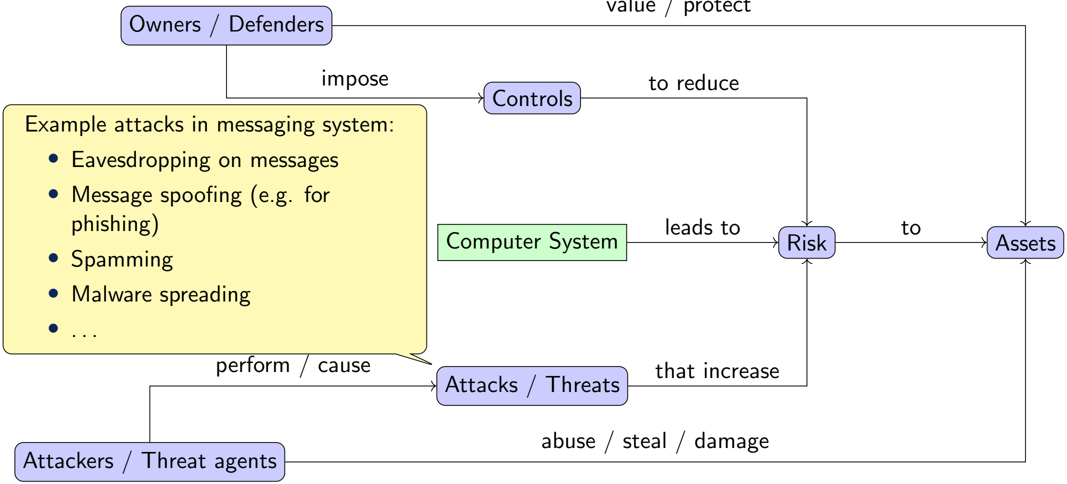
abuse / steal / damage

Attackers / Threat agents

Security concepts



Security concepts



Owners / Defenders

value / protect

impose

Controls

to reduce

Computer System

leads to

Risk

to

Assets

Attacks / Threats

that increase

perform / cause

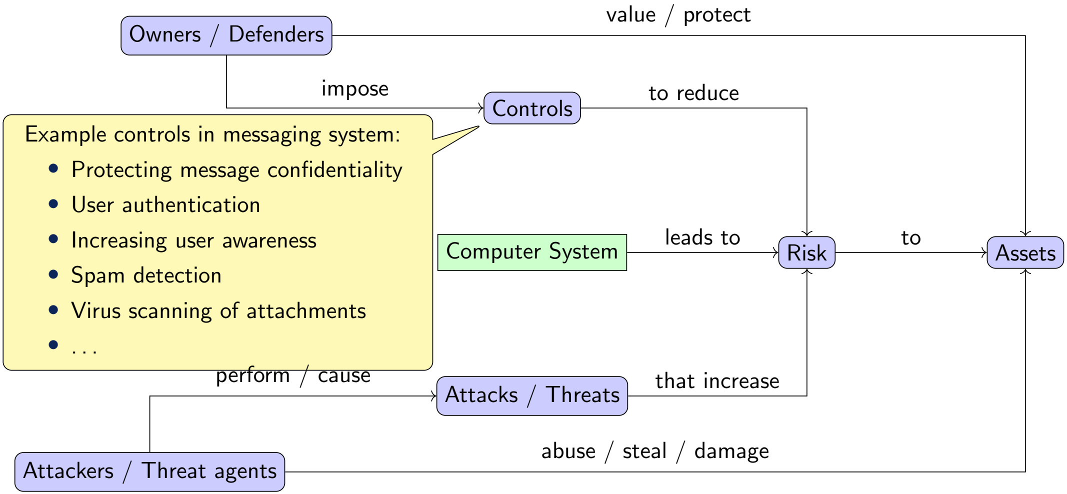
Attackers / Threat agents

abuse / steal / damage

Example attacks in messaging system:

- Eavesdropping on messages
- Message spoofing (e.g. for phishing)
- Spamming
- Malware spreading
- . . .

Security concepts



Owners / Defenders

value / protect

impose

Controls

to reduce

Example controls in messaging system:

- Protecting message confidentiality
- User authentication
- Increasing user awareness
- Spam detection
- Virus scanning of attachments
- . . .

Computer System

leads to

Risk

to

Assets

perform / cause

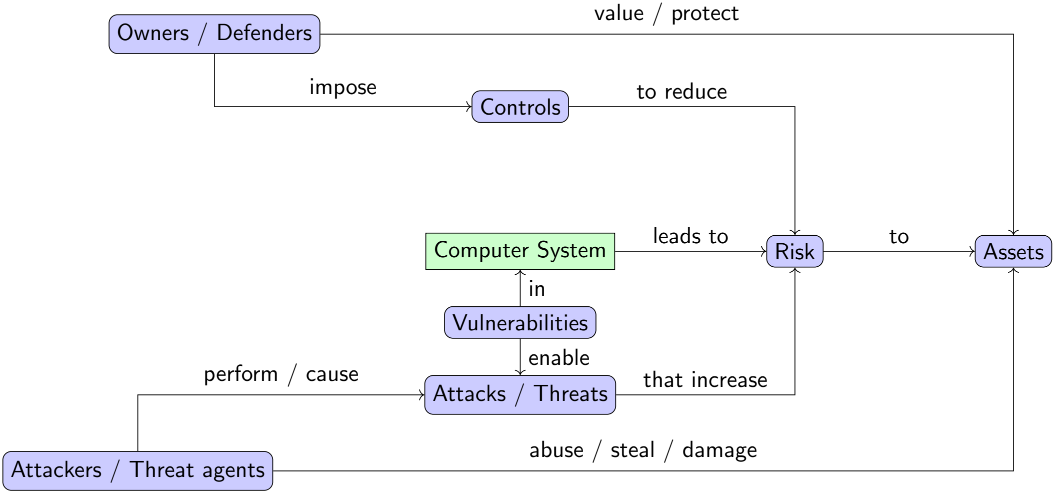
Attacks / Threats

that increase

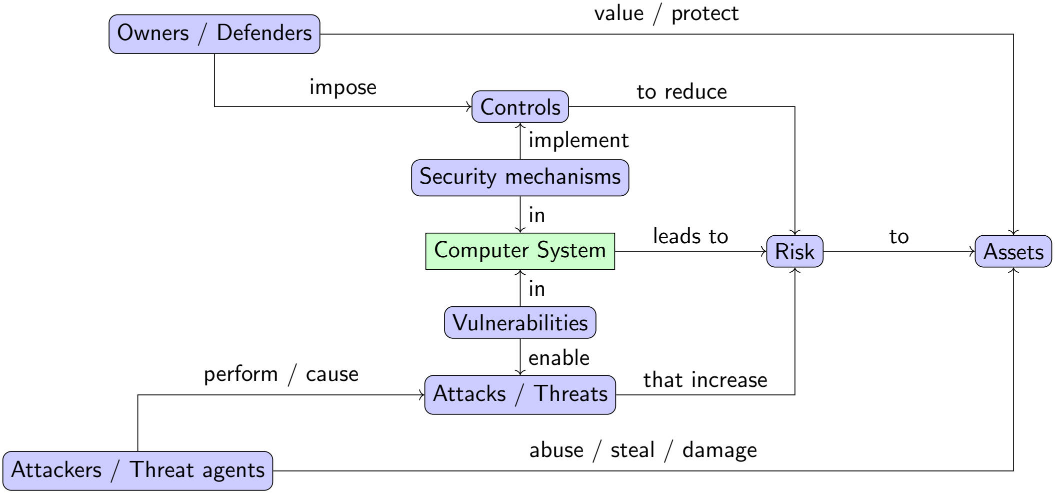
abuse / steal / damage

Attackers / Threat agents

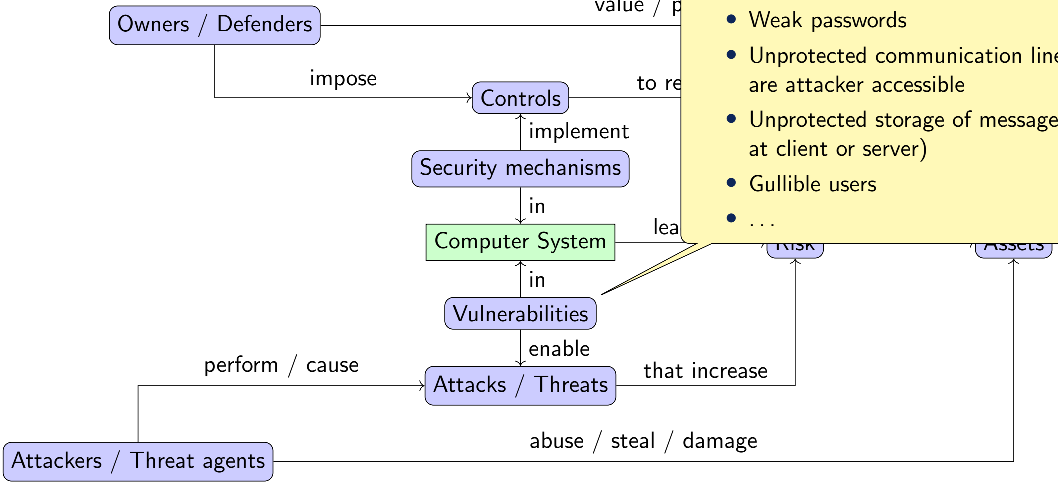
Security concepts



Security concepts



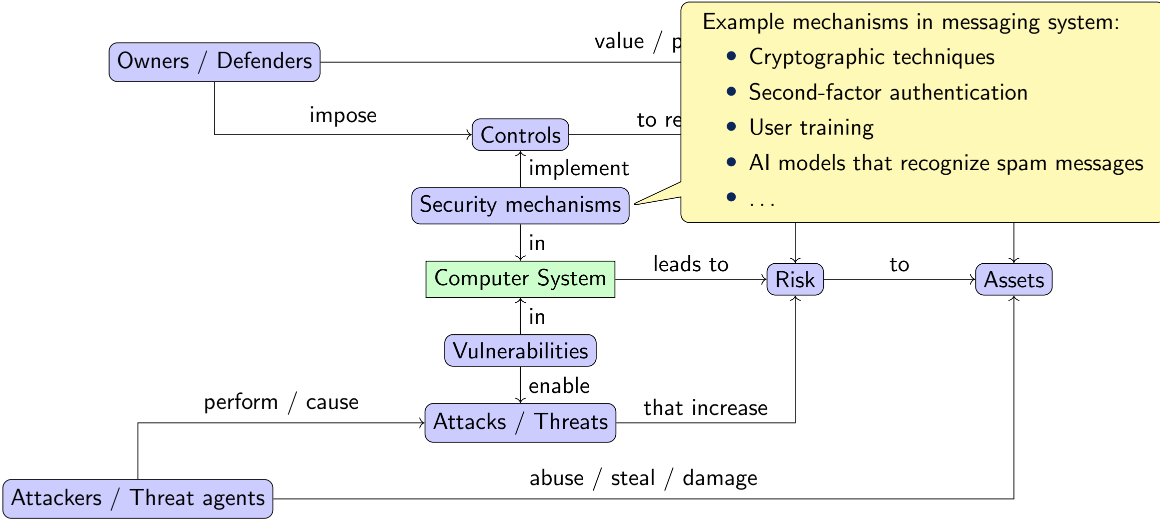
Security concepts



Example vulnerabilities in messaging system:

- Weak passwords
- Unprotected communication lines that are attacker accessible
- Unprotected storage of messages (either at client or server)
- Gullible users
- . . .

Security concepts



Example mechanisms in messaging system:

- Cryptographic techniques
- Second-factor authentication
- User training
- AI models that recognize spam messages
- . . .

Discussion

- The objective of computer security is to keep the risk to assets below an acceptable level.
- Computer security is a broad field, that is very interdisciplinary. Controls and security mechanisms can be technical, economical, legal, psychological, social, . . .
- Security engineering makes use of activities like:
  - Attacker modeling, to better understand attackers / threat agents.
  - Threat modeling, to better understand threats (potential attacks).
  - Security analysis, to understand if controls are sufficient to counter identified threats.
  - Risk analysis, to estimate remaining risk to assets.
  - . . .

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

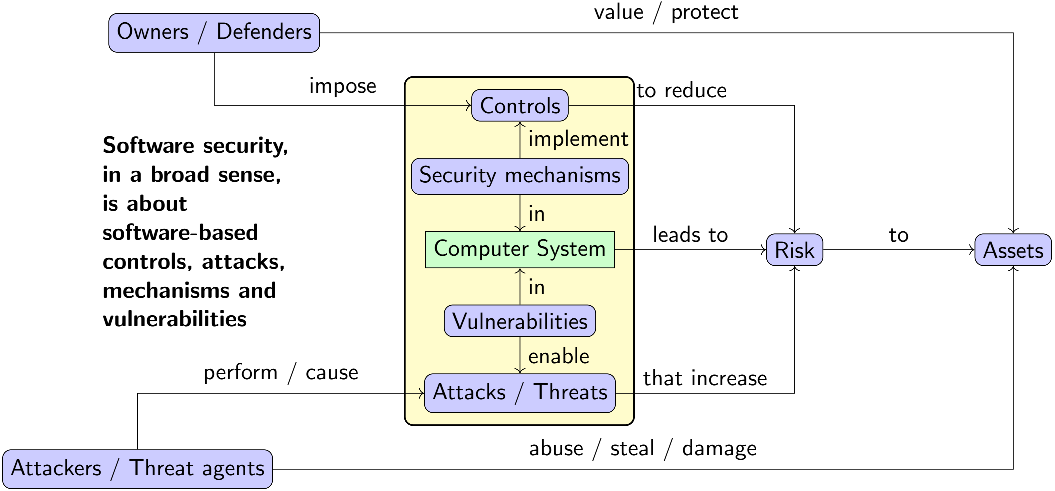
4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Software security

Software security, in a broad sense, is about software-based controls, attacks, mechanisms and vulnerabilities



Discussion

- In this interpretation, software security is very broad and includes at least cryptographic techniques, end-user aspects (e.g., usability), vulnerabity types (e.g., buffer overflows, injection vulnerabilities), attack techniques (e.g., return-oriented programming), access control and much more.
- For the remainder of these lecture, we will focus on controls (and corresponding mechanisms) that specify *acceptable behaviors* of a software system.
  - This is an important subclass of controls that can be conveniently studied using formal methods.
  - But it is indeed just a *subclass* of controls that matter in practice.
- More specifically we will focus on:
  - (A broad notion of) access control: *bad things should not happen* during software execution.
  - **(A broad notion of) information flow control: secrets should not leak from software executions.**

Examples

Access control:

- The client should not access the camera without getting user consent first.
- Allocated memory objects should not be accessed out of bounds.
- . . .

Information flow control:

- The client should not leak message content information to advertisement providers.
- The server should not leak the cryptographic keys used to protect the TLS connection to the client.
- The contact-matching component should not leak information about contacts that do not match.
- . . .

Formal security analysis

A rigorous analysis of the security of a system requires three ingredients:

- A *system model*: a rigorous description of the system we are trying to secure.
- A formalized *security objective*: a specification of what behaviors of the system are considered secure.
- An *attack model*: a precise definition of the class of attacks against which the system should be secure.

It can make sense to analyse security of the same system for different security objectives and under different attack models.

Modeling systems

We will consider systems that can be modeled as **transition systems**.
A transition system $(S, \to)$ consists of:

- a set of states S,
- and a transition relation $\to \subseteq S \times S$.

The systems we consider will often be programs in some specific programming language (a C-like language, or an assembly language). Standard techniques from **operational semantics** define transition systems that define the semantics of such programs.

Modeling security objectives: access control

Access control objectives say that some *bad things should not happen*.
They can be formalized by defining a unary predicate that defines what states are *good*.
The security objective is then that, for any initial state, the execution from that state should never reach a bad state. (Such objectives are also known as **safety properties**.)
Showing that a system satisfies this security objective is done by showing that an *invariant* for the initial states of the system implies the predicate.

Invariants

Given a labeled transition system $(S, \rightarrow)$, a predicate $I \subseteq S$ is an **invariant** for $S_0 \subseteq S$ iff:

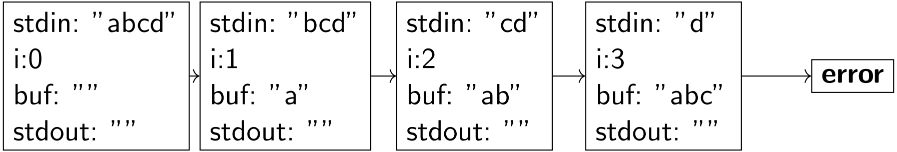
- $S_0 \subseteq I$
- $\forall s \in I, s \rightarrow s' \Rightarrow s' \in I$

Example

Security objective: the program should never output ”You win!”.

```c
void main() {
  char c = 0;
  char buf[3];
  int i = 0;
  while (buf[i] = getchar() != 'x') i++;
   if (c) printf("You win!");
  }
```

You could model this program as a transition system (sweeping some details under the carpet):



stdin: ”abcd”
i:0
buf: ””
stdout: ””

stdin: ”bcd”
i:1
buf: ”a”
stdout: ””

stdin: ”cd”
i:2
buf: ”ab”
stdout: ””

stdin: ”d”
i:3
buf: ”abc”
stdout: ””

error

Based on this model, the program is secure.

Example

Security objective: programs should never *get stuck*.
A program state is *good* if it either is a terminal state, or it can make a step.
The typical progress+preservation proofs use a *well-typedness invariant* to show that the security objective is satisfied:

- Initial states are well-typed.
- If a well-typed state can make a step, the new state is also well-typed (preservation).
- Well-typedness implies not being stuck (progress).

Modeling security objectives: information flow

Information flow properties say that *secrets should not leak*.

They can be formalized by defining a binary predicate that defines what pairs of states are *indistinguishable*.

The security objective is then that, for any two initial states that differ only in secrets, the executions from these two states should never become distinguishable. (Such objectives are also known as **noninterference properties**.)

Showing that a system satisfies this security objective is done by showing that a *simulation* for the set of pairs of initial states that differ only in secrets implies the relation.

Simulations

Given $(S, \rightarrow)$, a relation $\sim \subseteq S \times S$ is a **simulation** for $\sim_0 \subseteq S \times S$ iff:

- $\sim_0 \subseteq \sim$
- $\forall s^l \sim s^r$,
  - $s^l \rightarrow s^{l\prime} \Rightarrow s^r \rightarrow^* s^{r\prime} \wedge s^{l\prime} \sim s^{r\prime}$
  - $s^r \rightarrow s^{r\prime} \Rightarrow s^l \rightarrow^* s^{l\prime} \wedge s^{l\prime} \sim s^{r\prime}$

Example

```c
typedef struct item {
    bool secret;
    char* content;
} item_t;

int N;
item_t *items;

int main(int argc, char const *argv[]) {
    // Initialization ...
    int i = atoi(argv[1]); // i is provided as the first argument
    if (i >= N) printf("Not so many items\n");
    else if (items[i].secret) printf("Not accessible\n");
    else printf("%s\n", items[i].content);
}
```

We define states to be indistinguishable, if they have the same output.

Example: content of secret item does not leak

To specify that only content of secret items should not leak, $\sim_0$ relates any two initial states that are identical except for content of some secret items. E.g.,

i:1
N: 3
items: [(F,"A"),(T,"Secret"),(F,"B")]
stdout: ""

$\sim_0$

i:1
N:3
items: [(F,"A"),(T,"Private"),(F,"B")]
stdout: ""

Executions starting from such pairs of states remain indistinguishable, e.g.,

i:1
N: 3
items: [(F,"A"),(T,"Secret"),(F,"B")]
stdout: "Not accessible"

$\sim$

i:1
N:3
items: [(F,"A"),(T,"Private"),(F,"B")]
stdout: "Not accessible"

The relation $\sim$ that relates any two states that are identical except for content of some secret items can be shown to be a simulation that implies indistinguishability.

Example: number of items does leak

To specify that the number of secret items does not leak, $\sim_0$ relates any two initial states that have identical public items. E.g.,

i:2 N: 3 items: [(T,”S1”),(F,”B”), (T,”S2”)] stdout: ””	$\sim_0$	i:2 N: 2 items: [(T,”S1”),(F,”B”)] stdout: ””

Executions starting from such pairs of states can become distinguishable, e.g.,

i:2 N: 3 items: [(T,”S1”),(F,”B”), (T,”S2”)] stdout: ”Not accessible”	$\not\sim$	i:2 N: 2 items: [(T,”S1”),(F,”B”)] stdout: ”Not so many items”

There exists no simulation for this choice of $\sim_0$ that implies indistinguishability. The program *leaks* (something about) the number of secret items.

Modeling attacks

Our general approach for modeling attacks will be:

- Our system model should include all information and behavior exploited in the attacks we care about (possibly using overapproximation to simplify the model).
- Part of the initial state is considered to be under attacker control, for instance input provided by the attacker.

Designing an adequate system model, security objective and attack model is non-trivial. For an extensive discussion, see:

- Márton Bognár, Jo Van Bulck, Frank Piessens, *Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures*, IEEE S&P 2022.

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Abstraction

- We build systems using *layers of abstractions* to manage complexity, e.g.,

Structured Programming Language (e.g. Python or C)
Compiler or interpreter
Instruction Set Architecture (e.g. RISC-V or Intel x86)
Processor
Hardware description language (e.g. Verilog or VHDL)
Technology for implementing digital circuits

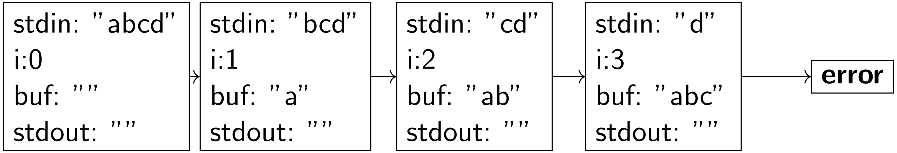
- One usually reasons about the functionality of the system at one (appropriately chosen) layer of abstraction.
- But attacks can rely on or exploit details of lower layers.

Example: memory corruption attacks

The protoypical example of layer-below attacks are *memory corruption attacks*.

```
void main() {
  char c = 0;
  char buf[3];
  int i = 0;
  while (buf[i] = getchar() != ’x’) i++;
   if (c) printf("You win!");
  }
```

We modeled this code at a high level of abstraction:



stdin: "abcd"
i:0
buf: ""
stdout: ""

stdin: "bcd"
i:1
buf: "a"
stdout: ""

stdin: "cd"
i:2
buf: "ab"
stdout: ""

stdin: "d"
i:3
buf: "abc"
stdout: ""

error

But what happens at lower levels of abstraction?

Example: buffer overread attacks

Obviously, this can also break confidentiality properties.

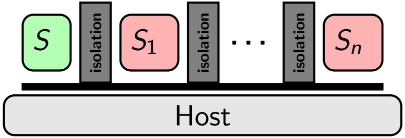
```c
#include<stdio.h>
#include<stdlib.h>
int public[] = {1,2,3};
int secret[] = {41,666,0xDEADBEEF};

void main(int argc, char const *argv[]) {
    int i = atoi(argv[1]); // i is provided as the first argument
    printf("%d",public[i]);
}
```

If we consider the `secret[]` array to be secret, one might expect this code to not leak secrets.
But what happens at lower levels of abstraction?

Resource sharing

- *Virtualization* and/or *sandboxing* enable sharing of a computation platform, e.g.,



Host	Isolation mechanism
Operating system	Process isolation
Hypervisor	VM isolation
Language VM	Language safety

- The isolation mechanism should limit the interaction between S and the S_i to specific identified communication channels.
- But most isolation mechanisms do *not* achieve perfect isolation and hence introduce new ways to interact with S.
- And even more, to support *confidential computing*, S should be isolated from the *host*, which is very hard.

Example attacks

- Fault injection attacks: hosts or subjects sharing the same platform can unexpectedly change parts of the state:
  - Rowhammer attacks
  - Voltage glitching attacks
  - . . .
- Side channel attacks: hosts or subjects sharing the same platform can unexpectedly read parts of the state:
  - Cache attacks
  - Page fault attacks
  - . . .

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Conclusions and outlook

- Computer security in general, as well as the subfield of software security, are broad and interdisciplinary fields, including both technical and non-technical challenges.
- But an important subset of technical security controls for software can be usefully modeled and studied mathematically.
- This study is complicated by the way in which we build computer systems.
- This week, we will illustrate this in depth by:
  - Formalizing a specific class of confidentiality properties for software.
  - Studying attacks against these properties (and corresponding mitigations), focusing on cross-layer attacks on shared platforms.

Main objective of these lectures

Study the impact of layering and resource sharing on confidentiality properties of software.

Overview

1. Introduction
   - 1.1 Computer security
   - 1.2 Software security
   - 1.3 Some key system engineering techniques: abstraction and resource sharing
   - 1.4 Conclusions and outlook
2. **From Structured Programming Language to ISA**
   - 2.1 A simple model for an imperative language
   - 2.2 A simple ISA model
3. Below the ISA: microarchitectural attacks
   - 3.1 The rise of microarchitectural attacks
   - 3.2 Cache attacks
   - 3.3 Transient execution attacks
4. Defenses
   - 4.1 Using fences
   - 4.2 ProSpeCT: a hardware/software codesign
5. Conclusions

A simple imperative language

$$
\begin{array}{rrr}
n \in & \mathbb{N} & \text{(Natural numbers)} \\
v \in & \mathit{Vars} & \text{(Mutable variables)} \\
a \in & \mathit{Arrays} & \text{(Mutable arrays)} \\
\odot \in & \{+, =, <, \ldots\} & \text{(Primitive operations on numbers)} \\
e ::= & n \mid v \mid e \odot e \mid \mathrm{size}(a) & \text{(Expressions)} \\
c ::= & v := e & \text{(Commands)} \\
\mid & v := a[e] & \\
\mid & a[e] := v & \\
\mid & \textbf{if } e \textbf{ then } c \textbf{ else } c & \\
\mid & \textbf{while } e \textbf{ do } c & \\
\mid & c; c & \\
\mid & \textbf{skip} &
\end{array}
$$

Operational semantics

A program state (m, ρ, c) consists of:

- $m \in \mathit{Arrays} \to \textbf{list}(\mathbb{N})$ maps arrays to their current value.
- $\rho \in \mathit{Vars} \to \mathbb{N}$ maps variables to their current value.
- Command c represents the continuation of the program.

Defining the semantics of expressions given (m, ρ) is trivial, for instance:

$$\frac{}{(m, \rho) \vdash n \downarrow n} \ (\text{E-Expr-Lit}) \qquad \frac{(m, \rho) \vdash e_1 \downarrow n_1 \qquad (m, \rho) \vdash e_2 \downarrow n_2}{(m, \rho) \vdash e_1 + e_2 \downarrow n_1 + n_2} \ (\text{E-Expr-Plus})$$

$$\frac{}{(m, \rho) \vdash v \downarrow \rho(v)} \ (\text{E-Expr-Var}) \qquad \frac{}{(m, \rho) \vdash \mathrm{size}(a) \downarrow \mathrm{length}(m(a))} \ (\text{E-Expr-Len})$$

Defining the semantics as a transition system is standard. We provide some of the rules on the following slide.

$$\frac{}{(m, \rho, \mathbf{skip}; c) \rightarrow (m, \rho, c)} \quad (\textsc{E-Stmt-SeqSkip})$$

$$\frac{(m, \rho, c_1) \rightarrow (m', \rho', c_1')}{(m, \rho, c_1; c_2) \rightarrow (m', \rho', c_1'; c_2)} \quad (\textsc{E-Stmt-Seq})$$

$$\frac{(m, \rho) \vdash e \downarrow n}{(m, \rho, v := e) \rightarrow (m, \rho[v \mapsto n], \mathbf{skip})} \quad (\textsc{E-Stmt-Assign})$$

$$\frac{(m, \rho) \vdash e \downarrow n \qquad n \neq 0}{(m, \rho, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \rightarrow (m, \rho, c_1)} \quad (\textsc{E-Stmt-If1})$$

$$\frac{(m, \rho) \vdash e \downarrow n \qquad n = 0}{(m, \rho, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \rightarrow (m, \rho, c_2)} \quad (\textsc{E-Stmt-If2})$$

$$\frac{(m, \rho) \vdash e \downarrow n \qquad n \neq 0}{(m, \rho, \mathbf{while}\ e\ \mathbf{do}\ c) \rightarrow (m, \rho, c; \mathbf{while}\ e\ \mathbf{do}\ c)} \quad (\textsc{E-Stmt-While1})$$

$$\frac{(m, \rho) \vdash e \downarrow n \qquad n = 0}{(m, \rho, \mathbf{while}\ e\ \mathbf{do}\ c) \rightarrow (m, \rho, \mathbf{skip})} \quad (\textsc{E-Stmt-While2})$$

$$\frac{(m, \rho) \vdash e \downarrow n \qquad n < \mathrm{length}(m(a))}{(m, \rho, v := a[e]) \rightarrow (m, \rho[v \mapsto m(a)[n]], \mathbf{skip})} \ (\textsc{E-Stmt-Load})$$

$$\frac{(m, \rho) \vdash e \downarrow n \qquad n < \mathrm{length}(m(a))}{(m, \rho, a[e] := v) \rightarrow (m[a \mapsto (m(a)[n \mapsto \rho(v)])], \rho, \mathbf{skip})} \ (\textsc{E-Stmt-Store})$$

Example

$(\{\ \text{items} \mapsto [3{,}14{,}7]\ \}, \{\ \text{i} \mapsto 2, \text{out} \mapsto 0\ \},$

```
if i > size(items) then out := 0
else v := items[i];
     if v < 10 then out := items[i]
     else out := 10
```

$)$

Example

$(\{ \text{ items} \mapsto [3,14,7] \}, \{ \text{ i} \mapsto 2, \text{ out} \mapsto 0 \},$

```
if i > size(items) then out := 0
else v := items[i];
     if v < 10 then out := items[i]
     else out := 10
```

$)$

$\rightarrow(\{ \text{ items} \mapsto [3,14,7] \}, \{ \text{ i} \mapsto 2, \text{ out} \mapsto 0 \},$

```
v := items[i];
if v < 10 then out := items[i]
else out := 10
```

$)$

Example

$(\{\ \text{items} \mapsto [3{,}14{,}7]\ \}, \{\ \text{i} \mapsto 2,\ \text{out} \mapsto 0\ \},$

```
if i > size(items) then out := 0
else v := items[i];
     if v < 10 then out := items[i]
     else out := 10
```

$)$

$\rightarrow(\{\ \text{items} \mapsto [3{,}14{,}7]\ \}, \{\ \text{i} \mapsto 2,\ \text{out} \mapsto 0\ \},$

```
v := items[i];
if v < 10 then out := items[i]
else out := 10
```

$)$

$\rightarrow^2(\{\ \text{items} \mapsto [3{,}14{,}7]\ \}, \{\ \text{i} \mapsto 2,\ \text{out} \mapsto 0,\ \text{v} \mapsto 7\ \},$

```
if v < 10 then out := items[i]
else out := 10
```

$)$

Example

$(\{\ \text{items} \mapsto [3,14,7]\ \}, \{\ \text{i} \mapsto 2,\ \text{out} \mapsto 0\ \},$

```
if i > size(items) then out := 0
else v := items[i];
     if v < 10 then out := items[i]
     else out := 10
```

$)$

$\rightarrow(\{\ \text{items} \mapsto [3,14,7]\ \}, \{\ \text{i} \mapsto 2,\ \text{out} \mapsto 0\ \},$

```
v := items[i];
if v < 10 then out := items[i]
else out := 10
```

$)$

$\rightarrow^2(\{\ \text{items} \mapsto [3,14,7]\ \}, \{\ \text{i} \mapsto 2,\ \text{out} \mapsto 0,\ \text{v} \mapsto 7\ \},$

```
if v < 10 then out := items[i]
else out := 10
```

$)$

$\rightarrow(\{\ \text{items} \mapsto [3,14,7]\ \}, \{\ \text{i} \mapsto 2,\ \text{out} \mapsto 0,\ \text{v} \mapsto 7\ \},$

```
out := items[i]
```

$)$

Example

$$(\{\ \mathsf{items} \mapsto [3,14,7]\ \}, \{\ \mathsf{i} \mapsto 2,\ \mathsf{out} \mapsto 0\ \}, \begin{array}{l}\texttt{if i > size(items) then out := 0}\\ \texttt{else v := items[i];}\\ \phantom{\texttt{else }}\texttt{if v < 10 then out := items[i]}\\ \phantom{\texttt{else }}\texttt{else out := 10}\end{array})$$

$$\rightarrow(\{\ \mathsf{items} \mapsto [3,14,7]\ \}, \{\ \mathsf{i} \mapsto 2,\ \mathsf{out} \mapsto 0\ \}, \begin{array}{l}\texttt{v := items[i];}\\ \texttt{if v < 10 then out := items[i]}\\ \texttt{else out := 10}\end{array})$$

$$\rightarrow^{2}(\{\ \mathsf{items} \mapsto [3,14,7]\ \}, \{\ \mathsf{i} \mapsto 2,\ \mathsf{out} \mapsto 0,\ \mathsf{v} \mapsto 7\ \}, \begin{array}{l}\texttt{if v < 10 then out := items[i]}\\ \texttt{else out := 10}\end{array})$$

$$\rightarrow(\{\ \mathsf{items} \mapsto [3,14,7]\ \}, \{\ \mathsf{i} \mapsto 2,\ \mathsf{out} \mapsto 0,\ \mathsf{v} \mapsto 7\ \}, \texttt{out := items[i]})$$

$$\rightarrow(\{\ \mathsf{items} \mapsto [3,14,7]\ \}, \{\ \mathsf{i} \mapsto 2,\ \mathsf{out} \mapsto 7,\ \mathsf{v} \mapsto 7\ \}, \textbf{skip})$$

Defining noninterference

We mark a subset of variables and arrays as *public*, and we define two program states $s = (m, \rho, c)$ and $s' = (m', \rho', c')$ to be *indistinguishable* if and only if:

- $\rho(v) = \rho'(v)$ for all public v.
- $m(a) = m'(a)$ for all public a.

We mark a subset of variables and arrays as *secret*, and we define two initial program states for command c, $s = (m, \rho, c)$ and $s' = (m', \rho', c)$ to differ only in secrets ($s \sim_0^c$ s') if and only if:

- $\rho(v) = \rho'(v)$ for all non-secret v.
- $m(a) = m'(a)$ for all non-secret a.

Definition

A command c is *noninterferent* if there exists a simulation for $\sim_0^c$ that implies indistinguishability.

For simplicity, we use the convention that variables/arrays that have a name starting with 's' are secret, and those starting with 'p' are public.

Simple Examples

- Insecure example:

```
public := secret
```

Simple Examples

- Insecure example:

```
public := secret
```

- Insecure example:

```
public := secret + public
```

Simple Examples

- Insecure example:

```
public := secret
```

- Insecure example:

```
public := secret + public
```

- Insecure example:

```
if secret=0 then public := 0
             else public := 1
```

Simple Examples

- Insecure example:

```
public := secret
```

- Insecure example:

```
public := secret + public
```

- Insecure example:

```
if secret=0 then public := 0
            else public := 1
```

- Secure example:

```
secret := public
```

More Examples

- Insecure example:

```
p[2] := secret
```

More Examples

- Insecure example:

```
p[2] := secret
```

- Insecure example:

```
public := s[2]
```

More Examples

- Insecure example:

```
p[2] := secret
```

- Insecure example:

```
public := s[2]
```

- Secure example:

```
s[secret] := secret1
```

More Examples

- Insecure example:

```
p[2] := secret
```

- Insecure example:

```
public := s[2]
```

- Secure example:

```
s[secret] := secret1
```

- Insecure example:

```
p[secret] := public
```

More Examples

- Insecure example:

```
p[2] := secret
```

- Insecure example:

```
public := s[2]
```

- Secure example:

```
s[secret] := secret1
```

- Insecure example:

```
p[secret] := public
```

- Secure example:

```
public := p[public2]
```

More Examples

- Insecure example:

  ```
  p[2] := secret
  ```

- Insecure example:

  ```
  public := s[2]
  ```

- Secure example:

  ```
  s[secret] := secret1
  ```

- Insecure example:

  ```
  p[secret] := public
  ```

- Secure example:

  ```
  public := p[public2]
  ```

For a fun introduction to breaking noninterference, see the IFC challenge by Chalmers: `https://ifc-challenge.appspot.com/`

Side channels

Is the following program secure?

```
while 10 < secret do secret := secret + 1
```

Side channels

Is the following program secure?

```
while 10 < secret do secret := secret + 1
```

What about the following one?

```
if secret< 10 then secret := secret + secret
                   else secret := secret + secret + secret
```

Side channels

Is the following program secure?

```
while 10 < secret do secret := secret + 1
```

What about the following one?

```
if secret< 10 then secret := secret + secret
                   else secret := secret + secret + secret
```

Our definition of noninterference is timing and termination *insensitive*. Variants of the definition exist that capture timing and termination side channels.

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Introduction

Instruction Set Architectures (ISA) are the interface between hardware and software. They are (in some sense) the *lowest-level* software programming languages.
The ISA of (almost) all modern hardware is unstructured: memory is a flat array of words, and a program is a sequence of simple instructions (machine code, assembly code).
Compilation of structured programming languages like C, Java or Rust requires the compiler to:

- map data structures on this unstructured memory,
- translate structured code to machine code.

This can cause code that looks secure at source code level to become insecure after compilation.

A simple ISA model

ISA program syntax:

$$\begin{array}{rcll}
\text{Register names} & r & \in & \mathrm{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow\ r & \mid \\
 & & & \textbf{jmp}\ e & \mid \\
 & & & \textbf{beqz}\ r\ n \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots]
\end{array}$$

A simple ISA model

ISA program syntax:

$$
\begin{array}{rcll}
\text{Register names} & r & \in & \text{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \mathbf{load}[e] & \mid \\
 & & & \mathbf{store}[e] \leftarrow \; r & \mid \\
 & & & \mathbf{jmp} \; e & \mid \\
 & & & \mathbf{beqz} \; r \; n \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots]
\end{array}
$$

Example program:

$$
\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ; \mathbf{while} \; (i < 2) \; \{ \\
1 & : & \mathbf{beqz} \; r_0 \; 6 & ; \\
2 & : & r_0 \leftarrow \mathbf{load}[a + i] & ; \quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ; \quad i = i + 1 \\
5 & : & \mathbf{jmp} \; 0 & ; \} \\
6 & : & \ldots &
\end{array}
$$

A simple ISA model

ISA program syntax:

$$\begin{array}{rccll}
\text{Register names} & r & \in & \text{Regs} & \\
\text{Natural numbers} & n & \in & \mathbb{N} & \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots & \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow\ r & \mid \\
 & & & \textbf{jmp}\ e & \mid \\
 & & & \textbf{beqz}\ r\ n & \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots] &
\end{array}$$

Example program:

$$\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ;\textbf{while}\ (i < 2)\ \{ \\
1 & : & \textbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \textbf{load}[a + i] & ;\quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ;\quad i = i + 1 \\
5 & : & \textbf{jmp}\ 0 & ;\} \\
6 & : & \ldots &
\end{array}$$

Program state:

$$\begin{array}{rccl}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

A simple ISA model

ISA program syntax:

$$
\begin{array}{rcll}
\text{Register names} & r & \in & \mathit{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow\ r & \mid \\
 & & & \textbf{jmp}\ e & \mid \\
 & & & \textbf{beqz}\ r\ n \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots]
\end{array}
$$

Example program:

$$
\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ;\textbf{while}\ (i < 2)\ \{ \\
1 & : & \textbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \textbf{load}[a + i] & ;\quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ;\quad i = i + 1 \\
5 & : & \textbf{jmp}\ 0 & ;\} \\
6 & : & \ldots
\end{array}
$$

Program state:

$$
\begin{array}{rcll}
\text{Register state} & \rho & \in & \mathit{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}
$$

Example execution:

Memory state $= [5, 4, \ldots]$

pc	0
a	0
i	0
sum	0
r_0	0

A simple ISA model

ISA program syntax:

$$\begin{array}{rclll}
\text{Register names} & r & \in & \text{Regs} & \\
\text{Natural numbers} & n & \in & \mathbb{N} & \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots & \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \mathbf{load}[e] & \mid \\
 & & & \mathbf{store}[e] \leftarrow \ r & \mid \\
 & & & \mathbf{jmp}\ e & \mid \\
 & & & \mathbf{beqz}\ r\ n & \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots] &
\end{array}$$

Example program:

$$\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ; \mathbf{while}\ (i < 2)\ \{ \\
1 & : & \mathbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \mathbf{load}[a + i] & ; \quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ; \quad i = i + 1 \\
5 & : & \mathbf{jmp}\ 0 & ; \} \\
6 & : & \ldots &
\end{array}$$

Program state:

$$\begin{array}{rccl}
\text{Register state} & \rho & \in & \text{Regs} \to \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

Example execution:

Memory state $= [5, 4, \ldots]$

pc	0
a	0
i	0
sum	0
r_0	0

$\rightarrow$

pc	1
a	0
i	0
sum	0
r_0	1

A simple ISA model

ISA program syntax:

$$\begin{array}{rcll}
\text{Register names} & r & \in & \text{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots \\
\text{Instructions} & i & ::= & r \leftarrow e \qquad\qquad\qquad\qquad \mid \\
 & & & r \leftarrow \textbf{load}[e] \qquad\qquad\quad \mid \\
 & & & \textbf{store}[e] \leftarrow \; r \qquad\qquad \mid \\
 & & & \textbf{jmp}\ e \qquad\qquad\qquad\qquad \mid \\
 & & & \textbf{beqz}\ r\ n \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots]
\end{array}$$

Example program:

$$\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ; \textbf{while}\ (i < 2)\ \{ \\
1 & : & \textbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \textbf{load}[a + i] & ; \quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ; \quad i = i + 1 \\
5 & : & \textbf{jmp}\ 0 & ; \} \\
6 & : & \ldots &
\end{array}$$

Program state:

$$\begin{array}{rcll}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

Example execution:

Memory state $= [5, 4, \ldots]$

pc	0
a	0
i	0
sum	0
r_0	0

→

pc	1
a	0
i	0
sum	0
r_0	1

→

pc	2
a	0
i	0
sum	0
r_0	1

A simple ISA model

ISA program syntax:

$$
\begin{array}{rccl}
\text{Register names} & r & \in & \text{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots \\
\text{Instructions} & i & ::= & r \leftarrow e \hfill \mid \\
 & & & r \leftarrow \textbf{load}[e] \hfill \mid \\
 & & & \textbf{store}[e] \leftarrow \; r \hfill \mid \\
 & & & \textbf{jmp}\ e \hfill \mid \\
 & & & \textbf{beqz}\ r\ n \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots]
\end{array}
$$

Example program:

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

Program state:

$$
\begin{array}{rccl}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}
$$

Example execution:

Memory state $= [5, 4, \ldots]$

pc	0
a	0
i	0
sum	0
r_0	0

→

pc	1
a	0
i	0
sum	0
r_0	1

→

pc	2
a	0
i	0
sum	0
r_0	1

→

pc	3
a	0
i	0
sum	0
r_0	5

A simple ISA model

ISA program syntax:

$$\begin{array}{rccll}
\text{Register names} & r & \in & \text{Regs} & \\
\text{Natural numbers} & n & \in & \mathbb{N} & \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots & \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow \ r & \mid \\
 & & & \textbf{jmp}\ e & \mid \\
 & & & \textbf{beqz}\ r\ n & \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots] &
\end{array}$$

Example program:

$$\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ; \textbf{while}\ (i < 2)\ \{ \\
1 & : & \textbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \textbf{load}[a + i] & ; \quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ; \quad i = i + 1 \\
5 & : & \textbf{jmp}\ 0 & ; \} \\
6 & : & \ldots &
\end{array}$$

Program state:

$$\begin{array}{rccl}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

Example execution:

Memory state $= [5, 4, \ldots]$

pc	0
a	0
i	0
sum	0
r_0	0

→

pc	1
a	0
i	0
sum	0
r_0	1

→

pc	2
a	0
i	0
sum	0
r_0	1

→

pc	3
a	0
i	0
sum	0
r_0	5

→

pc	4
a	0
i	0
sum	5
r_0	5

A simple ISA model

ISA program syntax:

$$\begin{array}{rcll}
\text{Register names} & r & \in & \text{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow \ r & \mid \\
 & & & \textbf{jmp}\ e & \mid \\
 & & & \textbf{beqz}\ r\ n \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots]
\end{array}$$

Example program:

$$\begin{array}{lcll}
0 & : & r_0 \leftarrow i < 2 & ; \textbf{while}\ (i < 2)\ \{ \\
1 & : & \textbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \textbf{load}[a + i] & ; \quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ; \quad i = i + 1 \\
5 & : & \textbf{jmp}\ 0 & ; \} \\
6 & : & \ldots
\end{array}$$

Program state:

$$\begin{array}{rcl}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

Example execution:

Memory state $= [5, 4, \ldots]$

pc	0
a	0
i	0
sum	0
r_0	0

→

pc	1
a	0
i	0
sum	0
r_0	1

→

pc	2
a	0
i	0
sum	0
r_0	1

→

pc	3
a	0
i	0
sum	0
r_0	5

→

pc	4
a	0
i	0
sum	5
r_0	5

→

pc	5
a	0
i	1
sum	5
r_0	5

A simple ISA model

ISA program syntax:

$$\begin{array}{rccl}
\text{Register names} & r & \in & \text{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \dots \\
\text{Instructions} & i & ::= & r \leftarrow e \qquad\qquad\qquad\qquad \mid \\
 & & & r \leftarrow \textbf{load}[e] \qquad\qquad\quad \mid \\
 & & & \textbf{store}[e] \leftarrow \; r \qquad\qquad \mid \\
 & & & \textbf{jmp}\ e \qquad\qquad\qquad\qquad \mid \\
 & & & \textbf{beqz}\ r\ n \\
\text{Programs} & p & ::= & [i_0, i_1, \dots]
\end{array}$$

Example program:

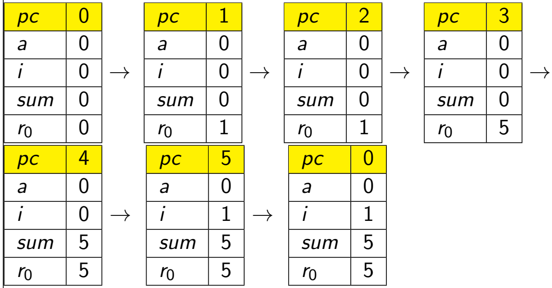
$$\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ;\textbf{while}\ (i < 2)\ \{ \\
1 & : & \textbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \textbf{load}[a + i] & ; \quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i + 1 & ; \quad i = i + 1 \\
5 & : & \textbf{jmp}\ 0 & ;\} \\
6 & : & \dots &
\end{array}$$

Program state:

$$\begin{array}{rccl}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \dots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

Example execution:

Memory state $= [5, 4, \dots]$



pc	0
a	0
i	0
sum	0
r_0	0

→

pc	1
a	0
i	0
sum	0
r_0	1

→

pc	2
a	0
i	0
sum	0
r_0	1

→

pc	3
a	0
i	0
sum	0
r_0	5

→

pc	4
a	0
i	0
sum	5
r_0	5

→

pc	5
a	0
i	1
sum	5
r_0	5

→

pc	0
a	0
i	1
sum	5
r_0	5

A simple ISA model

ISA program syntax:

$$\begin{array}{rrcll}
\text{Register names} & r & \in & \text{Regs} & \\
\text{Natural numbers} & n & \in & \mathbb{N} & \\
\text{Expressions} & e & ::= & n \mid r \mid e+e \mid e<e \mid \ldots & \\
\text{Instructions} & i & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow\ r & \mid \\
 & & & \textbf{jmp}\ e & \mid \\
 & & & \textbf{beqz}\ r\ n & \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots] &
\end{array}$$

Example program:

$$\begin{array}{rcll}
0 & : & r_0 \leftarrow i < 2 & ;\textbf{while}\ (i<2)\ \{ \\
1 & : & \textbf{beqz}\ r_0\ 6 & ; \\
2 & : & r_0 \leftarrow \textbf{load}[a+i] & ;\quad sum = sum + a[i] \\
3 & : & sum \leftarrow sum + r_0 & ; \\
4 & : & i \leftarrow i+1 & ;\quad i = i+1 \\
5 & : & \textbf{jmp}\ 0 & ;\} \\
6 & : & \ldots &
\end{array}$$

Program state:

$$\begin{array}{rcll}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

Example execution:

Memory state $= [5, 4, \ldots]$

pc	0		pc	1		pc	2		pc	3	
a	0		a	0		a	0		a	0	
i	0	$\rightarrow$	i	0	$\rightarrow$	i	0	$\rightarrow$	i	0	$\rightarrow$
sum	0		sum	0		sum	0		sum	0	
r_0	0		r_0	1		r_0	1		r_0	5	

pc	4		pc	5		pc	0		pc	6
a	0		a	0		a	0		a	0
i	0	$\rightarrow$	i	1	$\rightarrow$	i	1	$\rightarrow^*$	i	2
sum	5		sum	5		sum	5		sum	9
r_0	5		r_0	5		r_0	5		r_0	0

Compilation to this ISA

For the simple programming language we considered, compilation is relatively straightforward:

- (Scalar) variables are directly mapped on registers, and the register contains the corresponding value.
- Arrays are mapped on memory, the array variable is mapped on a register, and that register contains the start address of the array in memory.
- While loops and if-then-else are compiled to branches in the standard way.

We just explain compilation by means of an example. Formalizing the compiler would be an interesting exercise.

Example compilation

SOURCE

TARGET

```
sindex := 0;
secret := 0;
while secret < 100


  stmp := s[sindex];
  secret := stmp + secret;


public := p[public2];
```

$$
\begin{array}{rl}
 & \mathit{sindex} \leftarrow 0 \\
 & \mathit{secret} \leftarrow 0 \\
\mathrm{loop}: & \mathit{stest} \leftarrow \mathit{secret} < 100 \\
 & \textbf{beqz}\ \mathit{stest}\ \mathrm{endloop} \\
 & \mathit{stmp} \leftarrow \textbf{load}[\mathit{s} + \mathit{sindex}] \\
 & \mathit{secret} \leftarrow \mathit{secret} + \mathit{stmp} \\
 & \textbf{jmp}\ \mathrm{loop} \\
\mathrm{endloop}: & \mathit{public} \leftarrow \textbf{load}[\mathit{p} + \mathit{public}2]
\end{array}
$$

Do you see the potential security issue?

Example compilation

SOURCE

```
sindex := 0;
secret := 0;
while secret < 100


  stmp := s[sindex];
  secret := stmp + secret;


public := p[public2];
```

TARGET

```
          $\mathit{sindex} \leftarrow 0$
          $\mathit{secret} \leftarrow 0$
loop :    $\mathit{stest} \leftarrow \mathit{secret} < 100$
          $\mathbf{beqz}\ \mathit{stest}\ \mathrm{endloop}$
          $\mathit{stmp} \leftarrow \mathbf{load}[\mathit{s} + \mathit{sindex}]$
          $\mathit{secret} \leftarrow \mathit{secret} + \mathit{stmp}$
          $\mathbf{jmp}\ \mathrm{loop}$
endloop : $\mathit{public} \leftarrow \mathbf{load}[\mathit{p} + \mathit{public}2]$
```

Do you see the potential security issue?

Assume arrays `s` and `p` are each of size 10, and that the compiler maps array `p` on memory addresses 0 to 9, and `s` on addresses 10 to 19. An out-of-bounds value for `public2` leaks values from `s`.

Programs can be secure despite overflows

In this model, reading out of bounds to assign to a secret variable is secure.

```
sindex := 0;
secret := 0;
while secret < 100
  stmp := s[sindex];
  secret := stmp + secret;
secret2 := p[public2];
```

This observation can be useful in some settings (for instance, in the protection against transient execution attacks), but obviously out-of-bounds accesses are generally to be avoided.

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

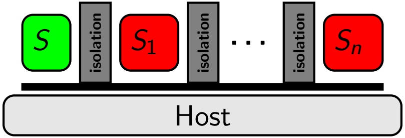
4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Introduction

It is common to share computation infrastructure or computation platforms among multiple, possibly mutually distrusting stakeholders. (Examples include: cloud, web browsers, mobile platforms, . . .)

Virtualization and/or *sandboxing* are used to enable sharing of a computation platform securely, e.g.:

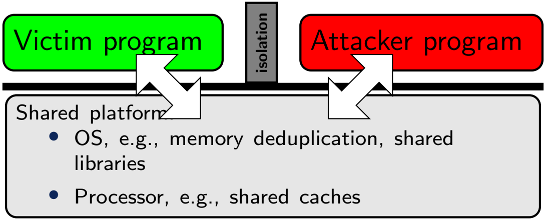


Hence, we have to consider attacks where the attacker has code running on the same platform.

Shared platform attacks

In the shared platform attack model, the attacker has the capability to run code on the same platform as the victim.

- Attacker and victim are isolated, e.g., through virtualization mechanisms.
- But the platform manages some resources shared between attacker and victim.
  - Optimizations in the management of resources are intended to be *transparent*.
  - But they can lead to side channels.



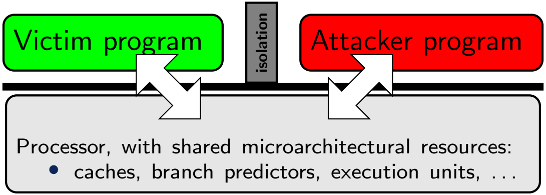
Shared platform:

- OS, e.g., memory deduplication, shared libraries
- Processor, e.g., shared caches

Microarchitectural attacks

Microarchitectural attacks are an important example of a class of attacks that are mainly relevant in the shared platform attack setting.

- Attacker and victim are *architecturally* isolated: separate memory and registers through some virtualization mechanism.
    - Isolation can be either symmetric or asymmetric.
- But they share *microarchitectural* resources.



Processor, with shared microarchitectural resources:

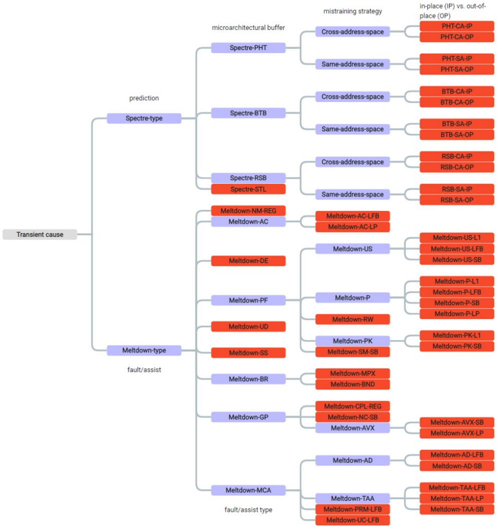
- caches, branch predictors, execution units, . . .

Microarchitectural attacks

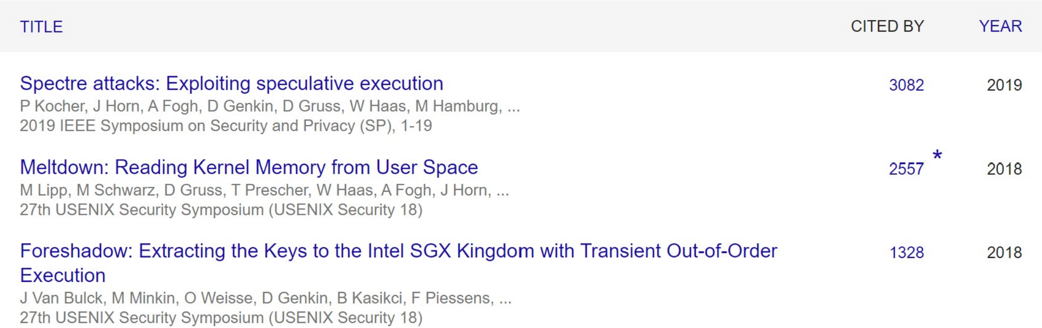
- Instances of microarchitectural side-channel attacks have been known for 20+ years, for instance, cache timing attacks.
    - Ge et al., *A survey of microarchitectural timing attacks and countermeasures on contemporary hardware*, J. Cryptographic Engineering, 2018
- The crypto community has developed solid countermeasures, for instance, constant-time programming.
    - Almeida et al., *Verifying Constant-Time Implementations*, USENIX Security 2016
- But in 2018, transient execution attacks were disclosed, a new class of powerful 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

Transient execution attacks: many variants

- A wide variety of instances of vulnerabilities that enable transient execution attacks have been found.
- On the right, an example classification tree
  - Originally from: Canella et al., *A Systematic Evaluation of Transient Execution Attacks and Defenses*, Usenix Security 2019.
  - Further extended and maintained at: `https://transient.fail/`



Transient execution attacks: academic and real-world 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	3082	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)	2557 *	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)	1328	2018

- 30+ CVE's in the first 5 years after discovery
- Mitigations in CPU microcode, operating systems and compilers

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Cache attacks

A cache is a smaller, faster memory situated between the processor and main memory. To speed up accesses to main memory, the cache *buffers* the contents of memory that the processor expects to access in the near future.
The state of the cache:

- is influenced by the memory addresses that the processor has accessed in the past, and
- can impact the time it takes to access memory in the future.

Hence, if an attacker and victim program share the same cache, the attacker can potentially infer something about memory addresses accessed by the victim, just by timing attacker memory accesses.
We will model a simple *prime+probe* attack.

Organization of a simple direct-mapped cache

A straightforward way to organize the cache is to use *direct mapping*:

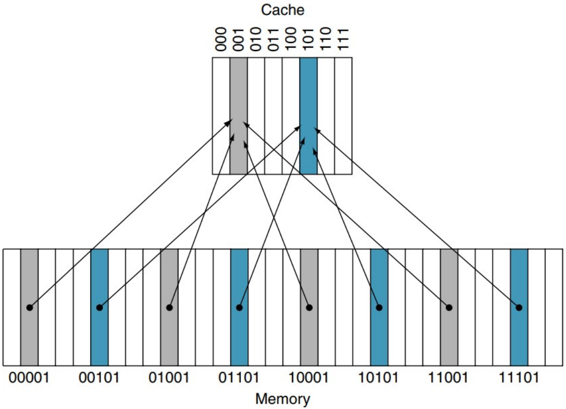
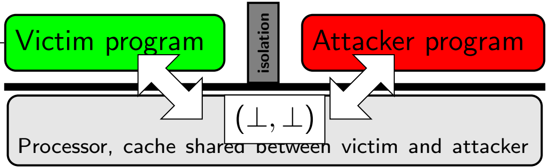


Figure from *Computer Organization and Design, RISC-V edition*, Patterson and Hennessy

Example cache attack

```
0 : beqz secret 3
1 : v ← load[13]
2 : jmp 4
3 : v ← load[14]
4 : ...
```

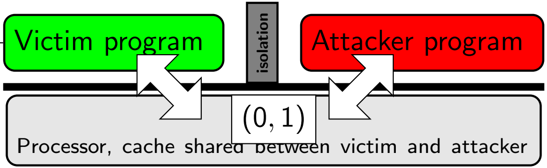


Assume memory addresses 0-9 belong to the attacker and 10-19 to the victim.
Assume a 2-element direct-mapped cache.
The attack proceeds as follows:

- Prime: the attacker accesses addresses 0 and 1 to get them into the cache.
- Now the victim code runs and evicts either address 0 or address 1 from the cache.
- Probe: the attacker measures access times to addresses 0 and 1.

Example cache attack

```
0 : beqz secret 3
1 : v ← load[13]
2 : jmp 4
3 : v ← load[14]
4 : ...
```

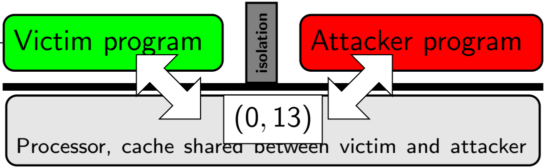


Assume memory addresses 0-9 belong to the attacker and 10-19 to the victim.
Assume a 2-element direct-mapped cache.
The attack proceeds as follows:

- Prime: the attacker accesses addresses 0 and 1 to get them into the cache.
- Now the victim code runs and evicts either address 0 or address 1 from the cache.
- Probe: the attacker measures access times to addresses 0 and 1.

Example cache attack

```
0 : beqz secret 3
1 : v ← load[13]
2 : jmp 4
3 : v ← load[14]
4 : ...
```



Assume memory addresses 0-9 belong to the attacker and 10-19 to the victim.
Assume a 2-element direct-mapped cache.
The attack proceeds as follows:

- Prime: the attacker accesses addresses 0 and 1 to get them into the cache.
- Now the victim code runs and evicts either address 0 or address 1 from the cache.
- Probe: the attacker measures access times to addresses 0 and 1.

Cache attacks and the constant time model

This simple example attack shows how information about the victim program execution leaks.

The constant time leakage model is a widely used (over)approximation of the information that can leak from a victim program. It assumes that:

- Memory *addresses* that are accessed leak (but not the values read/written from/to these addresses).
- The control flow (or, equivalently, the value of the program counter) leaks.
- Operands of instructions with data-dependent timing leak.

Hence, we can include cache attacks in our model by making all this information visible to the attacker, e.g., by storing it in a public part of the program state.

A program is *constant-time* (or complies with the *constant time policy*) if it is noninterferent in this extended model.

Examples

Obviously, the possibility of cache attacks impacts confidentiality properties of programs.

Insecure program

```
if secret then secret2 := 42
          else secret2 := 666
```

Secure version

```
secret2 :=
  secret * 42 + (1 - secret) * 666
```

Examples

Obviously, the possibility of cache attacks impacts confidentiality properties of programs.

Insecure program

```
if secret then secret2 := 42
          else secret2 := 666
```

```
secret := s[secret1]
```

Secure version

```
secret2 :=
   secret * 42 + (1 - secret) * 666
```

```
index := 0;
secret := 0;
while (index < sizeof(s))
  flag := index == secret1
  secret := secret + flag * s[index]
```

Examples

Obviously, the possibility of cache attacks impacts confidentiality properties of programs.

Insecure program

```
if secret then secret2 := 42
          else secret2 := 666
```

```
secret := s[secret1]
```

Secure version

```
secret2 :=
   secret * 42 + (1 - secret) * 666
```

```
index := 0;
secret := 0;
while (index < sizeof(s))
  flag := index == secret1
  secret := secret + flag * s[index]
```

Writing secure and efficient constant-time programs is an interesting challenge by itself, that we will not discuss further.
For a deeper discussion for the case of constant-time crypto code, see the relevant parts of:

- M. Barbosa et al., *SoK: Computer-Aided Cryptography*, IEEE S&P 2021

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Recall the ISA model

ISA program syntax:

$$\begin{array}{rcll}
\text{Register names} & r & \in & \text{Regs} \\
\text{Natural numbers} & n & \in & \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid r \mid e + e \mid e < e \mid \ldots \\
\text{Instructions} & i & ::= & r \leftarrow e \quad \mid \\
 & & & r \leftarrow \textbf{load}[e] \quad \mid \\
 & & & \textbf{store}[e] \leftarrow \ r \quad \mid \\
 & & & \textbf{jmp}\ e \quad \mid \\
 & & & \textbf{beqz}\ r\ n \\
\text{Programs} & p & ::= & [i_0, i_1, \ldots]
\end{array}$$

Program state:

$$\begin{array}{rcll}
\text{Register state} & \rho & \in & \text{Regs} \rightarrow \mathbb{N} \\
\text{Memory state} & m & ::= & [n_0, n_1, \ldots] \\
\text{Program counter} & pc & ::= & n \\
\text{Program state} & \sigma & ::= & (m, \rho, pc)
\end{array}$$

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, for instance for a slow memory load.
  - **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.

Modeling out-of-order and speculative execution

Extending program state:

$$
\begin{array}{rcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & | \\
 & & & r \leftarrow \textbf{load}[e] & | \\
 & & & \textbf{store}[e] \leftarrow \ r & | \\
 & & & pc \leftarrow n & | \\
 & & & @n' : pc \leftarrow n & | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}
$$

Modeling out-of-order and speculative execution

Extending program state:

$$\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & | \\
 & & & r \leftarrow \textbf{load}[e] & | \\
 & & & \textbf{store}[e] \leftarrow \ r & | \\
 & & & pc \leftarrow n & | \\
 & & & @n' : pc \leftarrow n & | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}$$

Example program:

$$\begin{array}{rcl}
0 & : & i \leftarrow 2 + 2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}$$

Modeling out-of-order and speculative execution

pc	0
i	0
n	0
r_0	0

→

Extending program state:

$$
\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & | \\
 & & & r \leftarrow \textbf{load}[e] & | \\
 & & & \textbf{store}[e] \leftarrow r & | \\
 & & & pc \leftarrow n & | \\
 & & & @n' : pc \leftarrow n & | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}
$$

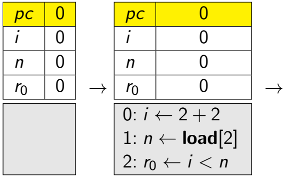
Example program:

$$
\begin{array}{rcl}
0 & : & i \leftarrow 2 + 2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz } r_0 \; 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}
$$

Example execution:

Memory state $= [5, 4, \mathbf{7}, \ldots]$

Modeling out-of-order and speculative execution



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2+2$

1: $n \leftarrow \textbf{load}[2]$

2: $r_0 \leftarrow i < n$

→

Extending program state:

$$\begin{array}{rrcl}
\text{In-flight instructions} & f & ::= & r \leftarrow e \quad | \\
 & & & r \leftarrow \textbf{load}[e] \quad | \\
 & & & \textbf{store}[e] \leftarrow r \quad | \\
 & & & pc \leftarrow n \quad | \\
 & & & @n' : pc \leftarrow n \quad | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu)
\end{array}$$

Example program:

$$\begin{array}{rcl}
0 & : & i \leftarrow 2+2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}$$

Example execution:

Memory state $= [5, 4, \mathbf{7}, \ldots]$

Modeling out-of-order and speculative execution

pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0
0: $i \leftarrow 2 + 2$ 1: $n \leftarrow \mathbf{load}[2]$ 2: $r_0 \leftarrow i < n$	

→

pc	0
i	0
n	0
r_0	0
0: $i \leftarrow 4$ 1: $n \leftarrow \mathbf{load}[2]$ 2: $r_0 \leftarrow i < n$	

→

Extending program state:

$$\begin{array}{rcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e \quad | \\
 & & & r \leftarrow \mathbf{load}[e] \quad | \\
 & & & \mathbf{store}[e] \leftarrow r \quad | \\
 & & & pc \leftarrow n \quad | \\
 & & & @n' : pc \leftarrow n \quad | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu)
\end{array}$$

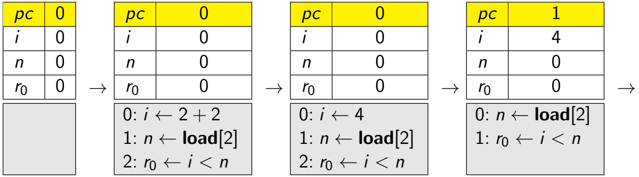
Example program:

$$\begin{array}{rcl}
0 & : & i \leftarrow 2 + 2 \\
1 & : & n \leftarrow \mathbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \mathbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}$$

Example execution:

Memory state $= [5, 4, \mathbf{7}, \ldots]$

Modeling out-of-order and speculative execution



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2 + 2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

Extending program state:

$$\begin{array}{rrcll} \text{In-flight instructions} & f & ::= & r \leftarrow e & | \\ & & & r \leftarrow \textbf{load}[e] & | \\ & & & \textbf{store}[e] \leftarrow r & | \\ & & & pc \leftarrow n & | \\ & & & @n' : pc \leftarrow n & | \\ \text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\ \text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) & \end{array}$$

Example program:

$$\begin{array}{rcl} 0 & : & i \leftarrow 2 + 2 \\ 1 & : & n \leftarrow \textbf{load}[2] \\ 2 & : & r_0 \leftarrow i < n \\ 3 & : & \textbf{beqz}\ r_0\ 5 \\ 4 & : & i \leftarrow 4 \times i \\ 5 & : & i \leftarrow i + 1 \end{array}$$

Example execution:

Memory state $= [5, 4, \mathbf{7}, \ldots]$

Modeling out-of-order and speculative execution

Extending program state:

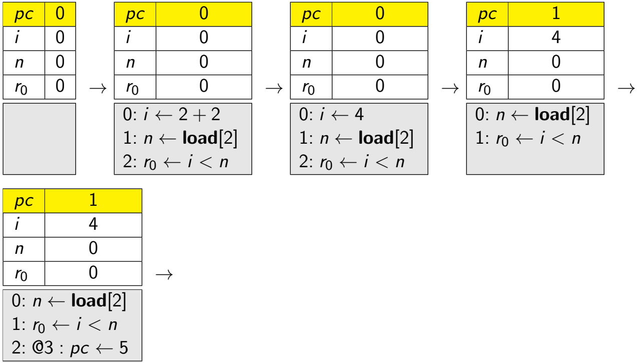
$$
\begin{array}{lrcl}
\text{In-flight instructions} & f & ::= & r \leftarrow e \quad | \\
 & & & r \leftarrow \mathbf{load}[e] \quad | \\
 & & & \mathbf{store}[e] \leftarrow r \quad | \\
 & & & pc \leftarrow n \quad | \\
 & & & @n' : pc \leftarrow n \quad | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu)
\end{array}
$$

Example program:

$$
\begin{array}{lcl}
0 & : & i \leftarrow 2+2 \\
1 & : & n \leftarrow \mathbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \mathbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i+1
\end{array}
$$

Example execution: INCORRECT prediction

Memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2+2$
1: $n \leftarrow \mathbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \mathbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \mathbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \mathbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$

→

Modeling out-of-order and speculative execution

Extending program state:

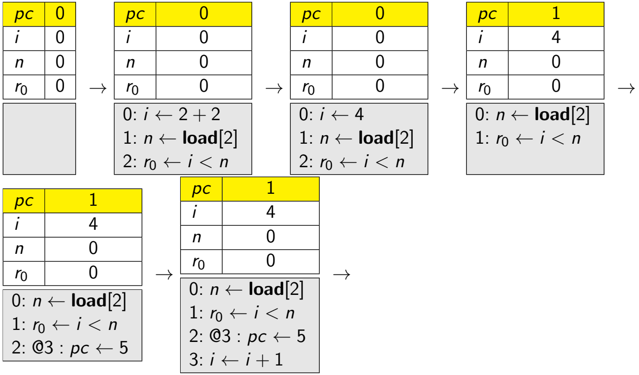
$$
\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & | \\
 & & & r \leftarrow \textbf{load}[e] & | \\
 & & & \textbf{store}[e] \leftarrow r & | \\
 & & & pc \leftarrow n & | \\
 & & & @n' : pc \leftarrow n & | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}
$$

Example program:

$$
\begin{array}{rcl}
0 & : & i \leftarrow 2 + 2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}
$$

Example execution: INCORRECT prediction

Memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2 + 2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow i + 1$

→

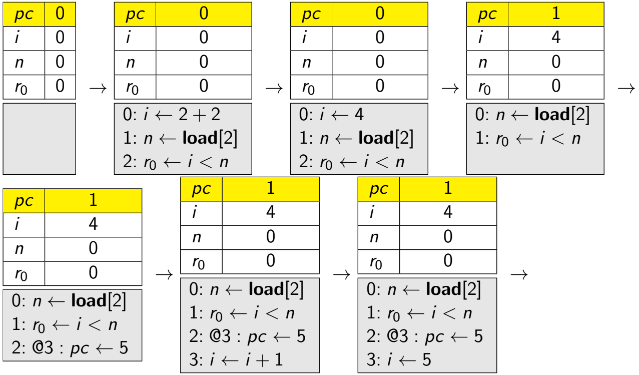
Modeling out-of-order and speculative execution

Extending program state:

$$\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow \; r & \mid \\
 & & & pc \leftarrow n & \mid \\
 & & & @n' : pc \leftarrow n & \mid \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}$$

Example program:

$$\begin{array}{rcl}
0 & : & i \leftarrow 2 + 2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz } r_0 \ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}$$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2 + 2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow i + 1$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow 5$

→

Example execution: INCORRECT prediction
Memory state $= [5, 4, \mathbf{7}, \ldots]$

Modeling out-of-order and speculative execution

Extending program state:

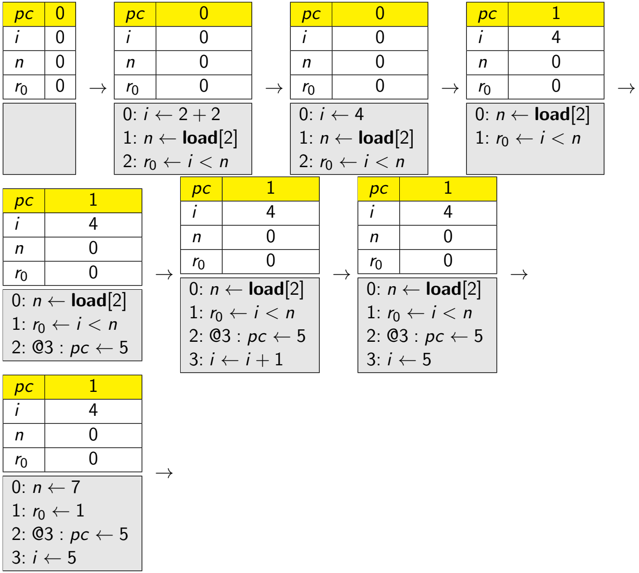
$$\begin{array}{rrcl}
\text{In-flight instructions} & f & ::= & r \leftarrow e \quad | \\
 & & & r \leftarrow \textbf{load}[e] \quad | \\
 & & & \textbf{store}[e] \leftarrow r \quad | \\
 & & & pc \leftarrow n \quad | \\
 & & & @n' : pc \leftarrow n \quad | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu)
\end{array}$$

Example program:

$$\begin{array}{rcl}
0 & : & i \leftarrow 2 + 2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}$$

Example execution: INCORRECT prediction

Memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2 + 2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow i + 1$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow 7$
1: $r_0 \leftarrow 1$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow 5$

→

Modeling out-of-order and speculative execution

Extending program state:

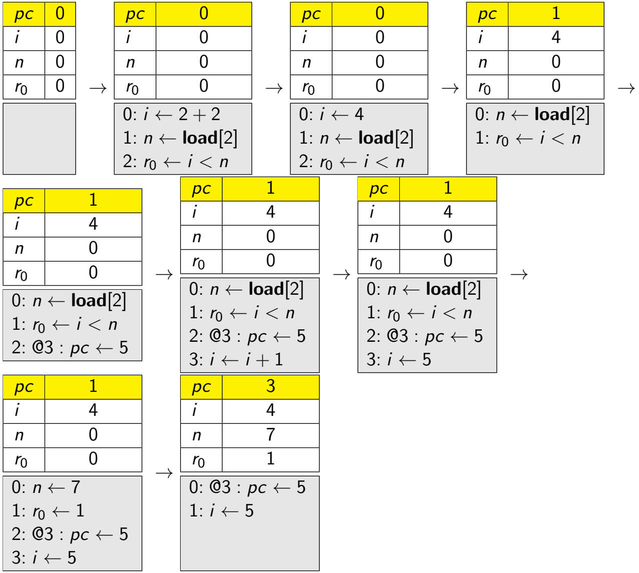
$$
\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & | \\
 & & & r \leftarrow \textbf{load}[e] & | \\
 & & & \textbf{store}[e] \leftarrow r & | \\
 & & & pc \leftarrow n & | \\
 & & & @n' : pc \leftarrow n & | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}
$$

Example program:

$$
\begin{array}{lcl}
0 & : & i \leftarrow 2+2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}
$$

Example execution: INCORRECT prediction

Memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2+2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow i + 1$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow 7$
1: $r_0 \leftarrow 1$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow 5$

→

pc	3
i	4
n	7
r_0	1

0: $@3 : pc \leftarrow 5$
1: $i \leftarrow 5$

Modeling out-of-order and speculative execution

Extending program state:

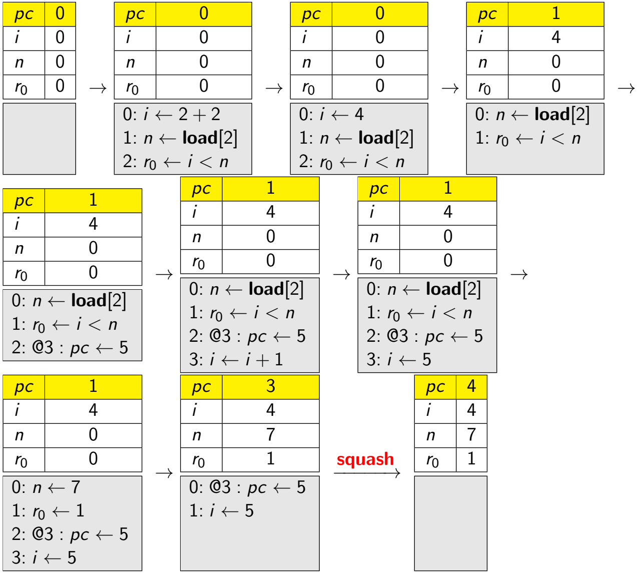
$$
\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & | \\
 & & & r \leftarrow \textbf{load}[e] & | \\
 & & & \textbf{store}[e] \leftarrow r & | \\
 & & & pc \leftarrow n & | \\
 & & & @n' : pc \leftarrow n & | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}
$$

Example program:

$$
\begin{array}{rcl}
0 & : & i \leftarrow 2+2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i+1
\end{array}
$$

Example execution: INCORRECT prediction

Memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2+2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow i+1$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow 5$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow 7$
1: $r_0 \leftarrow 1$
2: $@3 : pc \leftarrow 5$
3: $i \leftarrow 5$

→

pc	3
i	4
n	7
r_0	1

0: $@3 : pc \leftarrow 5$
1: $i \leftarrow 5$

$\xrightarrow{\textbf{squash}}$

pc	4
i	4
n	7
r_0	1

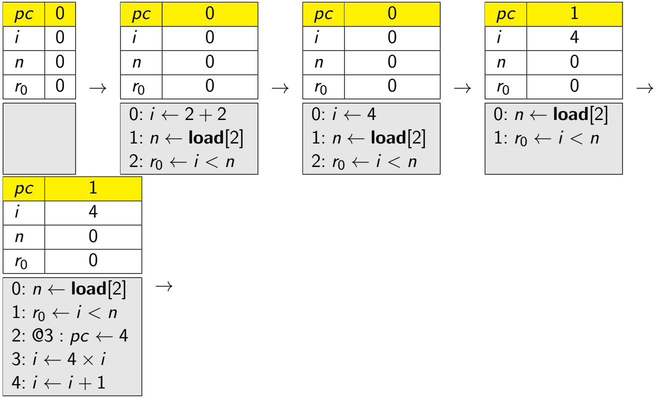
Modeling out-of-order and speculative execution

Extending program state:

$$\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \textbf{load}[e] & \mid \\
 & & & \textbf{store}[e] \leftarrow \ r & \mid \\
 & & & pc \leftarrow n & \mid \\
 & & & @n' : pc \leftarrow n & \mid \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}$$

Example program:

$$\begin{array}{rcl}
0 & : & i \leftarrow 2+2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i+1
\end{array}$$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2+2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 4 \times i$
4: $i \leftarrow i+1$

→

Example execution: CORRECT prediction
Memory state $= [5, 4, \mathbf{7}, \ldots]$

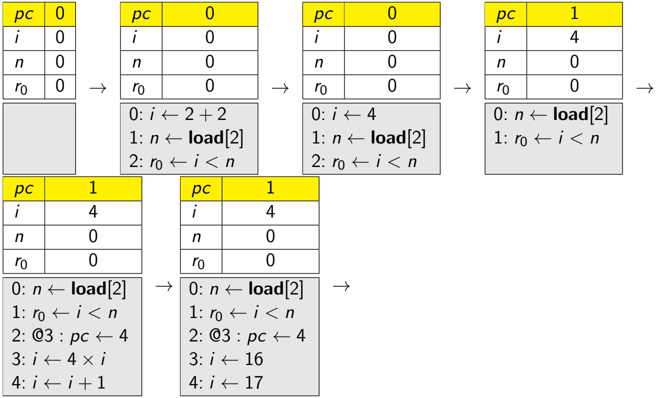
Modeling out-of-order and speculative execution

Extending program state:

$$\begin{array}{rrcll}
\text{In-flight instructions} & f & ::= & r \leftarrow e & \mid \\
 & & & r \leftarrow \mathbf{load}[e] & \mid \\
 & & & \mathbf{store}[e] \leftarrow r & \mid \\
 & & & pc \leftarrow n & \mid \\
 & & & @n' : pc \leftarrow n & \mid \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] & \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) &
\end{array}$$

Example program:

$$\begin{array}{rcl}
0 & : & i \leftarrow 2 + 2 \\
1 & : & n \leftarrow \mathbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \mathbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i + 1
\end{array}$$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2 + 2$
1: $n \leftarrow \mathbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \mathbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \mathbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

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

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \mathbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 16$
4: $i \leftarrow 17$

→

Example execution: CORRECT prediction
Memory state $= [5, 4, \mathbf{7}, \ldots]$

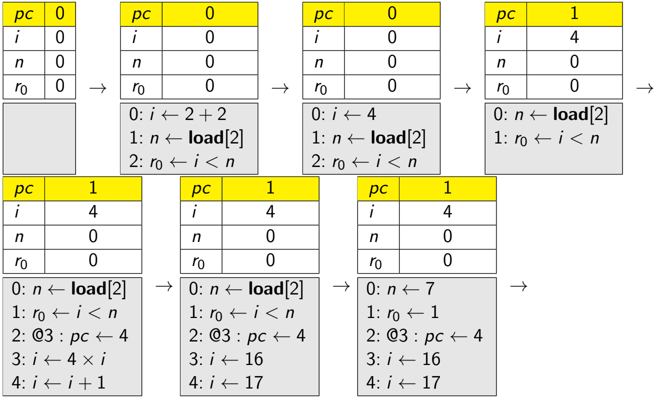
Modeling out-of-order and speculative execution

Extending program state:

$$
\begin{array}{rrcl}
\text{In-flight instructions} & f & ::= & r \leftarrow e \quad | \\
 & & & r \leftarrow \textbf{load}[e] \quad | \\
 & & & \textbf{store}[e] \leftarrow r \quad | \\
 & & & pc \leftarrow n \quad | \\
 & & & @n' : pc \leftarrow n \quad | \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu)
\end{array}
$$

Example program:

$$
\begin{array}{rcl}
0 & : & i \leftarrow 2+2 \\
1 & : & n \leftarrow \textbf{load}[2] \\
2 & : & r_0 \leftarrow i < n \\
3 & : & \textbf{beqz}\ r_0\ 5 \\
4 & : & i \leftarrow 4 \times i \\
5 & : & i \leftarrow i+1
\end{array}
$$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2+2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 4 \times i$
4: $i \leftarrow i+1$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 16$
4: $i \leftarrow 17$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow 7$
1: $r_0 \leftarrow 1$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 16$
4: $i \leftarrow 17$

→

Example execution: CORRECT prediction
Memory state $= [5, 4, \mathbf{7}, \ldots]$

Modeling out-of-order and speculative execution

Extending program state:

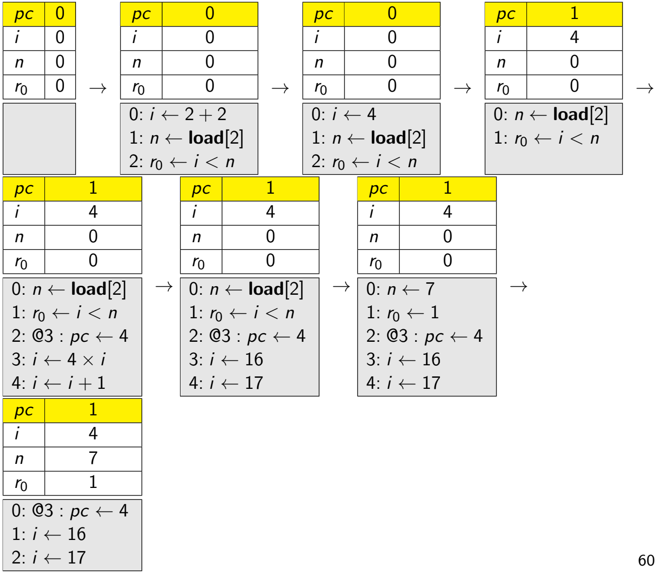
$$\begin{array}{rrcl} \text{In-flight instructions} & f & ::= & r \leftarrow e \quad | \\ & & & r \leftarrow \textbf{load}[e] \quad | \\ & & & \textbf{store}[e] \leftarrow r \quad | \\ & & & pc \leftarrow n \quad | \\ & & & @n' : pc \leftarrow n \quad | \\ \text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] \\ \text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu) \end{array}$$

Example program:

$$\begin{array}{rcl} 0 & : & i \leftarrow 2 + 2 \\ 1 & : & n \leftarrow \textbf{load}[2] \\ 2 & : & r_0 \leftarrow i < n \\ 3 & : & \textbf{beqz}\ r_0\ 5 \\ 4 & : & i \leftarrow 4 \times i \\ 5 & : & i \leftarrow i + 1 \end{array}$$

Example execution: CORRECT prediction

Memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2 + 2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 4 \times i$
4: $i \leftarrow i + 1$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 16$
4: $i \leftarrow 17$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow 7$
1: $r_0 \leftarrow 1$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 16$
4: $i \leftarrow 17$

→

pc	1
i	4
n	7
r_0	1

0: $@3 : pc \leftarrow 4$
1: $i \leftarrow 16$
2: $i \leftarrow 17$

Modeling out-of-order and speculative execution

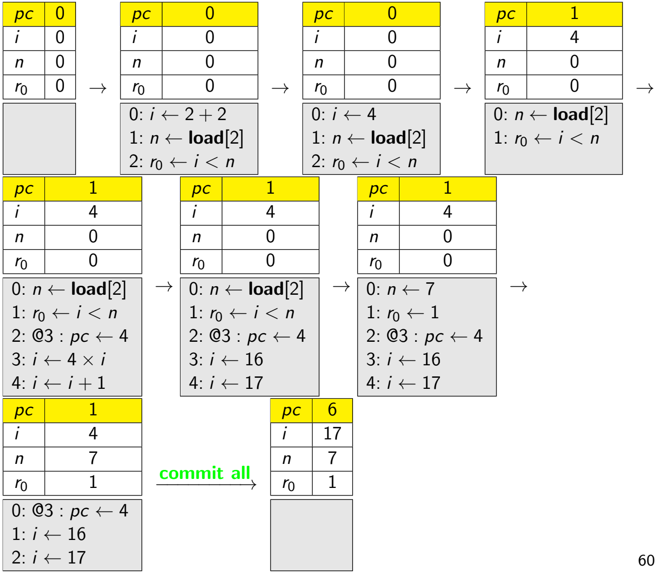
Extending program state:

$$\begin{array}{rrcl}
\text{In-flight instructions} & f & ::= & r \leftarrow e \qquad \mid \\
 & & & r \leftarrow \textbf{load}[e] \qquad \mid \\
 & & & \textbf{store}[e] \leftarrow r \qquad \mid \\
 & & & pc \leftarrow n \qquad \mid \\
 & & & @n' : pc \leftarrow n \qquad \mid \\
\text{Reorder buffer} & rob & ::= & [f_0, f_1, \ldots] \\
\text{Program state} & \sigma & ::= & (m, \rho, pc, rob, \mu)
\end{array}$$

Example program:

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

Example execution: CORRECT prediction
Memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	0
i	0
n	0
r_0	0

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 2 + 2$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	0
i	0
n	0
r_0	0

0: $i \leftarrow 4$
1: $n \leftarrow \textbf{load}[2]$
2: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 4 \times i$
4: $i \leftarrow i + 1$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 16$
4: $i \leftarrow 17$

→

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow 7$
1: $r_0 \leftarrow 1$
2: $@3 : pc \leftarrow 4$
3: $i \leftarrow 16$
4: $i \leftarrow 17$

→

pc	1
i	4
n	7
r_0	1

0: $@3 : pc \leftarrow 4$
1: $i \leftarrow 16$
2: $i \leftarrow 17$

$\xrightarrow{\text{commit all}}$

pc	6
i	17
n	7
r_0	1

Attack model

We must model two capabilities that attackers have on shared computation platforms:

- the attacker can perform classic microarchitectural attacks that leak victim information, e.g., a cache attack.
- the attacker can influence predictions and speculations, e.g., *training* a branch predictor.

Rather than building models of *how* to perform these attacks, we model their *effects* by explicitly including in the model what leaks to the attacker and what influence the attacker has.
This **simplifies** attacks and **overapproximates** attacker capabilities.

Attack model

The microarchitectural context μ models the capabilities of the attacker. It is an abstract *object* that receives anything the attacker can observe, and that is queried whenever the processor needs to make a prediction or scheduling decision.

$$\frac{\mu' = \mathrm{update}(\mu, \langle \mathit{leak} \rangle) \qquad d = \mathrm{next}(\mu') \qquad (m, \rho, \mathit{pc}, \mathit{rob}, \mu') \rightarrow^{d} (m', \rho', \mathit{pc}', \mathit{rob}', \mu'')}{(m, \rho, \mathit{pc}, \mathit{rob}, \mu) \rightarrow (m', \rho', \mathit{pc}', \mathit{rob}', \mu'')}$$

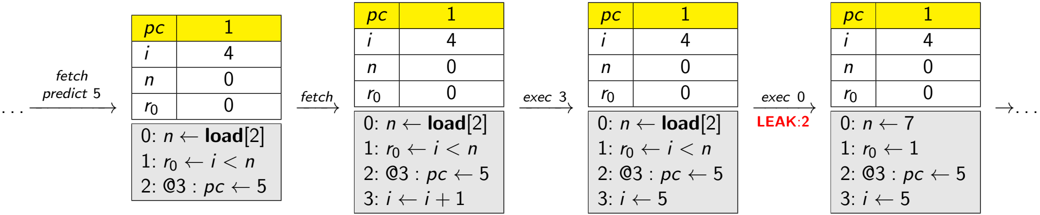
$$\frac{\mathit{spc} = \mathrm{apl}(\mathit{rob}, \mathit{pc}) \qquad P[\mathit{spc}] = \textbf{beqz}\ r\ n \qquad n' = \mathrm{predict}(\mu)}{(m, \rho, \mathit{pc}, \mathit{rob}, \mu') \rightarrow^{\mathrm{fetch}} (m, \rho, \mathit{pc}, \mathit{rob} + [@\mathit{spc} : \mathit{pc} \leftarrow n'], \mu)}$$

- Through update() calls, μ receives an (over)approximation of what leaks through microarchitectural side channel attacks.
- The semantics queries μ with next() or predict() to resolve all non-determinism related to scheduling or predictions.

Attack model: example

$$\begin{array}{lcl} 0 & : & i \leftarrow 2+2 \\ 1 & : & n \leftarrow \textbf{load}[2] \\ 2 & : & r_0 \leftarrow i < n \\ 3 & : & \textbf{beqz}\ r_0\ 5 \\ 4 & : & i \leftarrow 4 \times i \\ 5 & : & i \leftarrow i + 1 \end{array}$$

Example execution with memory state $= [5, 4, \mathbf{7}, \ldots]$



pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: @3 : $pc \leftarrow 5$

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: @3 : $pc \leftarrow 5$
3: $i \leftarrow i + 1$

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow \textbf{load}[2]$
1: $r_0 \leftarrow i < n$
2: @3 : $pc \leftarrow 5$
3: $i \leftarrow 5$

pc	1
i	4
n	0
r_0	0

0: $n \leftarrow 7$
1: $r_0 \leftarrow 1$
2: @3 : $pc \leftarrow 5$
3: $i \leftarrow 5$

Attack model: discussion

- Our attack model simplifies and overapproximates what a shared platform attacker can do and learn.
- This is convenient for reasoning about defenses: showing a processor to be secure under a model where the attacker *automatically* gets any information that could possibly leak, and where the attacker can *fully* determine any processor choice, provides more confidence than under weaker attack models.
    - However, it also makes attacks look much simpler than they are in practice. The original Spectre, Meltdown and Foreshadow papers explain how to set up real attacks on specific processors.
- In example code, we will use a pseudo-instruction **leak** r to represent an instruction sequence that leaks the contents of r.

  For concreteness, we define **leak** r to be $d \leftarrow \textbf{load}[r]$ where d is some dummy register not used in the program, but it could be any instruction or instruction sequence that leaks r.

Transient execution attacks

We have seen that a processor sometimes executes *transient* instructions: instructions that are executed out-of-order, but squashed before they ever impact the architectural state. The existence of transient instructions has two important consequences for security:

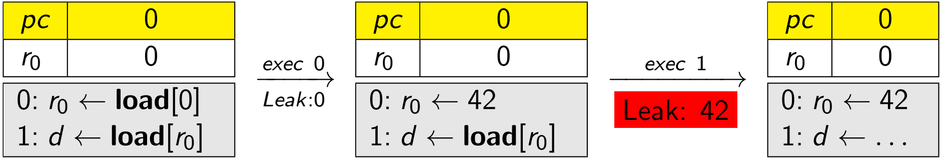
- transient instructions leak information in line with the leakage model: e.g., executing a load *transiently* will impact the cache, and hence leak the memory address that is accessed.
  This is surprising and counterintuitive: instructions that one does not expect to execute, can actually leak information.
- transient instructions can *access* information that is protected. This occurs in two different ways, leading to *Meltdown-style* or *Spectre-style* attacks.

Meltdown

Suppose some memory addresses are *architecturally* inaccessible (for instance, kernel memory inaccessible to user programs).
Consider the following program:

$$\begin{array}{lll} 0: & r_0 \leftarrow \mathbf{load}[0] & \textit{; this raises a fault} \\ 1: & \mathbf{leak}\ r_0 & \textit{; expands to } d \leftarrow \mathbf{load}[r_0] \end{array}$$

On a Meltdown vulnerable processor, values loaded from inaccessible memory addresses can be accessible to transient instructions, and hence be leaked.



pc	0
r_0	0

0: $r_0 \leftarrow \mathbf{load}[0]$
1: $d \leftarrow \mathbf{load}[r_0]$

$\xrightarrow[\mathit{Leak}:0]{\mathit{exec}\ 0}$

pc	0
r_0	0

0: $r_0 \leftarrow 42$
1: $d \leftarrow \mathbf{load}[r_0]$

$\xrightarrow{\mathit{exec}\ 1}$

Leak: 42

pc	0
r_0	0

0: $r_0 \leftarrow 42$
1: $d \leftarrow \dots$

Hence, user code can just read kernel memory.

Spectre

In a Spectre-style attack, transient instructions access information that is protected *in software*.
We will discuss a couple of Spectre examples. For each example:

- There is code operating on 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
  In particular, all the example programs are constant time programs.
- Yet, because of speculation and out-of-order execution, the secrets do leak

Spectre V1 (Spectre-PHT)

```
0 : $len \leftarrow \textbf{load}[a - 1]$    ; $\textit{length field}$
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$                  ; $\textbf{if}\ (i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a + i]$    ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : $\ldots$                                 ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	

Spectre V1 (Spectre-PHT)

```
0 : $len \leftarrow$ load$[a-1]$      ; length field
1 : $r_0 \leftarrow i < len$
2 : beqz $r_0$ 5               ; if $(i < len)\{$
3 : $r_0 \leftarrow$ load$[a+i]$      ;     $r_0 = a[i]$
4 : leak $r_0$
5 : ...                       ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	

pc	0
a	1
len	0
i	2
r_0	0

Spectre V1 (Spectre-PHT)

```
0 : $len \leftarrow \textbf{load}[a-1]$     ; length field
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$               ; if $(i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a+i]$      ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : ...                      ; $\}$
```

	Address	Value	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	

pc	0
a	1
len	0
i	2
r_0	0

fetch
fetch
fetch
pred 3
fetch
fetch
→

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow \textbf{load}[a+i]$
4: $d \leftarrow \textbf{load}[r_0]$

Spectre V1 (Spectre-PHT)

```
0 : $len \leftarrow \textbf{load}[a-1]$   ; $length\ field$
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$               ; $\textbf{if}\ (i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a+i]$   ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : ...                              ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	

pc	0
a	1
len	0
i	2
r_0	0

fetch
fetch
fetch
pred 3
fetch
fetch →

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow \textbf{load}[a+i]$
4: $d \leftarrow \textbf{load}[r_0]$

$\xrightarrow[Leak:3]{exec\ 3}$

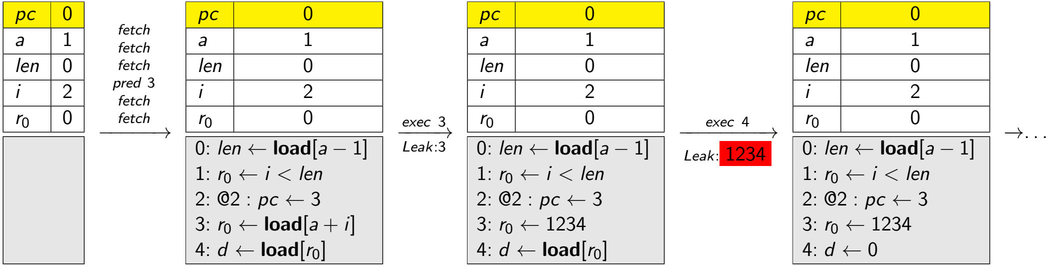
pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow 1234$
4: $d \leftarrow \textbf{load}[r_0]$

Spectre V1 (Spectre-PHT)

```
0 : $len \leftarrow \textbf{load}[a-1]$     ; length field
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$                  ; $\textbf{if}\ (i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a+i]$     ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : ...                                      ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	🟥 1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	



pc	0
a	1
len	0
i	2
r_0	0

fetch
fetch
fetch
pred 3
fetch
fetch →

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow \textbf{load}[a+i]$
4: $d \leftarrow \textbf{load}[r_0]$

exec 3 →
Leak:3

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow 1234$
4: $d \leftarrow \textbf{load}[r_0]$

exec 4 →
Leak: 🟥 1234

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow 1234$
4: $d \leftarrow 0$

→...

Spectre V2 (Spectre-BTB)

```
0   :  $r_0 \leftarrow$ load[7]   ; load a secret into $r_0$
1   :  $fp \leftarrow$ load[8]    ; load a "function pointer" to a trusted function
2   :  jmp $fp$              ; call trusted function that safely accesses secret
... :  ...
20  :  $r_0 \leftarrow 0$          ; trusted function just clears secret
21  :  jmp 3
... :  ...
31  :  leak $r_0$
... :  ...
```

Address	*Value*	
. . .	. . .	
1234 :	0	
. . .	. . .	
8 :	20	
7 :	1234	← *SECRET*
. . .	. . .	
0 :	0	

Spectre V2 (Spectre-BTB)

```
0   :  $r_0 \leftarrow$ load[7]    ; load a secret into $r_0$
1   :  $fp \leftarrow$ load[8]    ; load a "function pointer" to a trusted function
2   :  jmp $fp$              ; call trusted function that safely accesses secret
... :  ...
20  :  $r_0 \leftarrow 0$            ; trusted function just clears secret
21  :  jmp 3
... :  ...
31  :  leak $r_0$
... :  ...
```

pc	0
fp	0
r_0	0

Address	*Value*	
. . .	. . .	
1234 :	0	
. . .	. . .	
8 :	20	
7 :	1234	← *SECRET*
. . .	. . .	
0 :	0	

Spectre V2 (Spectre-BTB)

0	:	$r_0 \leftarrow$ **load**$[7]$	*; load a secret into* r_0
1	:	$fp \leftarrow$ **load**$[8]$	*; load a "function pointer" to a trusted function*
2	:	**jmp** fp	*; call trusted function that safely accesses secret*
...	:	...	
20	:	$r_0 \leftarrow 0$	*; trusted function just clears secret*
21	:	**jmp** 3	
...	:	...	
31	:	**leak** r_0	
...	:	...	

Address	*Value*	
...	...	
1234 :	0	
...	...	
8 :	20	
7 :	1234	← *SECRET*
...	...	
0 :	0	

pc	0
fp	0
r_0	0

fetch
fetch
fetch
pred 31
fetch
$\longrightarrow$

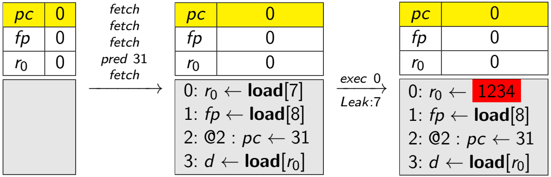
pc	0
fp	0
r_0	0

0: $r_0 \leftarrow$ **load**$[7]$
1: $fp \leftarrow$ **load**$[8]$
2: $@2 : pc \leftarrow 31$
3: $d \leftarrow$ **load**$[r_0]$

Spectre V2 (Spectre-BTB)

```
0   :  $r_0 \leftarrow$ load[7]   ; load a secret into $r_0$
1   :  $fp \leftarrow$ load[8]    ; load a "function pointer" to a trusted function
2   :  jmp $fp$                   ; call trusted function that safely accesses secret
... :  ...
20  :  $r_0 \leftarrow 0$         ; trusted function just clears secret
21  :  jmp 3
... :  ...
31  :  leak $r_0$
... :  ...
```

Address	*Value*	
. . .	. . .	
1234 :	0	
. . .	. . .	
8 :	20	
7 :	1234	← *SECRET*
. . .	. . .	
0 :	0	



pc	0
fp	0
r_0	0

fetch
fetch
fetch
pred 31
fetch
⟶

pc	0
fp	0
r_0	0

0: $r_0 \leftarrow$ **load**[7]
1: $fp \leftarrow$ **load**[8]
2: @2 : $pc \leftarrow 31$
3: $d \leftarrow$ **load**$[r_0]$

exec 0
⟶
Leak:7

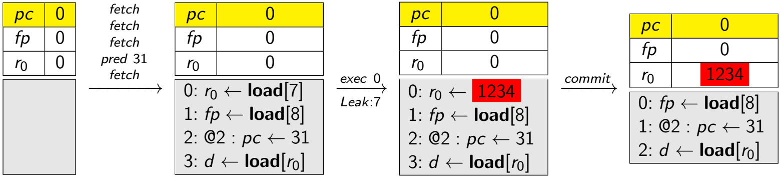
pc	0
fp	0
r_0	0

0: $r_0 \leftarrow$ 1234
1: $fp \leftarrow$ **load**[8]
2: @2 : $pc \leftarrow 31$
3: $d \leftarrow$ **load**$[r_0]$

Spectre V2 (Spectre-BTB)

```
0   :  $r_0 \leftarrow \textbf{load}[7]$    ; load a secret into $r_0$
1   :  $fp \leftarrow \textbf{load}[8]$     ; load a "function pointer" to a trusted function
2   :  jmp $fp$                              ; call trusted function that safely accesses secret
... :  ...
20  :  $r_0 \leftarrow 0$                    ; trusted function just clears secret
21  :  jmp 3
... :  ...
31  :  leak $r_0$
... :  ...
```

Address	*Value*	
...	...	
1234 :	0	
...	...	
8 :	20	
7 :	1234	← *SECRET*
...	...	
0 :	0	



pc	0
fp	0
r_0	0

fetch
fetch
fetch
pred 31
fetch
→

pc	0
fp	0
r_0	0

0: $r_0 \leftarrow \textbf{load}[7]$
1: $fp \leftarrow \textbf{load}[8]$
2: @2 : $pc \leftarrow 31$
3: $d \leftarrow \textbf{load}[r_0]$

$\xrightarrow[\textit{Leak}:7]{\textit{exec}\ 0}$

pc	0
fp	0
r_0	0

0: $r_0 \leftarrow$ 1234
1: $fp \leftarrow \textbf{load}[8]$
2: @2 : $pc \leftarrow 31$
3: $d \leftarrow \textbf{load}[r_0]$

$\xrightarrow{\textit{commit}}$

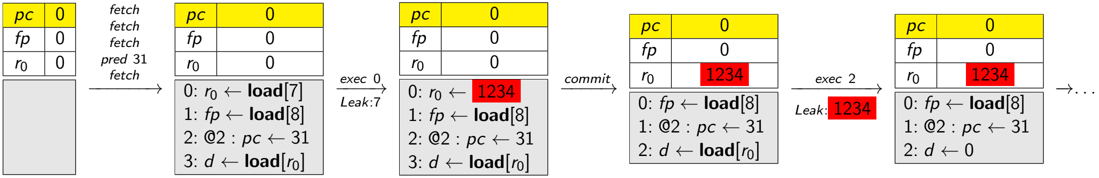
pc	0
fp	0
r_0	1234

0: $fp \leftarrow \textbf{load}[8]$
1: @2 : $pc \leftarrow 31$
2: $d \leftarrow \textbf{load}[r_0]$

Spectre V2 (Spectre-BTB)

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

Address	*Value*	
. . .	. . .	
1234 :	0	
. . .	. . .	
8 :	20	
7 :	1234	← *SECRET*
. . .	. . .	
0 :	0	



pc	0
fp	0
r_0	0

$\xrightarrow{\text{fetch, fetch, fetch, pred 31, fetch}}$

pc	0
fp	0
r_0	0

0: r_0 ← **load**[7]
1: *fp* ← **load**[8]
2: @2 : *pc* ← 31
3: *d* ← **load**[r_0]

$\xrightarrow[\text{Leak:7}]{\text{exec 0}}$

pc	0
fp	0
r_0	0

0: r_0 ← 1234
1: *fp* ← **load**[8]
2: @2 : *pc* ← 31
3: *d* ← **load**[r_0]

$\xrightarrow{\text{commit}}$

pc	0
fp	0
r_0	1234

0: *fp* ← **load**[8]
1: @2 : *pc* ← 31
2: *d* ← **load**[r_0]

$\xrightarrow[\text{Leak: 1234}]{\text{exec 2}}$

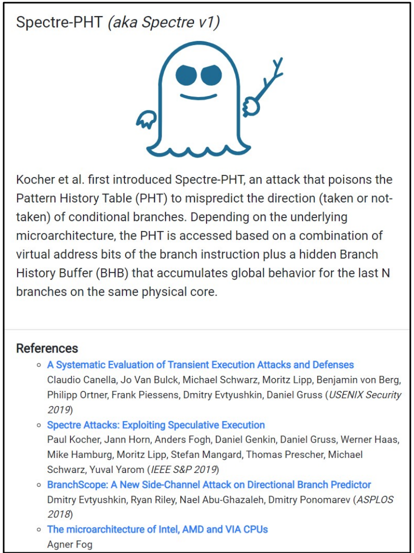
pc	0
fp	0
r_0	1234

0: *fp* ← **load**[8]
1: @2 : *pc* ← 31
2: *d* ← 0

→. . .

Example attacks: discussion

- For attacks beyond basic Meltdown and Spectre, and for pointers to full descriptions of the attacks:
  - See: `https://transient.fail/`
- It is worth emphasizing that these attacks undermine all isolation / sandboxing mechanisms.
  - Meltdown broke user/kernel isolation.
  - Spectre broke software based sandboxing (and other isolation mechanisms).
  - Foreshadow broke enclave and VM isolation.
  - . . .



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

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Defending against transient execution attacks

- It is generally accepted that Meltdown-style vulnerabilities should be fixed in hardware.
- However, defending against Spectre purely in hardware seems to be too expensive.
- How can we write software on out-of-order and speculative processors, that computes on secrets, yet does not leak these secrets to a shared platform attacker? We will discuss two approaches:
  - The use of *fences* as one of the state-of-practice methods.
  - A representative proposal for a hardware-software co-design from research.

Spectre V1 with speculation fences

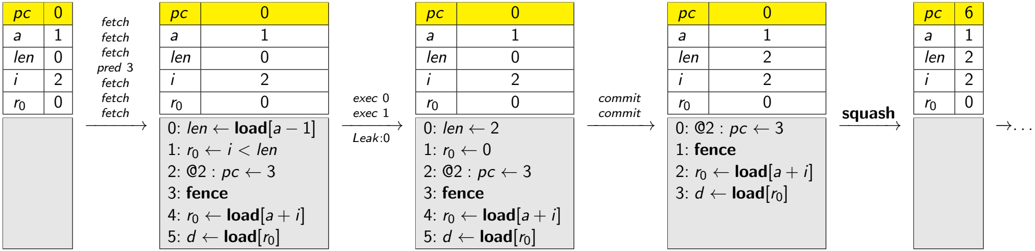
```
0 : $len \leftarrow$ **load**$[a - 1]$      ; *length field*
1 : $r_0 \leftarrow i < len$
2 : **beqz** $r_0$ 6                ; **if** $(i < len)\{$
3 : **fence**
4 : $r_0 \leftarrow$ **load**$[a + i]$      ;     $r_0 = a[i]$
5 : **leak** $r_0$
6 : . . .                    ; $\}$
```

	Address	*Value*	
	. . .	. . .	
	1234 :	0	
	. . .	. . .	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	

Spectre V1 with speculation fences

```
0 : $len \leftarrow$ **load**$[a-1]$   ; length field
1 : $r_0 \leftarrow i < len$
2 : **beqz** $r_0$ 6                   ; **if** $(i < len)\{$
3 : **fence**
4 : $r_0 \leftarrow$ **load**$[a+i]$   ;    $r_0 = a[i]$
5 : **leak** $r_0$
6 : ...                                ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	



pc	0
a	1
len	0
i	2
r_0	0

$\xrightarrow{\text{fetch, fetch, fetch, pred 3, fetch, fetch, fetch}}$

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow$ **load**$[a-1]$
1: $r_0 \leftarrow i < len$
2: $@2 : pc \leftarrow 3$
3: **fence**
4: $r_0 \leftarrow$ **load**$[a+i]$
5: $d \leftarrow$ **load**$[r_0]$

$\xrightarrow[\text{Leak}:0]{\text{exec 0, exec 1}}$

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow 2$
1: $r_0 \leftarrow 0$
2: $@2 : pc \leftarrow 3$
3: **fence**
4: $r_0 \leftarrow$ **load**$[a+i]$
5: $d \leftarrow$ **load**$[r_0]$

$\xrightarrow{\text{commit, commit}}$

pc	0
a	1
len	2
i	2
r_0	0

0: $@2 : pc \leftarrow 3$
1: **fence**
2: $r_0 \leftarrow$ **load**$[a+i]$
3: $d \leftarrow$ **load**$[r_0]$

$\xrightarrow{\textbf{squash}}$

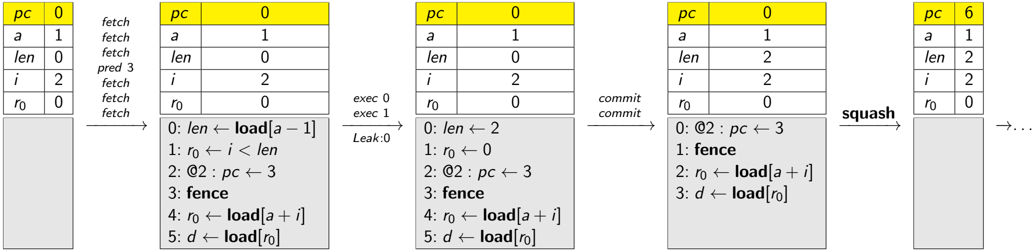
pc	6
a	1
len	2
i	2
r_0	0

$\rightarrow \dots$

Spectre V1 with speculation fences

```
0 : $len \leftarrow$ load$[a-1]$   ; length field
1 : $r_0 \leftarrow i < len$
2 : beqz $r_0$ 6                   ; if $(i < len)\{$
3 : fence
4 : $r_0 \leftarrow$ load$[a+i]$   ;     $r_0 = a[i]$
5 : leak $r_0$
6 : ...                            ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	



pc	0
a	1
len	0
i	2
r_0	0

fetch
fetch
fetch
pred 3
fetch
fetch
fetch
⟶

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow$ **load**$[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: **fence**
4: $r_0 \leftarrow$ **load**$[a+i]$
5: $d \leftarrow$ **load**$[r_0]$

exec 0
exec 1
⟶
Leak:0

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow 2$
1: $r_0 \leftarrow 0$
2: @2 : $pc \leftarrow 3$
3: **fence**
4: $r_0 \leftarrow$ **load**$[a+i]$
5: $d \leftarrow$ **load**$[r_0]$

commit
commit
⟶

pc	0
a	1
len	2
i	2
r_0	0

0: @2 : $pc \leftarrow 3$
1: **fence**
2: $r_0 \leftarrow$ **load**$[a+i]$
3: $d \leftarrow$ **load**$[r_0]$

squash
⟶

pc	6
a	1
len	2
i	2
r_0	0

⟶ ...

But note that always fencing this is inefficient, and selective fencing is hard.

Hardware/software co-designs

Pure software defenses are important for legacy processors
But better defenses can be obtained by HW/SW codesign

Definition

A processor provides secure speculation for a policy P if enforcing P based on the architectural semantics implies that P holds when executing on the actual processor.

I.e., as far as P is concerned, one can ignore speculation. Good designs exist for:

- The sandboxing policy, see:
  - Guarnieri et al., *Hardware/software contracts for secure speculation*, IEEE S&P 2021
- The constant-time policy, see:
  - Fustos et al., *SpectreGuard: An Efficient Data-Centric Defense Mechanism against Spectre Attacks*, DAC 2019
  - Schwarz et al., *ConTExT: A Generic Approach for Mitigating Spectre*, NDSS 2020.
  - Choudhary et al., *Speculative Privacy Tracking (SPT)*, MICRO 2021
  - **Daniel et al., ProSpeCT: Provably Secure Speculation for the Constant-Time Policy, Usenix Security 2023**

The ProSpeCT defense

```
0 : $len \leftarrow \textbf{load}[a-1]$   ; $\textit{length field}$
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$               ; $\textbf{if}\ (i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a+i]$   ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : $\ldots$                              ; $\}$
```

Code should be constant time

The ProSpeCT defense

```
0 : $len \leftarrow \textbf{load}[a-1]$   ; length field
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$               ; $\textbf{if}\ (i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a+i]$   ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : ...                                    ; $\}$
```

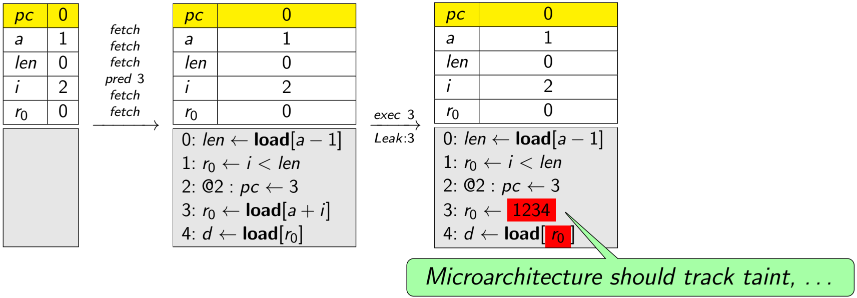
	Address	Value	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← SECRET
	2 :	5	
a :	1 :	3	
len :	0 :	2	

Memory should be partitioned in secret/public

The ProSpeCT defense

```
0 : $len \leftarrow$ load$[a-1]$   ; length field
1 : $r_0 \leftarrow i < len$
2 : beqz $r_0$ 5               ; if $(i < len)\{$
3 : $r_0 \leftarrow$ load$[a+i]$   ;     $r_0 = a[i]$
4 : leak $r_0$
5 : ...                        ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	🟥 1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	



pc	0
a	1
len	0
i	2
r_0	0

$\xrightarrow{\text{fetch fetch fetch pred 3 fetch fetch}}$

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow$ **load**$[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow$ **load**$[a+i]$
4: $d \leftarrow$ **load**$[r_0]$

$\xrightarrow[\text{Leak:3}]{\text{exec 3}}$

pc	0
a	1
len	0
i	2
r_0	0

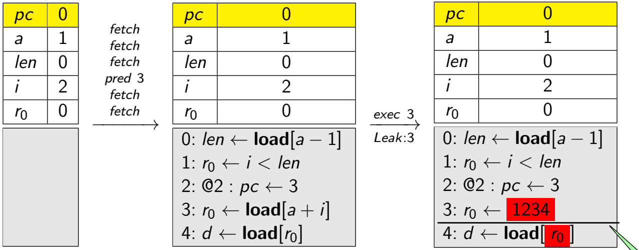
0: $len \leftarrow$ **load**$[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow$ 🟥 1234
4: $d \leftarrow$ **load**[🟥 r_0]

Microarchitecture should track taint, . . .

The ProSpeCT defense

```
0 : $len \leftarrow \textbf{load}[a-1]$      ; length field
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$                  ; if $(i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a+i]$      ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : ...                                      ; }
```

	Address	*Value*
	...	...
	1234 :	0
	...	...
	3 :	🟥 1234 ← *SECRET*
	2 :	5
a :	1 :	3
len :	0 :	2



pc	0
a	1
len	0
i	2
r_0	0

$\xrightarrow{\textit{fetch}\ \textit{fetch}\ \textit{fetch}\ \textit{pred}\ 3\ \textit{fetch}\ \textit{fetch}}$

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: $@2 : pc \leftarrow 3$
3: $r_0 \leftarrow \textbf{load}[a+i]$
4: $d \leftarrow \textbf{load}[r_0]$

$\xrightarrow[\textit{Leak}:3]{\textit{exec}\ 3}$

pc	0
a	1
len	0
i	2
r_0	0

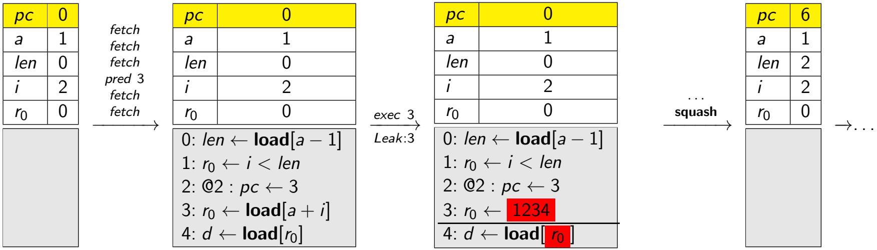
0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: $@2 : pc \leftarrow 3$
3: $r_0 \leftarrow$ 🟥 1234
4: $d \leftarrow \textbf{load}[$🟥 $r_0]$

. . . , and fence execution before it reaches a leaky instruction

The ProSpeCT defense

```
0 : $len \leftarrow \textbf{load}[a-1]$      ; length field
1 : $r_0 \leftarrow i < len$
2 : $\textbf{beqz}\ r_0\ 5$                  ; $\textbf{if}\ (i < len)\{$
3 : $r_0 \leftarrow \textbf{load}[a+i]$      ;     $r_0 = a[i]$
4 : $\textbf{leak}\ r_0$
5 : ...                                      ; $\}$
```

	Address	*Value*	
	...	...	
	1234 :	0	
	...	...	
	3 :	1234	← *SECRET*
	2 :	5	
a :	1 :	3	
len :	0 :	2	



pc	0
a	1
len	0
i	2
r_0	0

$\xrightarrow{\text{fetch, fetch, fetch, pred 3, fetch, fetch}}$

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow \textbf{load}[a+i]$
4: $d \leftarrow \textbf{load}[r_0]$

$\xrightarrow[\text{Leak:3}]{\text{exec 3}}$

pc	0
a	1
len	0
i	2
r_0	0

0: $len \leftarrow \textbf{load}[a-1]$
1: $r_0 \leftarrow i < len$
2: @2 : $pc \leftarrow 3$
3: $r_0 \leftarrow$ 1234
4: $d \leftarrow \textbf{load}[r_0]$

$\xrightarrow[]{\text{. . . }\ \textbf{squash}}$

pc	6
a	1
len	2
i	2
r_0	0

→ . . .

Summary of results on ProSpeCT

- A proof that ProSpeCT-compliant hardware provides secure speculation for the constant-time policy
  - Secure against all known Spectre variants
  - Supports declassification of secrets
- An implementation in an out-of-order RISC-V core
  - First synthesizable implementation of a speculating core that is Spectre resistant for constant-time code
  - Publicly available: `https://github.com/proteus-core/prospect`
- An experimental evaluation

Setting	25**S**/75**C**	50**S**/50**C**	75**S**/25**C**	90**S**/10**C**
baseline	100%	100%	100%	100%
P(`key`)	100%	100%	100%	100%
P(`all`)	110%	125%	136%	145%

For details see the Usenix 2023 paper, extended version available at: `https://arxiv.org/abs/2302.12108`

Overview

1. Introduction

1.1 Computer security

1.2 Software security

1.3 Some key system engineering techniques: abstraction and resource sharing

1.4 Conclusions and outlook

2. From Structured Programming Language to ISA

2.1 A simple model for an imperative language

2.2 A simple ISA model

3. Below the ISA: microarchitectural attacks

3.1 The rise of microarchitectural attacks

3.2 Cache attacks

3.3 Transient execution attacks

4. Defenses

4.1 Using fences

4.2 ProSpeCT: a hardware/software codesign

5. Conclusions

Conclusions

- *Functionality* of software systems is usually studied most naturally at a single level of abstraction.
- It is tempting to also study *security* at that same level of abstraction.
- However, in many cases practical attacks exist that exploit aspects of the system that are hidden at that level of abstraction.
  - This is particularly true for confidentiality properties because "abstractions are leaky" (Butler Lampson, Hints and Principles for Computer System Design)
- A very interesting challenge is extending functional specifications of abstractions with "security specifications" that allow multiple abstraction layers to contribute to security.
  - This will require "vertical integration" across abstraction layers (Hennessy and Patterson, A New Golden Age for Computer Architecture)

QUESTIONS?