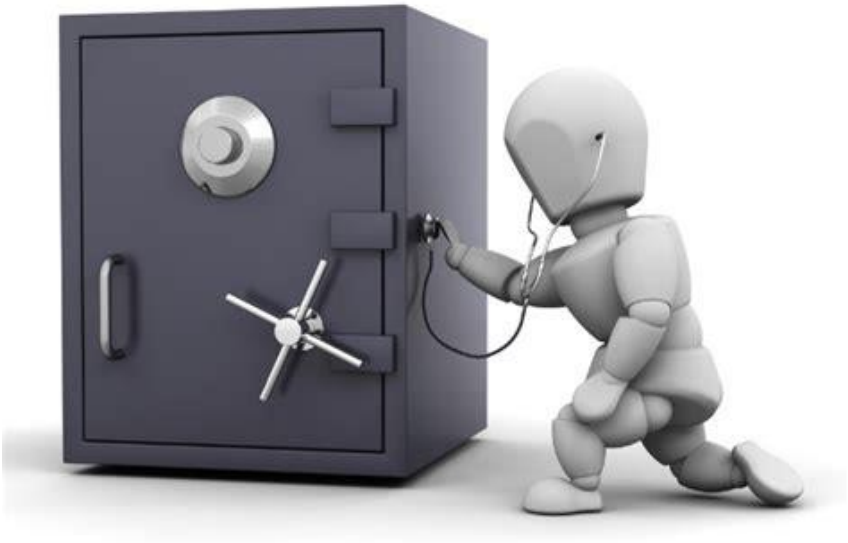
# Formal methods to protect against Microarchitectural attacks



PEPR Sécurité – Winter school

Lesly-Ann Daniel – KU Leuven

# Who am I?

**2018–2021**

## Phd Student

- Symbolic Binary-Level Code Analysis for Security
- CEA List & Université Côte d'Azur
- *Sébastien Bardin and Tamara Rezk*

**2021**

## Postdoc

- Hardware /Software co-Designs for Microarchitectural Security
- KU Leuven (Belgium)
- *Frank Piessens*

# Outline

1. **Microarchitectural side-channel attacks**
   - What are microarchitectural side-channel attacks?
   - How can formal methods help to protect against them?

2. **Spectre attacks**
   - Hardware optimizations can leak secrets!
   - Model the microarchitecture with formal methods?

3. **Mind the gap: model <> HW**
   - HW/SW contracts to help bridging the gap

# PART 1

## Microarchitectural side-channel attacks

*How formal methods can help you protect your secrets from the vagaries of time*

# What are side-channels?

# Programs manipulate secret data

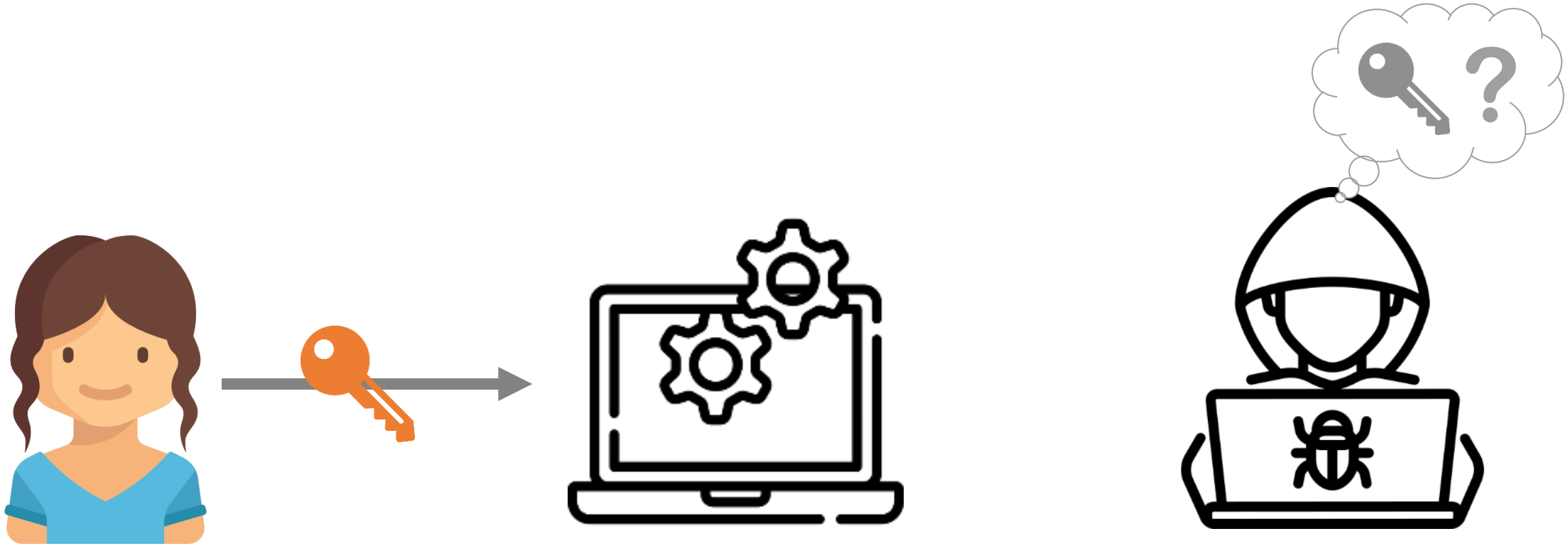
**Critical software is prevalent:**

- Secure communications
- Online banking
- Protect health data

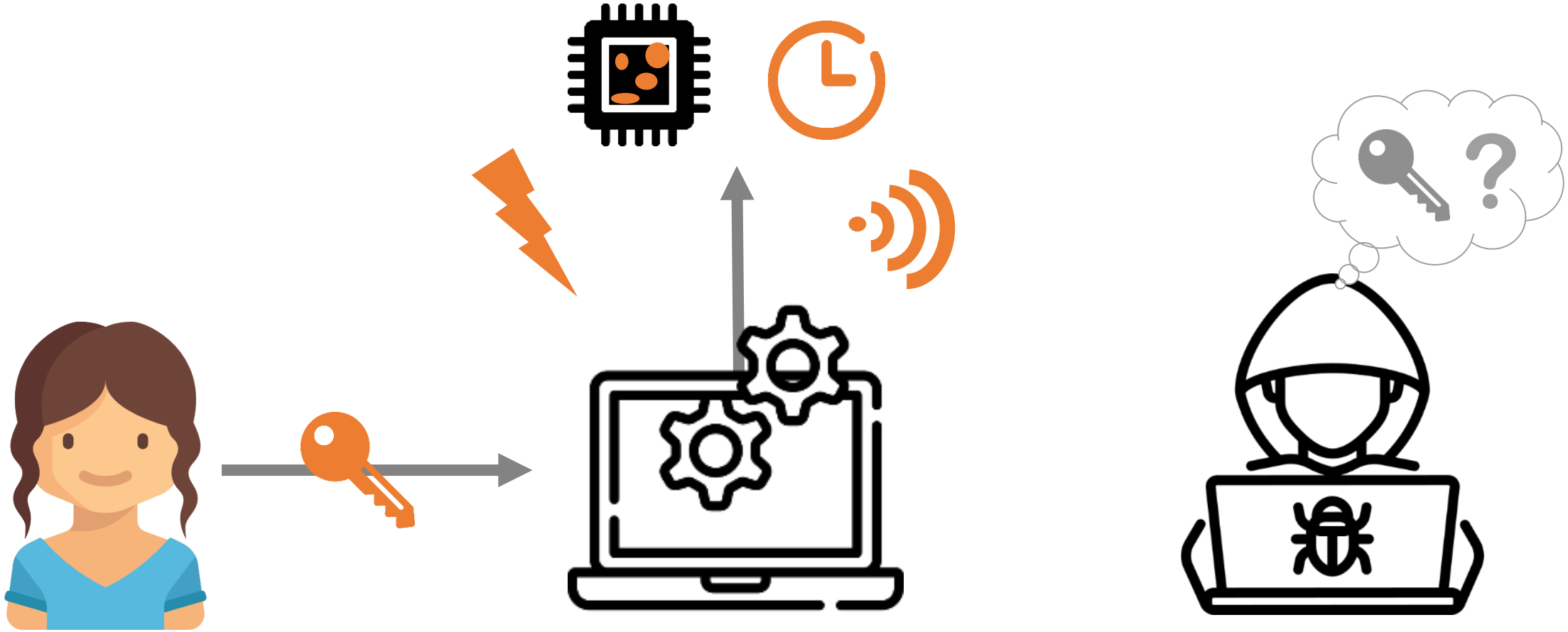

**Their security relies on cryptography:**

- Mathematical guarantees
- Verified implementations (no bugs, functional)
- *But what about their execution in the physical world?*

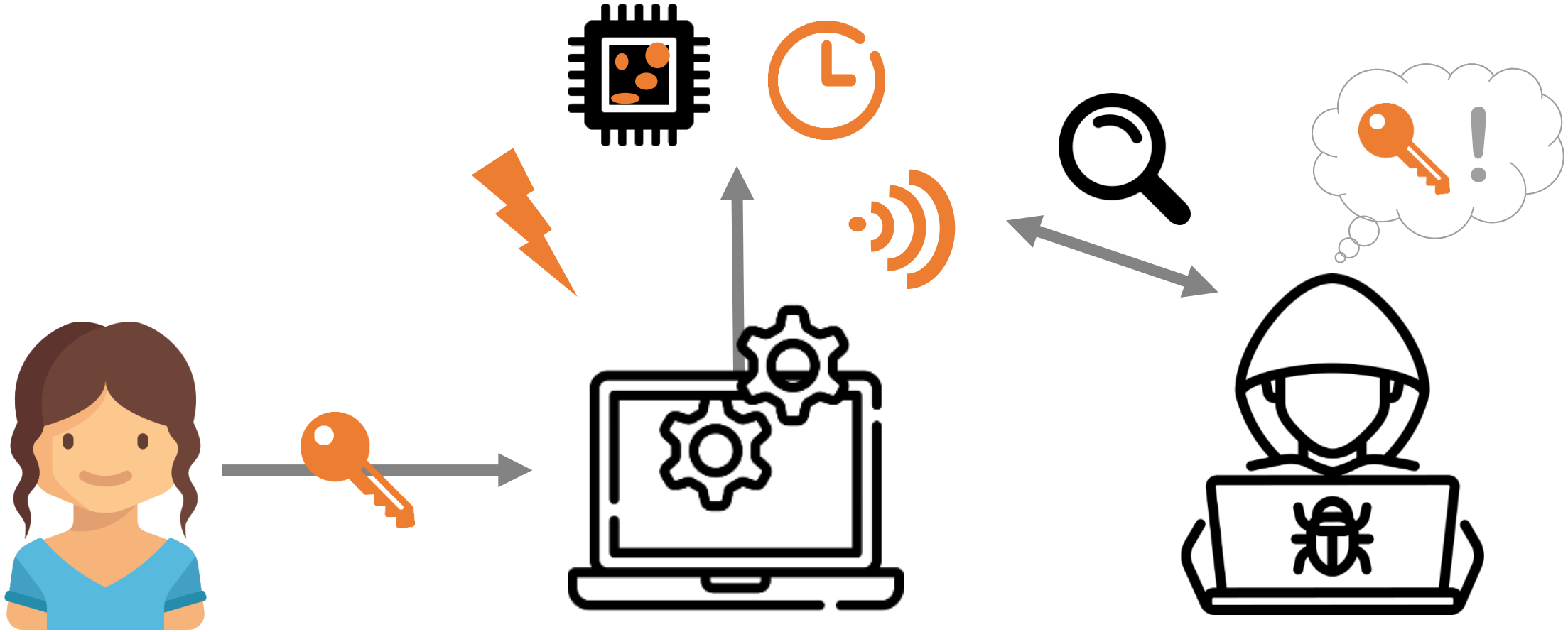
# … that can be observed by attackers!



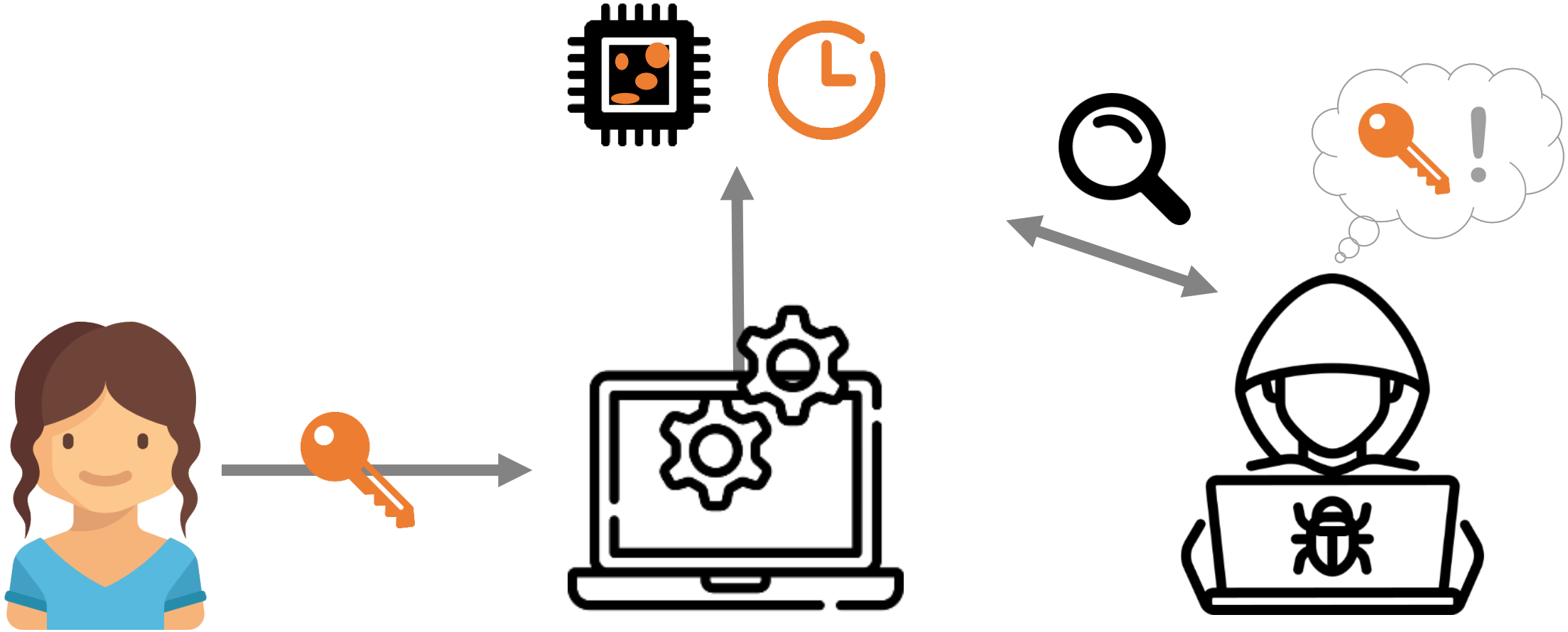
# … that can be observed by attackers!



# … that can be observed by attackers!



# … that can be observed by attackers!

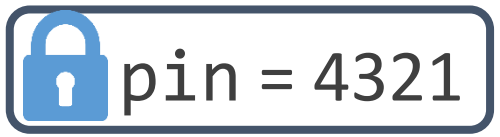


*Timing and microarchitectural attacks can be run remotely* [1]

[1] Remote Timing Attacks Are Practical, David Brumley and Dan Boneh at USENIX 2003

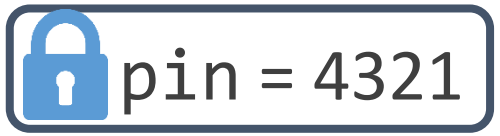
# Concrete example

```
bool check_pin(char* guess) {
  for (i=0; i<4; i++)
    if (guess[i] != pin[i])
      return false;
  return true;
}
```




# Concrete example

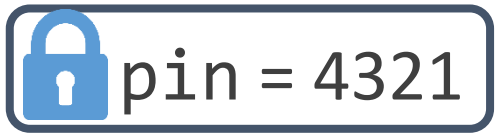
```
bool check_pin(char* guess) {
  for (i=0; i<4; i++)
    if (guess[i] != pin[i])
      return false;
  return true;
}
```




0000 → 1s
1000 → 1s
2000 → 1s
3000 → 1s
4000 → 2s
5000 → 1s
…

# Concrete example

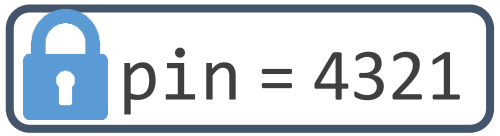
```
bool check_pin(char* guess) {
  for (i=0; i<4; i++)
    if (guess[i] != pin[i])
      return false;
  return true;
}
```




0000 → 1s
1000 → 1s
2000 → 1s
3000 → 1s
4000 → 2s
5000 → 1s
…

# Concrete example

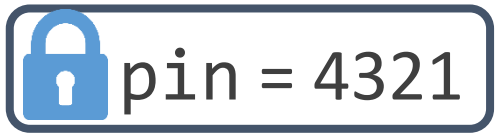
```
bool check_pin(char* guess) {
  for (i=0; i<4; i++)
    if (guess[i] != pin[i])
      return false;
  return true;
}
```




4000 → 2s
4100 → 2s
4200 → 2s
4300 → 3s
4400 → 2s
4500 → 2s
…

# Concrete example

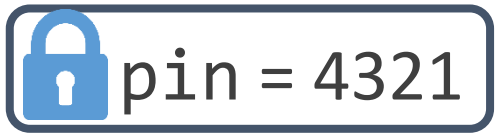
```
bool check_pin(char* guess) {
  for (i=0; i<4; i++)
    if (guess[i] != pin[i])
      return false;
  return true;
}
```




4000 → 2s
4100 → 2s
4200 → 2s
4300 → 3s
4400 → 2s
4500 → 2s
…

# Concrete example

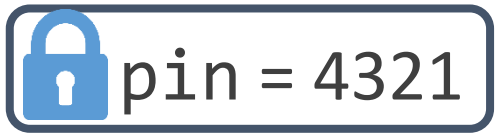
```
bool check_pin(char* guess) {
  for (i=0; i<4; i++)
    if (guess[i] != pin[i])
      return false;
  return true;
}
```




4000 → 2s
4100 → 2s
4200 → 2s
4300 → 3s
4400 → 2s
4500 → 2s
…

# Concrete example

```
bool check_pin(char* guess) {
  for (i=0; i<4; i++)
    if (guess[i] != pin[i])
      return false;
  return true;
}
```




**Attack Complexity**:
from $10^4$
to $10 \times 4$

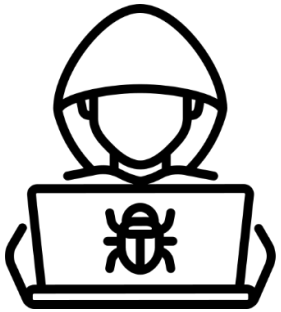
# Countermeasure

```
bool check_pin(char* guess) {
  good = true;
  for (i=0; i<4; i++)
    good &= guess[i] == pin[i];
  return good;
}
```

Make timing independent of secret
Remove secret-dependent branch!

# More generally, how can secret leak?

```
if secret
then  foo()  → trace
else  bar()  → trace'
```



trace → secret

trace' → ~~secret~~

**Control-flow leaks**

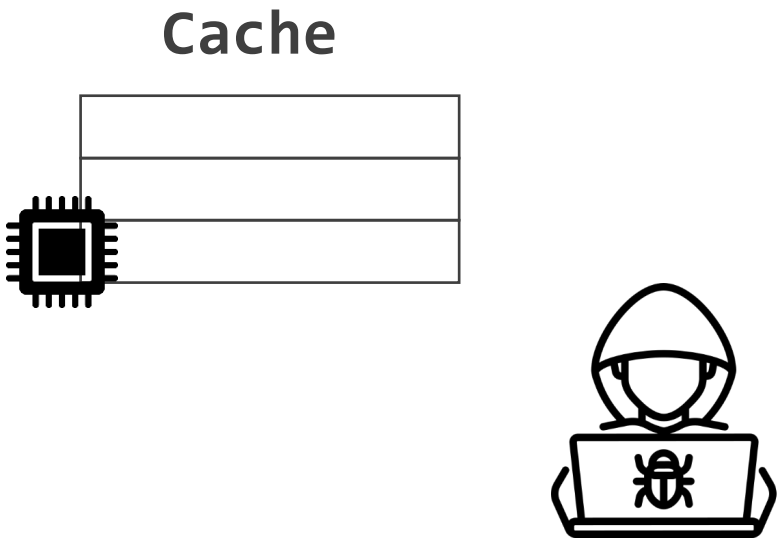
- end-to-end timing
- different resource consumption
- branch predictor state
- instruction cache
- instruction prefetcher
- micro-op cache
- …

# More generally, how can secret leak?

**Memory accesses leak**

- caches
- data pre-fetchers
- load/store dependencies
- …

```
x = tab[secret]
```

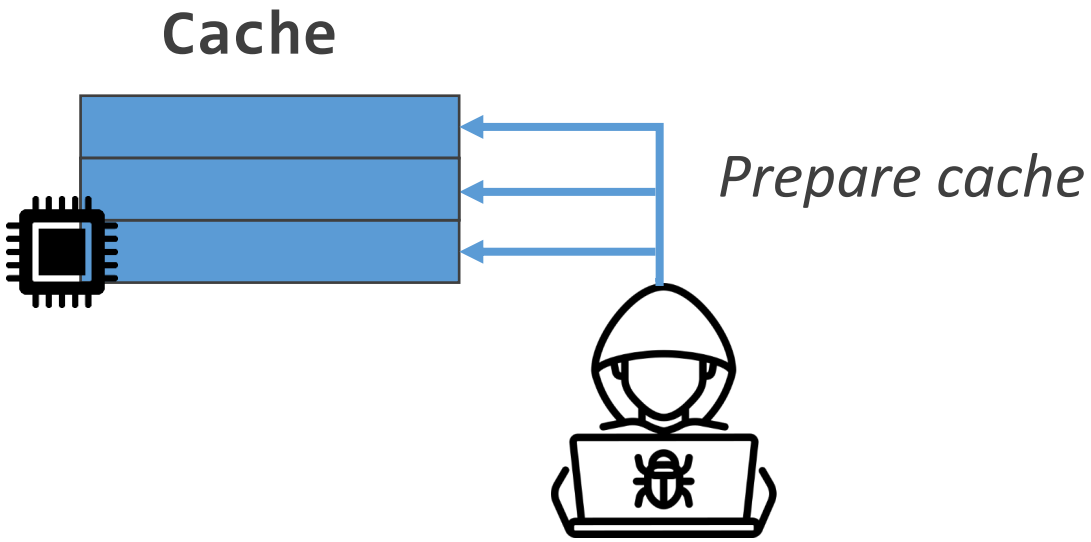


# More generally, how can secret leak?

**Memory accesses leak**

- caches
- data pre-fetchers
- load/store dependencies
- …

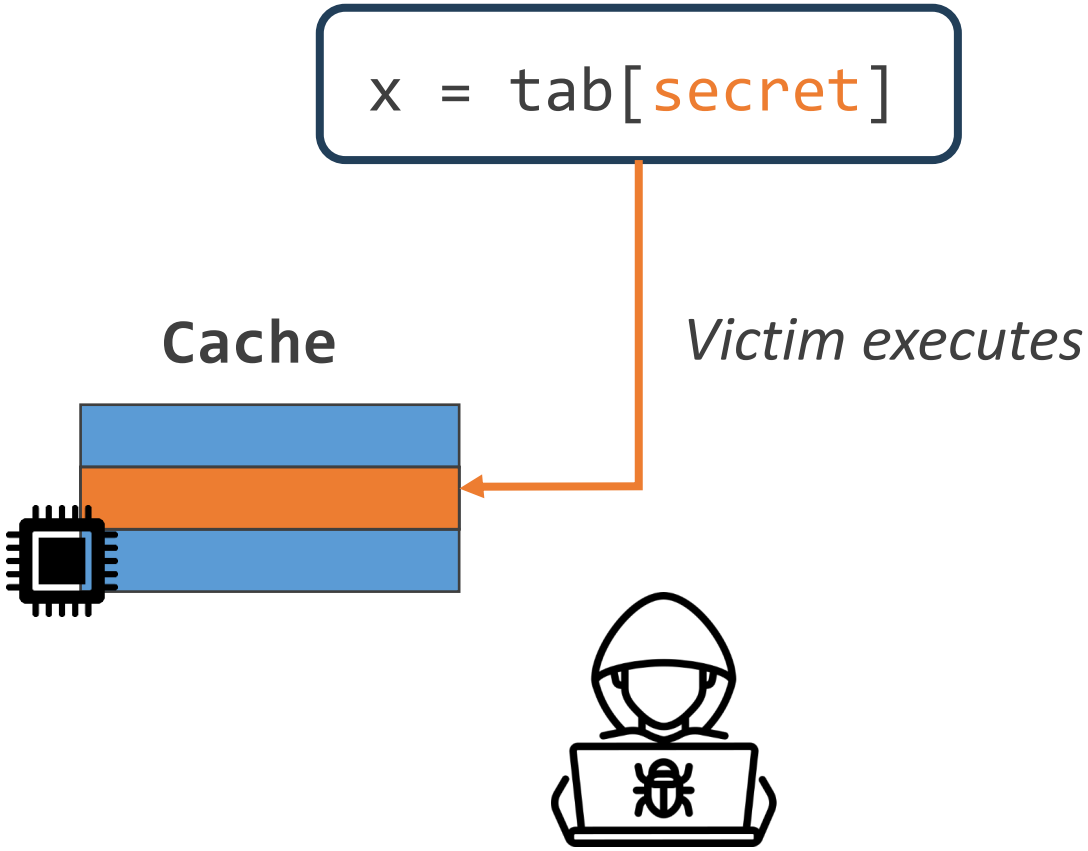
```
x = tab[secret]
```



# More generally, how can secret leak?

**Memory accesses leak**

- caches
- data pre-fetchers
- load/store dependencies
- …

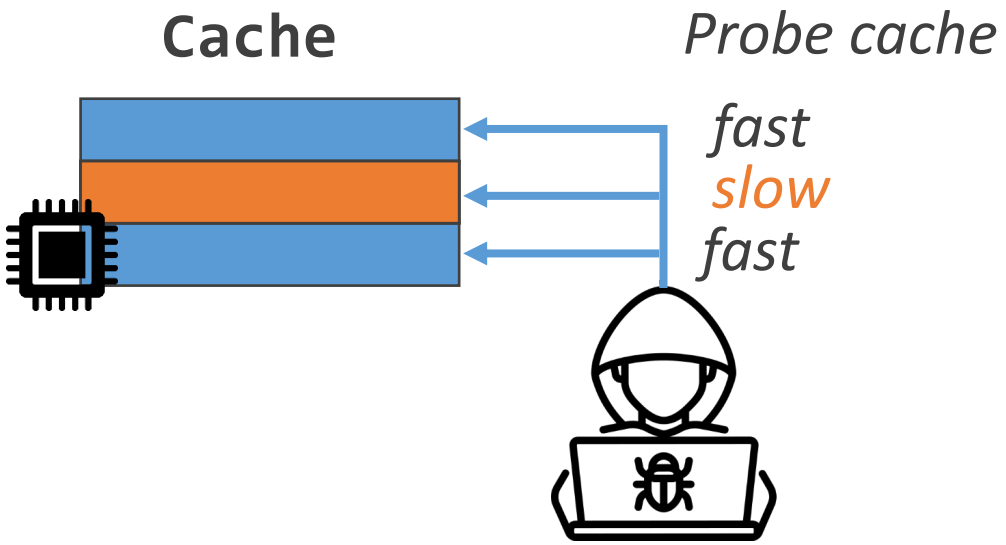


# More generally, how can secret leak?

**Memory accesses leak**

- caches
- data pre-fetchers
- load/store dependencies
- …

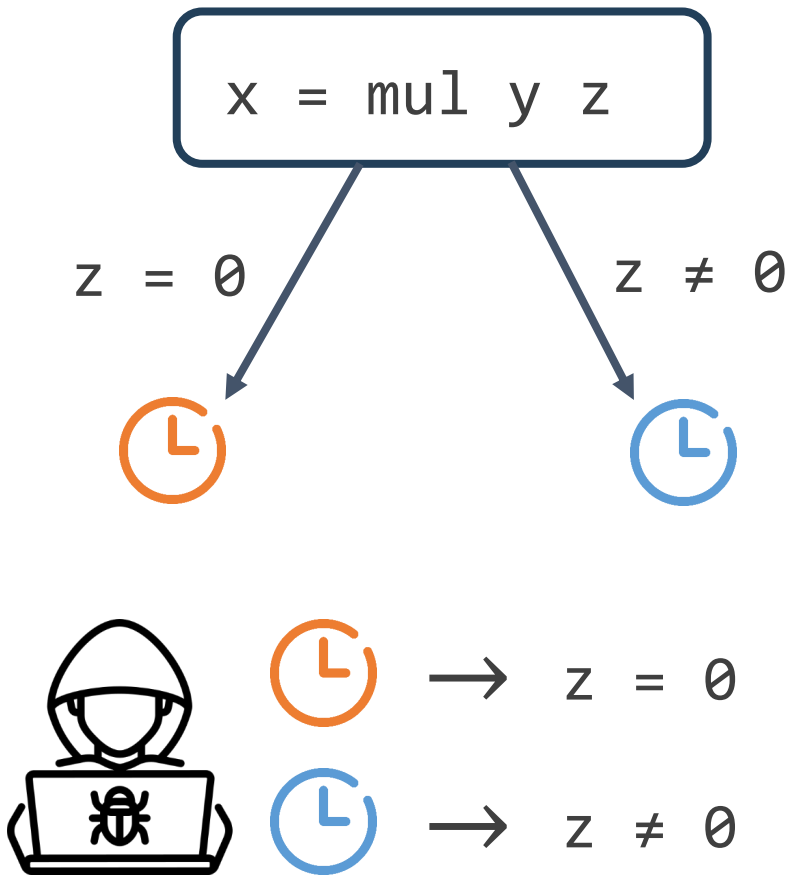
```
x = tab[secret]
```



# More generally, how can secret leak?

**Variable time instructions leak**

- divisions
- multiplication
- depends on microarchitecture
- …



# Why does it matter?

## Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems

Paul C. Kocher

Cryptography Research, Inc.
607 Market Street, 5th Floor, San Francisco, CA 94105, USA.
E-mail: paul@cryptography.com.

**Abstract.** By carefully measuring the amount of time required to perform private key operations, attackers may be able to find fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems. Against a vulnerable system, the attack is computationally inexpensive and often requires only known ciphertext. Actual systems are potentially at risk, including cryptographic tokens, network-based cryptosystems, and other applications where attackers can make reasonably accurate timing measurements. Techniques for preventing the attack for RSA and Diffie-Hellman are presented. Some cryptosystems will need to be revised to protect against the attack, and new protocols and algorithms may need to incorporate measures to prevent timing attacks.

1996

## Cache-timing attacks on AES

Daniel J. Bernstein *

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607–7045
djb@cr.yp.to

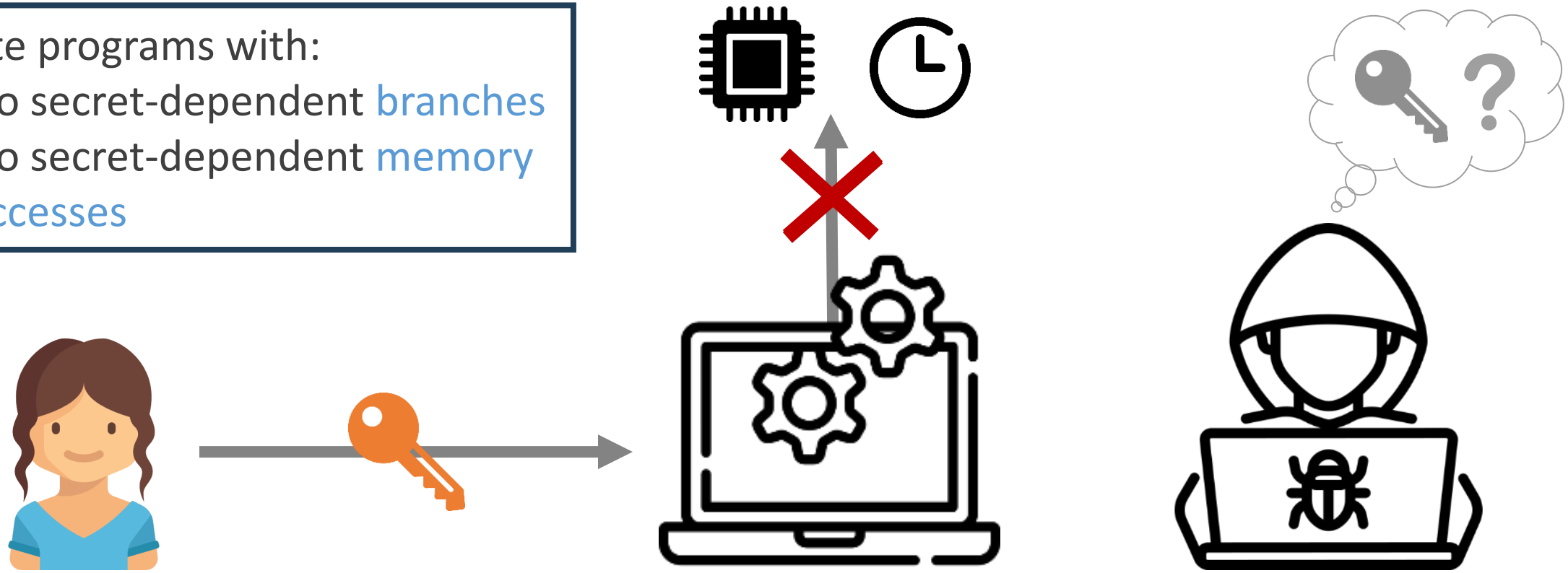
**Abstract.** This paper demonstrates complete AES key recovery from known-plaintext timings of a network server on another computer. This attack should be blamed on the AES design, not on the particular AES library used by the server; it is extremely difficult to write constant-time high-speed AES software for common general-purpose computers. This paper discusses several of the obstacles in detail.

2005

# Solution? Constant-time programming!

Write programs with:

- No secret-dependent branches
- No secret-dependent memory accesses



Already used in many cryptographic implementations

# Constant-time is not easy to implement

```
uint32_t select(uint32_t x, uint32_t y, bool secret) {
  if (secret) return x;
  else return y;
}
```

```
uint32_t ct_select(uint32_t x, uint32_t y, bool secret) {
  signed b = 0 - secret;
  return (x & b) | (y & ~b);
}
```

# Compilers can break constant-time!

```
uint32_t ct_select(uint32_t x, uint32_t y, bool secret) {
  signed b = 0 - secret;
  return (x & b) | (y & ~b);
}
```

clang-3.0 –O0

```
public ct_select_u32_v4
ct_select_u32_v4 proc near

var_14= dword ptr -14h
var_D= byte ptr -0Dh
var_C= dword ptr -0Ch
var_8= dword ptr -8
arg_0= dword ptr  4
arg_4= dword ptr  8
arg_8= byte ptr  0Ch

push    esi
sub     esp, 10h
mov     al, [esp+14h+arg_8]
mov     ecx, [esp+14h+arg_4]
mov     edx, [esp+14h+arg_0]
mov     [esp+14h+var_8], edx
mov     [esp+14h+var_C], ecx
and     al, 1
mov     [esp+14h+var_D], al
mov     al, [esp+14h+var_D]
and     al, 1
movzx   ecx, al
mov     edx, 0
sub     edx, ecx
mov     [esp+14h+var_14], edx
mov     ecx, [esp+14h+var_8]
and     ecx, [esp+14h+var_14]
mov     edx, [esp+14h+var_C]
mov     esi, [esp+14h+var_14]
xor     esi, 0FFFFFFFFh
and     esi, edx
or      esi, ecx
mov     eax, esi
add     esp, 10h
pop     esi
retn
ct_select_u32_v4 endp
```

clang-3.0 –O3

```
public ct_select_u32_v4
ct_select_u32_v4 proc near

arg_0= byte ptr  4
arg_4= byte ptr  8
arg_8= byte ptr  0Ch

mov     al, [esp+arg_8]
test    al, al
jz      short loc_804842F
```

```
lea     eax, [esp+arg_0]
mov     eax, [eax]
retn
```

```
loc_804842F:
lea     eax, [esp+arg_4]
mov     eax, [eax]
retn
ct_select_u32_v4 endp
```

# Compilers can break constant-time!

```
uint32_t ct_select(uint32_t x, uint32_t y, bool secret) {
  signed b = 0 - secret;
  return (x & b) | (y & ~b);
}
```

clang-3.0 –O0

```
public ct_select_u32_v4
ct_select_u32_v4 proc near

var_14= dword ptr -14h
var_D= byte ptr -0Dh
var_C= dword ptr -0Ch
var_8= dword ptr -8
arg_0= dword ptr  4
arg_4= dword ptr  8
arg_8= byte ptr  0Ch

push    esi
sub     esp, 10h
mov     al, [esp+14h+arg_8]
mov     ecx, [esp+14h+arg_4]
mov     edx, [esp+14h+arg_0]
mov     [esp+14h+var_8], edx
mov     [esp+14h+var_C], ecx
and     al, 1
mov     [esp+14h+var_D], al
mov     al, [esp+14h+var_D]
and     al, 1
movzx   ecx, al
mov     edx, 0
sub     edx, ecx
mov     [esp+14h+var_14], edx
mov     ecx, [esp+14h+var_8]
and     ecx, [esp+14h+var_14]
mov     edx, [esp+14h+var_C]
mov     esi, [esp+14h+var_14]
xor     esi, 0FFFFFFFFh
and     esi, edx
or      esi, ecx
mov     eax, esi
add     esp, 10h
pop     esi
retn
ct_select_u32_v4 endp
```

clang-3.0 –O3

```
public ct_select_u32_v4
ct_select_u32_v4 proc near

arg_0= byte ptr  4
arg_4= byte ptr  8
arg_8= byte ptr  0Ch

mov     al, [esp+arg_8]
test    al, al
jz      short loc_804842F
```

```
lea     eax, [esp+arg_0]
mov     eax, [eax]
retn
```

```
loc_804842F:
lea     eax, [esp+arg_4]
mov     eax, [eax]
retn
ct_select_u32_v4 endp
```

Need to reason about CT at low-level (assembly)!

# How to formally define constant-time?

# Constant-time programming, formally?

Side-channel observations produced by program executions must be independent from secret input

# Constant-time programming, formally?

Side-channel observations produced by program executions must be independent from secret input

How do we formalize program executions?

# System model

**Small asm language**

$$\text{(Values) } v \in \mathbb{V} \qquad \text{(Registers) } \mathtt{x} \in \mathbb{R} \qquad \text{(Labels) } \ell \in \mathbb{L}$$

$$\begin{aligned}
\langle exp \rangle &::= v \mid \mathtt{x} \\
\langle inst \rangle &::= \mathtt{add}\ \mathtt{x}\ \langle exp \rangle\ \langle exp \rangle \mid \mathtt{mul}\ \mathtt{x}\ \langle exp \rangle\ \langle exp \rangle \\
&\quad \mid \mathtt{load}\ \mathtt{x}\ \langle exp \rangle \mid \mathtt{store}\ \langle exp \rangle\ \langle exp \rangle \\
&\quad \mid \mathtt{beqz}\ \langle exp \rangle\ \ell \mid \mathtt{jmp}\ \langle exp \rangle \\
\text{(Program)} \quad P &: \mathbb{L} \to \langle inst \rangle
\end{aligned}$$

**Configurations**

$$\sigma = \langle r, m \rangle \text{ where } \begin{cases} r : \mathbb{R} \to \mathbb{V} & \text{(Register map)} \\ m : \mathbb{V} \to \mathbb{V} & \text{(Memory)} \end{cases}$$

# System model

**Expression evaluation** $[\![\mathrm{e}]\!]_r = v$

**Instruction evaluation** $\sigma \rightarrow \sigma'$

$$\text{ADD}\quad \frac{\ell = r(\texttt{pc}) \qquad P[\ell] = \texttt{add x } e_1\ e_2 \qquad v = [\![e_1]\!]_r + [\![e_2]\!]_r \qquad r' = r[\texttt{x} \mapsto v][\texttt{pc} \mapsto \ell + 1]}{\langle m, r\rangle \rightarrow \langle m, r'\rangle}$$

# Constant-time programming, formally?

Side-channel observations produced by program executions must be independent from secret input



How do we define side-channel observations?

# Define side-channel observations

**Semantics instrumented with observations**

$$\sigma \xrightarrow{o} \sigma' \text{ with } o \in \mathcal{O} \text{ (Set of observations)}$$

**Constant-time observation mode** (or leakage model)

- Program counter is observable
- Memory addresses are observable

$$\mathcal{O} = \{\bullet, load\ a, store\ a, pc\ \ell\}$$

*Other observation modes are possible*

# Define side-channel observations

*Additions leak an atomic leakage*

$$\text{ADD}\quad \frac{\ell = r(\texttt{pc}) \qquad P[\ell] = \texttt{add x } e_1\ e_2 \qquad v = [\![e_1]\!]_r + [\![e_2]\!]_r \qquad r' = r[\texttt{x} \mapsto v][\texttt{pc} \mapsto \ell + 1]}{\langle m, r\rangle \xrightarrow{\bullet} \langle m, r'\rangle}$$

*Loads leak their address*

$$\text{LOAD}\quad \frac{\ell = r(\texttt{pc}) \qquad P[\ell] = \texttt{load x } e \qquad a = [\![e]\!]_r \qquad r' = r[\texttt{x} \mapsto m(a)][\texttt{pc} \mapsto \ell + 1]}{\langle m, r\rangle \xrightarrow{\textit{load } a} \langle m, r'\rangle}$$

# Define side-channel observations

*Control-flow instruction leak their target*

BEQZ-TAKEN

$$\frac{P[r(\mathtt{pc})] = \mathtt{beqz}\ e\ \ell \qquad [\![e]\!]_r = 0 \qquad r' = r[\mathtt{pc} \mapsto \ell]}{\langle m, r\rangle \xrightarrow{pc\ \ell} \langle m, r'\rangle}$$

BEQZ-NONTAKEN

$$\frac{P[r(\mathtt{pc})] = \mathtt{beqz}\ e\ \ell \qquad [\![e]\!]_r \neq 0 \qquad \ell' = incr(\mathtt{pc}) \qquad r' = r[\mathtt{pc} \mapsto \ell']}{\langle m, r\rangle \xrightarrow{pc\ \ell'} \langle m, r'\rangle}$$

# Constant-time programming, formally?

Side-channel observations produced by program executions must be independent from secret input

What does it mean to be independent from secret input?

# Define security

**Define public/secrets**

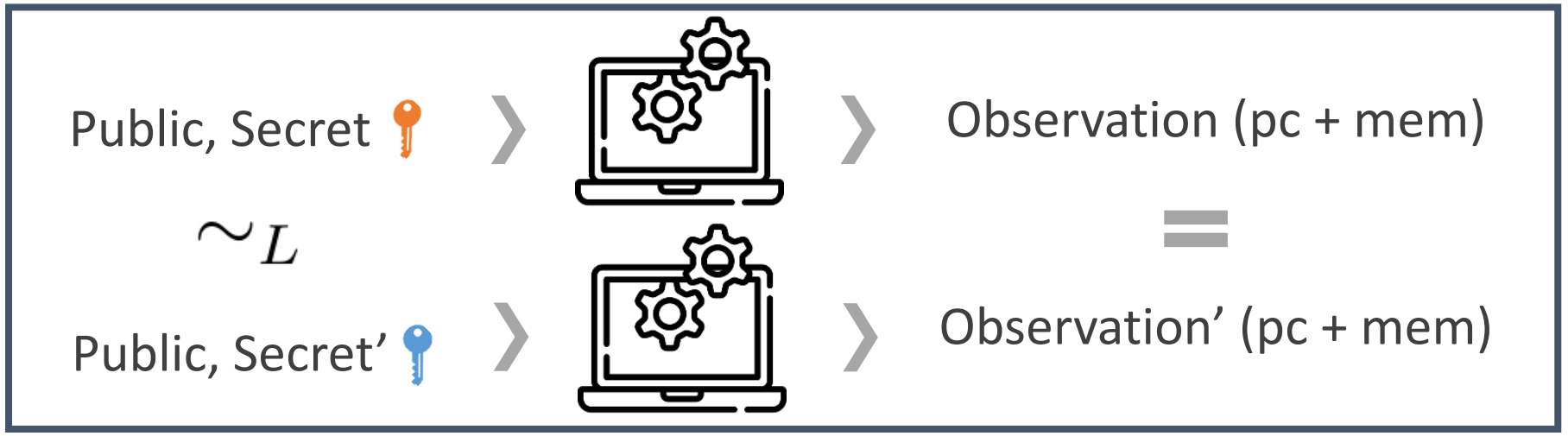
Partition state into *public (low)* / *secret (high)* registers and memory

**Low-equivalence relation** $\sigma \sim_L \sigma'$

Two configurations are *low-equivalent*
if they have the same public values

# Define security

**Definition 1** (Security). *A program $P$ is secure, if and only if:*
*For any pair of initial configurations $\sigma_0$, $\sigma'_0$,*
*if $\sigma_0 \sim_L \sigma'_0$ and $\sigma_0 \xrightarrow{o}{}^{n} \sigma_n$*
*then*
$\sigma'_0 \xrightarrow{o'}{}^{n} \sigma'_n$ *and* $o = o'$



# Define security

**Definition 1** (Security). *A program $P$ is secure, if and only if:*
*For any pair of initial configurations $\sigma_0$, $\sigma'_0$,*
*if $\sigma_0 \sim_L \sigma'_0$ and $\sigma_0 \xrightarrow{o}{}^{n} \sigma_n$*
*then*
$\sigma'_0 \xrightarrow{o'}{}^{n} \sigma'_n$ *and* $o = o'$

*Property relating 2 execution traces (2-hypersafety)*

# Now how do we verify CT?

# Several approaches

**A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries**


Antoine Geimer
Univ. Lille, CNRS, Inria
Univ. Rennes, CNRS, IRISA
Lille, France

Mathéo Vergnolle
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Frédéric Recoules
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Lesly-Ann Daniel
KU Leuven, imec-DistriNet
Leuven, Belgium

Sébastien Bardin
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Clémentine Maurice
Univ. Lille, CNRS, Inria
Lille, France


## Static

- Type systems
- Abstract interpretation
- Symbolic execution

## Dynamic

- Record and compare observations
- Statistical tests
- Fuzzing
- Dynamic symbolic execution

# Several approaches

**A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries**


Antoine Geimer
Univ. Lille, CNRS, Inria
Univ. Rennes, CNRS, IRISA
Lille, France

Mathéo Vergnolle
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Frédéric Recoules
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Lesly-Ann Daniel
KU Leuven, imec-DistriNet
Leuven, Belgium

Sébastien Bardin
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Clémentine Maurice
Univ. Lille, CNRS, Inria
Lille, France


## Static

- Type systems
- Abstract interpretation
- Symbolic execution

## Dynamic

- Record and compare observations
- Statistical tests
- Fuzzing
- Dynamic symbolic execution

# Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976
[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

# Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic store**

$\mathtt{p} \mapsto p$

$\mathtt{s} \mapsto s$

Can error be reached?


[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976

[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

# Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic store**

$\mathtt{p} \mapsto p$

$\mathtt{s} \mapsto s$

$\mathtt{c} \mapsto p \times s - 48$

Can error be reached?


[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976

[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

# Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

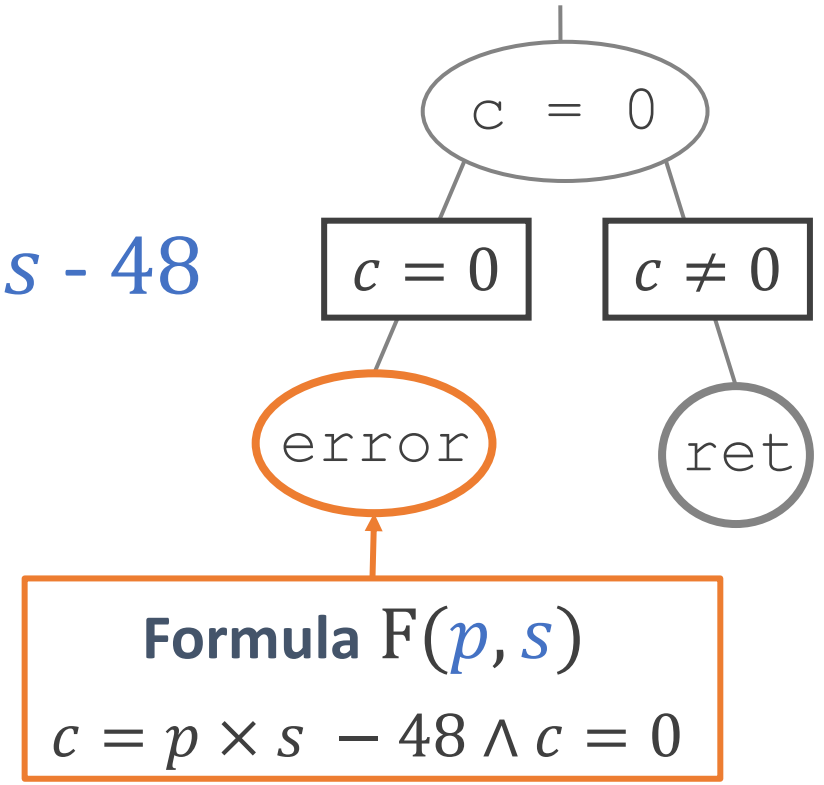
**Symbolic store**

$\mathrm{p} \mapsto p$

$\mathrm{s} \mapsto s$

$\mathrm{c} \mapsto p \times s - 48$

**Path predicate**





[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976

[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

# Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

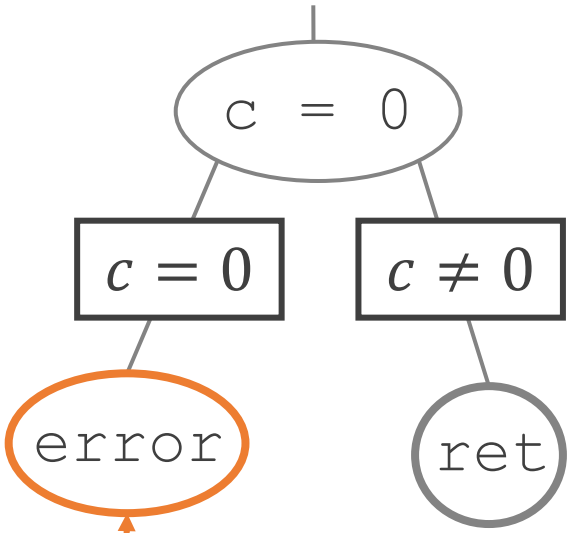
**Symbolic store**

$p \mapsto p$

$s \mapsto s$

$c \mapsto p \times s - 48$

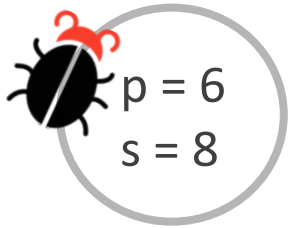
**Path predicate**




**Formula** $F(p, s)$

$c = p \times s - 48 \wedge c = 0$

**SMT-Solver**





[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976

[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

# CT is a 2-hypersafety!

**Definition 1** (Security). *A program $P$ is secure, if and only if:*
*For any pair of initial configurations $\sigma_0$, $\sigma'_0$,*
*if $\sigma_0 \sim_L \sigma'_0$ and $\sigma_0 \xrightarrow{o}{}^{n} \sigma_n$*
*then*
$\sigma'_0 \xrightarrow{o'}{}^{n} \sigma'_n$ *and* $o = o'$

*Property relating 2 execution traces (2-hypersafety)*

*Verification techniques/tools for safety do not apply*

Secure Information Flow by Self-Composition*

Gilles Barthe[1] Pedro R. D'Argenio[2]
Tamara Rezk (corresponding author) [3]

**Key idea:** Turn a 2-hypersafety property of a program **P** to a safety property of a self-composed program **P;P'**

*Can re-use verification techniques/tools for safety!*

# SE for constant-time via self-composition

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can c = 0 depend on s?

# SE for constant-time via self-composition

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic Execution**

**Formula** $\mathrm{F}(p, s)$

$$c = p \times s - 48$$

# SE for constant-time via self-composition

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic Execution**

**Formula** $\mathrm{F}(p, s)$

$$c = p \times s - 48$$

**Self-composition**

**Self-composed formula**

$\mathrm{F}(p, s, p', s')$

$$c = p \times s - 48 \wedge$$

$$c' = p' \times s' - 48$$

*Models 2 executions*

# SE for constant-time via self-composition

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic Execution**

**Formula** $\mathrm{F}(p, s)$

$c = p \times s - 48$

**Self-composition**

**Self-composed formula**

$\mathrm{F}(p, s, p', s')$

$p = p' \wedge$ $c = p \times s - 48 \wedge$ $c' = p' \times s' - 48$ $\wedge\, c = 0 \neq c' = 0$

*= public*

*Models 2 executions*

*Can branch differ?*

# SE for constant-time via self-composition

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic Execution**

**Formula** $\mathrm{F}(p, s)$

$$c = p \times s - 48$$

$$\mathrm{F}(p, s, p', s')$$

$$p = p' \wedge \begin{matrix} c = p \times s - 48 \\ c' = p' \times s' - 48 \end{matrix} \wedge \mathrm{c} = 0 \neq c' = 0$$

**SMT-Solver**

p = 6, s = 8
p' = 6, s'=1

# Beyond self-composition: Optimization for SE

**Limitations:**

- Whole formula is duplicated $\mathrm{F}(p, s, p', s')$
- High number of queries to the solver

Many techniques to optimize self-composed programs…

*Parallel SC, Product programs, Lazy SC, etc.*

# Beyond self-composition: Optimization for SE

Relational Symbolic Execution

Gian Pietro Farina*[1], Stephen Chong†[2] and Marco Gaboardi‡[1]

[1]University at Buffalo, SUNY
[2]Harvard University

- 2 execution in 1 SE instance
- Maximize sharing
- Spare queries

BINSEC/REL: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level

Lesly-Ann Daniel*, Sébastien Bardin*, Tamara Rezk†

* CEA, List, Université Paris-Saclay, France
† INRIA Sophia-Antipolis, INDES Project, France

lesly-ann.daniel@cea.fr, sebastien.bardin@cea.fr, tamara.rezk@inria.fr

- RelSE for CT
- Optimization for binary-level

# Formalization and theorems

**Theorem: RelSE Correct for Bug-Finding**

CT-query is satisfiable at step n-1 in RelSE

$$\Longrightarrow \exists\ \sigma_0 \sim_L \sigma_0' \ \wedge \begin{array}{l} \sigma_0 \xrightarrow{o}{}^{n} \sigma_n \\ \sigma_0' \xrightarrow{o'}{}^{n} \sigma_n' \end{array} \wedge\ o \neq o'$$

**Theorem: Correct for Bounded-Verification**

No CT-query is satisfiable for all n paths in RelSE

$$\Longrightarrow \forall\ \sigma_0 \sim_L \sigma_0' \ \wedge \begin{array}{l} \sigma_0 \xrightarrow{o}{}^{n} \sigma_n \\ \sigma_0' \xrightarrow{o'}{}^{n} \sigma_n' \end{array} \Longrightarrow o = o'$$

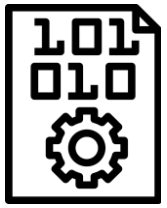
# And concretely?

# BINSEC/REL: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level

Lesly-Ann Daniel*, Sébastien Bardin*, Tamara Rezk†

**Binary**

X86-32 / 64
RISC-V 32
ARMv7/AARCH64/AMD64



**Configuration**

Concretize esp, .data, canaries, …
Libc stubs

IR

**Analysis**

SE/RelSE
Backward-bounded SE
Concrete interpretation

**Helpers**

Loader for ELF/PE
Build & simplify formulas
[…]

**SMT-Solver**

Boolector
Bitwuzla
z3, cvc4, yices

Binsec/Rel https://binsec.github.io/

*CT-analysis of cryptographic primitives*

# Preservation of constant-time by compilers

**11 compiler versions**

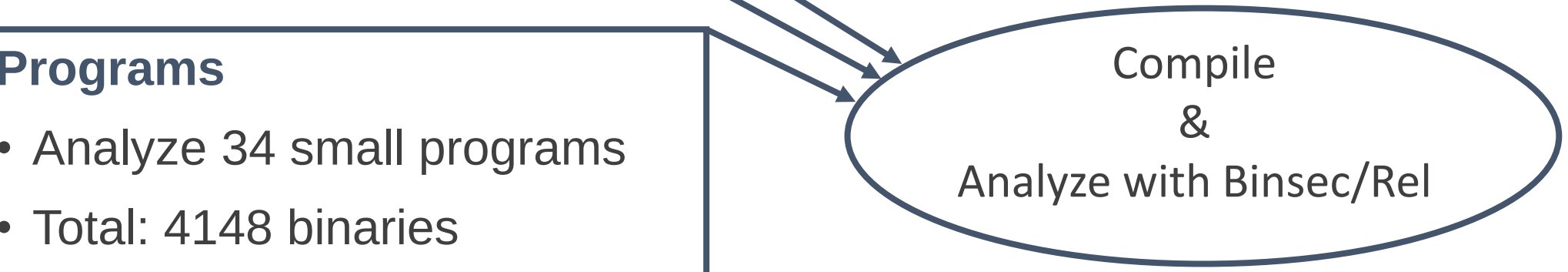
- 5 versions of clang for x86
- 5 versions of gcc for x86
- 1 version of gcc for ARM

**Optimization setups**

- Optimization level O1 … O3
- Individual optimizations
  - X86-cmov-converter, if-conversion

**Programs**

- Analyze 34 small programs
- Total: 4148 binaries




https://github.com/binsec/rel_bench/tree/main/properties_vs_compilers/ct

Fully reproducible build: Nix virtual env

# LLVM-IR ≠ Binary!

**Source**

```
int sort2(int *out2, int *in2) {
  signed char c;
  c = (in2[0] < in2[1]) - 1;
  out2[0] = (~c & in2[0]) | (c & in2[1]);
  out2[1] = (~c & in2[1]) | (c & in2[0]);
  return (in2[0] < in2[1]);
}
```

**LLVM-IR**

```
; Function Attrs: nounwind
define i32 @sort2(i32* nocapture %out2, i32* nocapture readonly %in2) {
  %1 = load i32* %in2, align 4, !tbaa !1
  %2 = getelementptr inbounds i32* %in2, i32 1
  %3 = load i32* %2, align 4, !tbaa !1
  %not. = icmp sge i32 %1, %3
  %4 = select i1 %not., i32 %3, i32 %1
  store i32 %4, i32* %out2, align 4, !tbaa !1
  %5 = load i32* %2, align 4, !tbaa !1
  %6 = load i32* %in2, align 4, !tbaa !1
  %7 = select i1 %not., i32 %6, i32 %5
  %8 = getelementptr inbounds i32* %out2, i32 1
  store i32 %7, i32* %8, align 4, !tbaa !1
  %9 = load i32* %in2, align 4, !tbaa !1
  %10 = load i32* %2, align 4, !tbaa !1
  %11 = icmp slt i32 %9, %10
  %12 = zext i1 %11 to i32
  ret i32 %12
}
```

**Binary**

```
public sort2
sort2 proc near

arg_0= dword ptr  4
arg_4= dword ptr  8

push    esi
mov     eax, [esp+4+arg_4]
mov     edx, [eax]
mov     esi, [eax+4]
lea     ecx, [eax+4]
cmp     edx, esi
jge     short loc_80483B3

mov     esi, edx

loc_80483B3:
mov     edx, [esp+4+arg_0]
mov     [edx], esi
mov     esi, eax
jge     short loc_80483BF

mov     esi, ecx

loc_80483BF:
mov     ecx, [esi]
mov     [edx+4], ecx
mov     ecx, [eax]
cmp     ecx, [eax+4]
setl    al
movzx   eax, al
pop     esi
retn
sort2 endp
```

Backend passes can still introduce violations!

# Clang adds secret dependent memory access

```
1  void sort2(i32* out, i32* in) {
2    a0 = load in[0]
3    a1 = load in[1]
4    a = select (a0 < a1) a0 a1
5    store a out[0]
6    b1 = load in[1]
7    b0 = load in[0]
8    b = select (a0 < a1) b1 b0
9    store b out[1] }
```

**LLVM-IR**

```
1  sort2:
2          esi := load (in+0)
3          edi := load (in+4)
4          cmp esi edi
5          edi := cmovle esi
6          store (out+0) edi
7          ecx := in+0
8          edx := in+4
9          edx := cmovge ecx
10         ecx := load edx
11         store (out+4) ecx
```

**clang-9 –m32 –O3 –march=i686**

# Recap

- Constant-Time = de facto standard against microarchitectural SCA
- We can formalize CT as a 2-hypersafety

$$\begin{array}{c} \textit{For any pair of initial configurations } \sigma_0, \sigma_0', \\ \textit{if } \sigma_0 \sim_L \sigma_0' \textit{ and } \sigma_0 \xrightarrow{o}{}^{n} \sigma_n \\ \textit{then} \\ \sigma_0' \xrightarrow{o'}{}^{n} \sigma_n' \textit{ and } o = o' \end{array}$$

- There are tools to verify crypto primitives / find bugs


- We can find cool bugs introduced by compilers

*LLVM analysis is not sufficient!*

# PART 2

## Spectre Attacks

*Or why is my code still leaking and what can I do about it?*

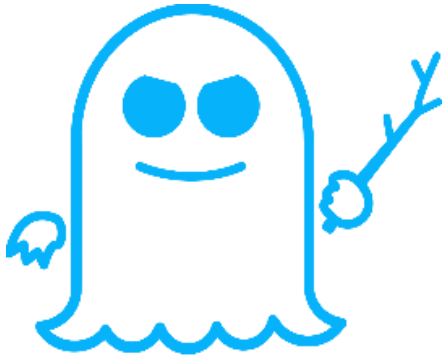
# Spectres are haunting our code

**Spectre Attacks: Exploiting Speculative Execution**

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]

**2018**

- Exploit speculations in (almost all) processors
- Wrong speculation = transient executions
- Transient executions are reverted at architectural level
- But *not the microarchitectural state* (e.g. cache)



***Idea.*** *Force victim to encode secret data in microarchitecture during transient execution & recover them with microarchitectural attacks*

# Constant-time is vulnerable to Spectre

```
   char array[len]
   char mysecret
1: if (idx < len)
2:     x = array[idx]
3:     leak(x)
```



**Is this code secure?**

Leaks x to the microarchitectural state (e.g. load, or branch instr.)

Secure iff `mysecret` does not flow to `leak`

# Constant-time is vulnerable to Spectre

```
   char array[len]
   char mysecret
1: if (idx < len)
2:     x = array[idx]
3:     leak(x)
```

**ISA (sequential) execution**

Conditional bound check ensures `idx` is in bounds

x only contains public data

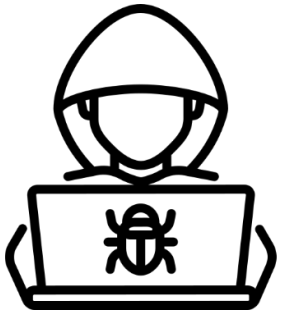
# Constant-time is vulnerable to Spectre

```
   char array[len]
   char mysecret
1: if (idx < len)
2:     x = array[idx]
3:     leak(x)
```

**Actual (speculative) execution**

Branch condition can be (mis)predicted

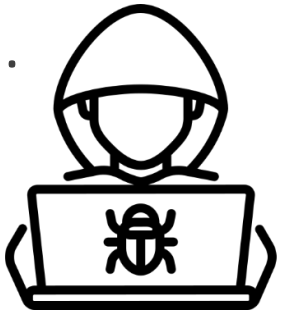
Can I exploit that to `leak(mysecret)` ?



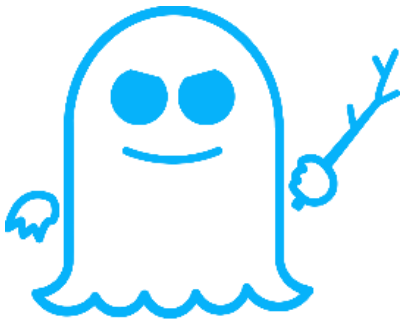
# Constant-time is vulnerable to Spectre

```
   char array[len]
   char mysecret
1: if (idx < len)
2:     x = array[idx]
3:     leak(x)
```

1. **Trains** branch predictor to predict true
2. **Run** victim with `idx = len`
   - Branch is mispredicted to true 
   - Oob access to `mysecret`
   - Transient execution `leak(mysecret)` 
3. **Extract** `mysecret` from microarch.



# Many variants of Spectre



1. Misspeculation leads to transient execution

   *Many sources of speculation*

2. Transient execution leaks secret via side-channel

   *Many side-channel vectors (timing, caches, buffers, etc.)*

Many sources of speculation ⇔ many variants of Spectre [1]

- *Spectre-PHT: conditional branch*
- *Spectre-BTB: indirect branch*
- *Sprectre-RSB: return address*
- *Spectre-STL: memory dependencies*
- *etc. (see [2] for the most recent list)*

[1] Canella, Claudio, et al. "A systematic evaluation of transient execution attacks and defenses." *USENIX Security* (2019)
[2] Randal, Allison. "This is how you lose the transient execution war." *arXiv* (2023).

# Countermeasures?

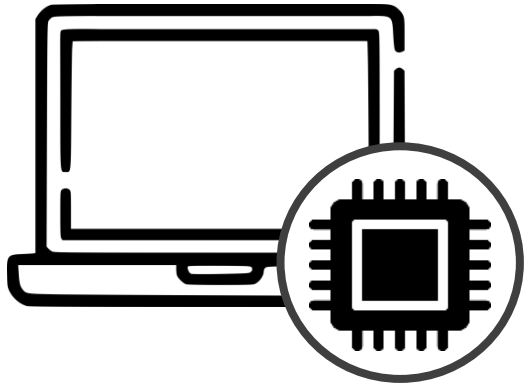
# How to protect against Spectre?

## In Software?


- Speculation barriers (`fence`)
- Load hardening
- Retpolines
- etc.

☺ Full software solution
☹ Variant-specific
☹ Can be costly

## In Hardware?



Microarchitectural partitioning,
Invisible speculation,
OISA, STT, SPT, ConTExT, etc.

☺ Better performance
☺ Comprehensive (but not always)
☹ Adoption is harder

# Fences to block speculative execution

```
   char array[len]
   char mysecret
1: if (idx < len)
2:     x = array[idx]
3:     fence
4:     leak(x)
```

- Branch is mispredicted to true 
- `fence` stalls until branch is resolved
- Rollback before `leak(mysecret)` 

*Transiently execution only until fence*

# Speculative Load Hardening

```
   char array[len]
   char mysecret
1: if (idx < len)
2:     x = array[idx]
3:     x &= (idx < len)
4:     leak(x)
```

- Branch is mispredicted to true 
- Oob access to `mysecret`
- x = 0 if branch is mispredicted
- `leak(0)` 

But where should I insert protections?
Don’t worry kid, formal methods can help you!

# Speculative Constant-Time (SCT)

**Constant-Time Foundations for the New Spectre Era**

Sunjay Cauligi[†] Craig Disselkoen[†] Klaus v. Gleissenthall[†]
Dean Tullsen[†] Deian Stefan[†] Tamara Rezk[⋆] Gilles Barthe[♠♣]

[†]UC San Diego, USA [⋆]INRIA Sophia Antipolis, France
[♠]MPI for Security and Privacy, Germany [♣]IMDEA Software Institute, Spain

**Idea:** Security in the constant-time observation mode on a *speculative semantics*

*Many flavors of microarchitectural semantics / ways to define security (see [1])*

[1] Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., & Stefan, D. (2022, May). SoK: Practical foundations for software Spectre defenses. *SP'22*

# Why is that hard?

**Problem.** Formalize microarchitectural semantics with predictions and out-of-order execution

**Challenge.** Microarchitectural features are complex, often undocumented

**Goals.** Find suitable abstraction to reason about Spectre

- Capture all variants of Spectre
- Keep it simple

# First, how does my microarchitecture work?


- In-order
- Get instruction from memory
- Decode instruction
- Fill reorder buffer (ROB)
- Predict branches

- Out-of-order
- Execute instructions from ROB
- Depends on available operands/execution units
- Rollback incorrect pred.

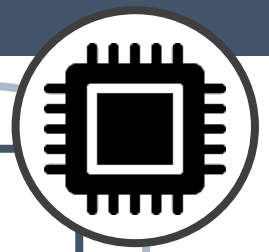
- In-order
- Commits oldest instruction from ROB
- Write result in register file/memory

*Simplified view*

# Out-of-order execution

**Program**

```
1: beqz (i < len) 4
2:    add a a i
3:    load x a
4: […]
```



**Register File** *r*

```
pc ↦ 1
a ↦ 0xf0
i ↦ 0
len ↦ 4
```

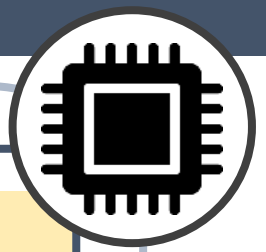
**ROB** *buf*

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:   add a a i
l3:   load x a
l4: […]
```



**Register File** *r*

| | |
|---|---|
| pc | ↦ l1 |
| a | ↦ 0xf0 |
| i | ↦ 0 |
| len | ↦ 4 |

**ROB** *buf*

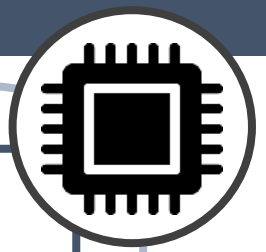
***fetch: false***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l1
a ↦ 0xf0
i ↦ 0
len ↦ 4
```

**ROB** *buf*

```
pc ← l2      @ l1
```

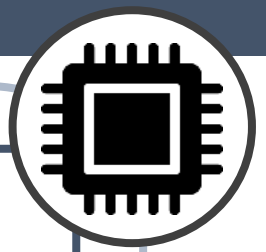
***fetch: false***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l1
a ↦ 0xf0
i ↦ 0
len ↦ 4
```

**ROB** *buf*

```
pc ← l2     @ l1
```

*apl(buf, r)*
=
*r*[pc↦l2]

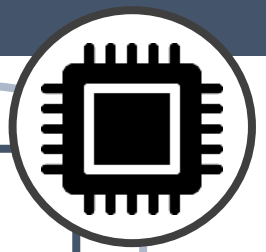
***fetch***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l1
a ↦ 0xf0
i ↦ 0
len ↦ 4
```

**ROB** *buf*

```
pc ← l2      @ l1
add a a i    @ ε
pc ← l3      @ ε
```

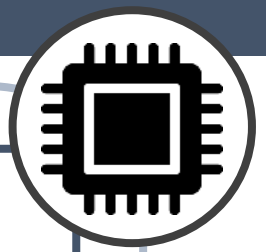
***fetch***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l1
a ↦ 0xf0
i ↦ 0
len ↦ 4
```

**ROB** *buf*

```
pc ← l2      @ l1
add a a i    @ ε
pc ← l3      @ ε
load x a     @ ε
pc ← l4      @ ε
```

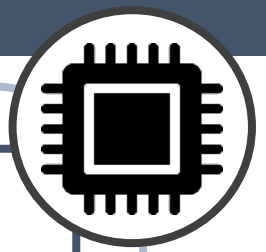
***fetch***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l1
a ↦ 0xf0
i ↦ 0
len ↦ 4
```

**ROB** *buf*

```
pc ← l2      @ l1
add a a i    @ ε
pc ← l3      @ ε
load x a     @ ε
pc ← l4      @ ε
```

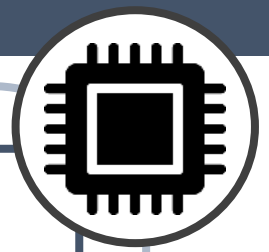
***execute 3***

**Directive**

# Out-of-order execution

**Program**

```
1: beqz (i < len) 4
2:    add a a i
3:    load x a
4: […]
```



**Register File** *r*

```
pc ↦ 1
a ↦ 0xf0
i ↦ 0
len ↦ 4
```

**ROB** *buf*

```
pc ← 2      @ 1
add a a i   @ ε
pc ← 3      @ ε
load x a    @ ε
pc ← 4      @ ε
```

*apl(buf[..2], r)*
=
*r*[pc↦3]
[a ↦ ⊥]

Unresolved dep.
Stuck!

~~*execute 3*~~

**Directive**

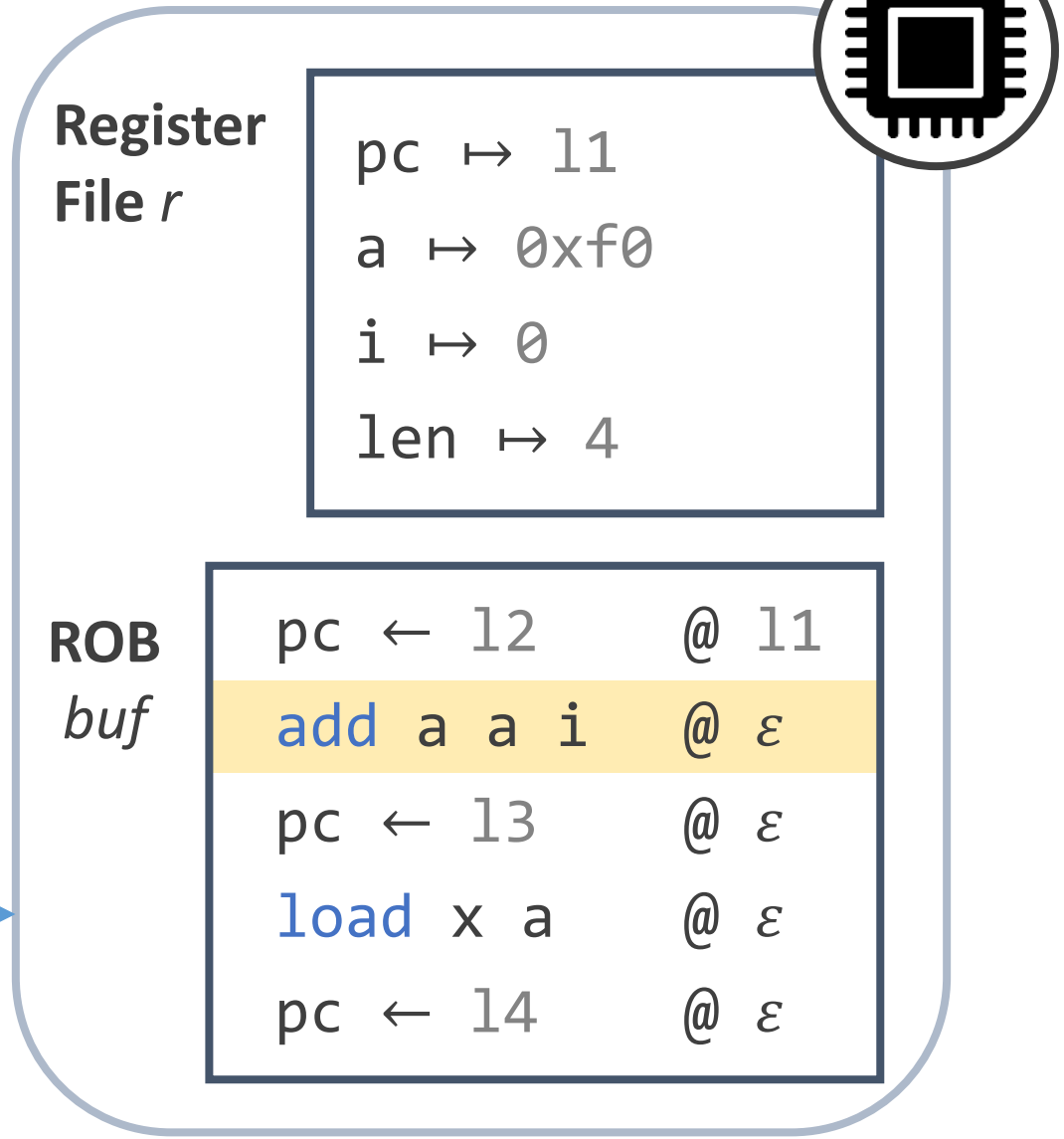
# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```

***execute 1***

**Directive**



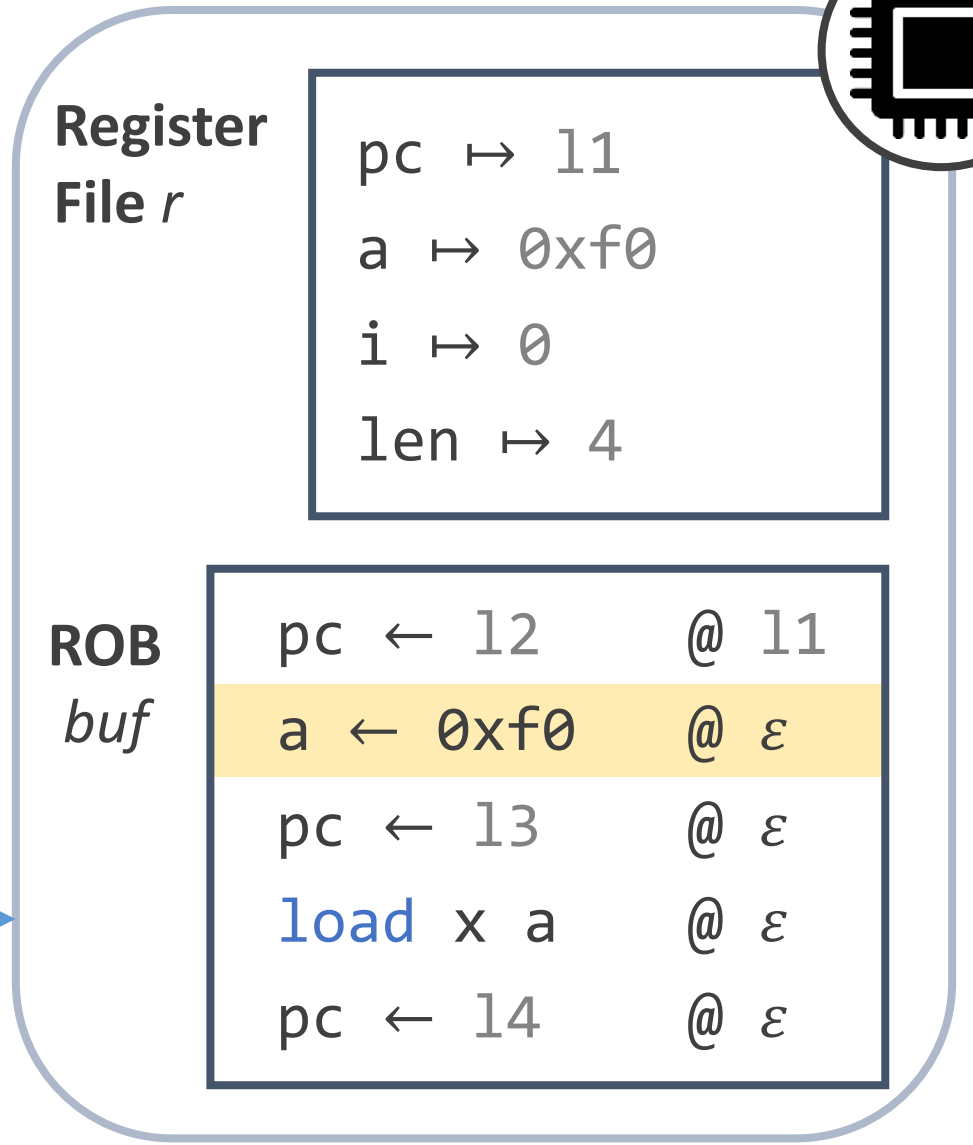
# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```

***execute 1***

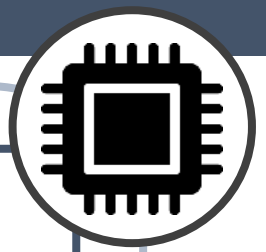
**Directive**



# Reorder Buffer (ROB)

**Program**

```
1: beqz (i < len) 4
2:    add a a i
3:    load x a
4: […]
```



**Register File** *r*

| | |
|---|---|
| pc | ↦ 1 |
| a | ↦ 0xf0 |
| i | ↦ 0 |
| len | ↦ 4 |

**ROB** *buf*

| | |
|---|---|
| pc ← 2 | @ 1 |
| a ← 0xf0 | @ ε |
| pc ← 3 | @ ε |
| load x a | @ ε |
| pc ← 4 | @ ε |

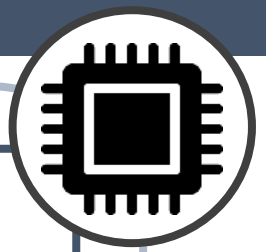
***execute 0***

**Directive**

# Out-of-order execution

**Program**

```
1: beqz (i < len) 4
2:    add a a i
3:    load x a
4: […]
```



**Register File** *r*

| | |
|---|---|
| pc | ↦ 1 |
| a | ↦ 0xf0 |
| i | ↦ 0 |
| len | ↦ 4 |

*Good prediction*
*Commit!*

**ROB** *buf*

| | |
|---|---|
| pc ← 2 | @ ε |
| a ← 0xf0 | @ ε |
| pc ← 3 | @ ε |
| load x a | @ ε |
| pc ← 4 | @ ε |

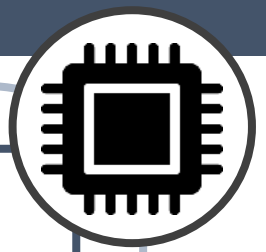
***execute 0***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l1
a ↦ 0xf0
i ↦ 0
len ↦ 0
```

*Bad prediction Rollback!*

**ROB** *buf*

```
pc ← l4      @ ε
```

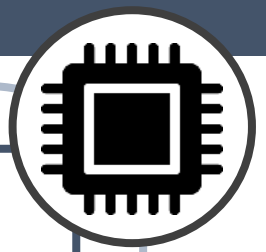
***execute 0***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:   add a a i
l3:   load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l1
a ↦ 0xf0
i ↦ 0
len ↦ 0
```

**ROB** *buf*

```
pc ← l4     @ ε
```

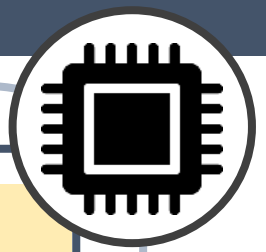
***retire***

**Directive**

# Out-of-order execution

**Program**

```
l1: beqz (i < len) l4
l2:    add a a i
l3:    load x a
l4: […]
```



**Register File** *r*

```
pc ↦ l4
a ↦ 0xf0
i ↦ 0
len ↦ 0
```

**ROB** *buf*

***retire***

**Directive**

# Now, how do we formalize that?

**Small `asm` language**

$$\begin{array}{l}
(\text{Values})\ v \in \mathbb{V} \qquad (\text{Registers})\ \mathtt{x} \in \mathbb{R} \qquad (\text{Labels})\ \ell \in \mathbb{L} \\
\quad \langle exp \rangle ::= v \mid \mathtt{x} \\
\quad \langle inst \rangle ::= \mathtt{add}\ \mathtt{x}\ \langle exp \rangle\ \langle exp \rangle \mid \mathtt{mul}\ \mathtt{x}\ \langle exp \rangle\ \langle exp \rangle \\
\qquad\qquad \mid \mathtt{load}\ \mathtt{x}\ \langle exp \rangle \mid \mathtt{store}\ \langle exp \rangle\ \langle exp \rangle \\
\qquad\qquad \mid \mathtt{beqz}\ \langle exp \rangle\ \ell \mid \mathtt{jmp}\ \langle exp \rangle \mid \mathtt{fence} \\
(\text{Program}) \quad P : \mathbb{L} \to \langle inst \rangle
\end{array}$$

**Configurations**

$$\sigma = \langle r, m, buf \rangle \text{ where } \begin{cases} r : \mathbb{R} \to \mathbb{V} & (\text{Register map}) \\ m : \mathbb{V} \to \mathbb{V} & (\text{Memory}) \\ buf : \langle inst_{rob} \rangle\ list & (\text{Reorder buffer}) \end{cases}$$

# Microarchitectural semantics

**Semantics instrumented with observations and attacker directives**

$$\sigma \xrightarrow[d]{o} \sigma' \text{ with } \begin{cases} o \in \mathcal{O} & \text{(Observation)} \\ d \in \mathcal{D} & \text{(Directive)} \end{cases}$$

**Attacker directives**

- Model attacker ability to influence scheduling / predictions

$$\mathcal{D} = \{fetch, execute\ i, retire\}$$

# Microarchitectural semantics

**Semantics instrumented with observations and attacker directives**

$$\sigma \xrightarrow[d]{o} \sigma' \text{ with } \begin{cases} o \in \mathcal{O} & \text{(Observation)} \\ d \in \mathcal{D} & \text{(Directive)} \end{cases}$$

**Constant-time observation mode** (or leakage model)

- Program counter is observable (also commit and rollback)
- Memory addresses are observable

$$\mathcal{O} = \{\bullet, \mathit{load}\ a, \mathit{store}\ a, \mathit{pc}\ \ell, \mathit{commit}, \mathit{rollback}\}$$

# Example: add instruction

FETCH-ADD

$$\frac{\ell = [\![\texttt{pc}]\!]_{apl(buf,r)} \qquad P[\ell] = \texttt{add x } e_1\ e_2 \qquad buf' = buf \cdot (\texttt{add x } e_1\ e_2@\varepsilon) \cdot (\texttt{pc} \leftarrow \ell + 1@\varepsilon)}{\langle m, r, buf\rangle \xrightarrow[fetch]{} \langle m, r, buf'\rangle}$$

EXECUTE-ADD

$$\frac{|buf| = i \qquad \texttt{fence} \notin buf \qquad inst = \texttt{add x } e_1\ e_2@\varepsilon \qquad r' = apl(buf, r) \qquad v = [\![e_1]\!]_{r'} + [\![e_2]\!]_{r'} \qquad inst' = \texttt{x} \leftarrow v@\varepsilon}{\langle m, r, buf \cdot inst \cdot buf'\rangle \xrightarrow[execute\ i]{\bullet} \langle m, r, buf \cdot inst' \cdot buf'\rangle}$$

RETIRE

$$\frac{inst = \texttt{x} \leftarrow v@\varepsilon \qquad r' = r[\texttt{x} \mapsto v]}{\langle m, r, inst \cdot buf'\rangle \xrightarrow[retire]{} \langle m, r', buf'\rangle}$$

# Example: branches

FETCH-BRANCH-TAKEN

$$\frac{\ell = [\![\mathtt{pc}]\!]_{apl(buf,r)} \qquad P[\ell] = \mathtt{beqz}\ e\ \ell' \qquad buf' = buf \cdot (\mathtt{pc} \leftarrow \ell' @ \ell)}{\langle m, r, buf\rangle \xrightarrow[fetch\ true]{} \langle m, r, buf'\rangle}$$

EXECUTE-COMMIT-BRANCH-TAKEN

$$\frac{|buf| = i \qquad \mathtt{fence} \notin buf \qquad inst = \mathtt{pc} \leftarrow \ell' @ \ell \qquad P[\ell] = \mathtt{beqz}\ e\ \ell' \qquad [\![e]\!]_{apl(buf,r)} = 0 \qquad inst' = \mathtt{pc} \leftarrow \ell' @ \varepsilon}{\langle m, r, buf \cdot inst \cdot buf'\rangle \xrightarrow[execute\ i]{commit \cdot pc\ \ell'} \langle m, r, buf \cdot inst' \cdot buf'\rangle}$$

EXECUTE-ROLLBACK-BRANCH-TAKEN

$$\frac{|buf| = i \qquad \mathtt{fence} \notin buf \qquad inst = \mathtt{pc} \leftarrow \ell' @ \ell \qquad P[\ell] = \mathtt{beqz}\ e\ \ell' \qquad [\![e]\!]_{apl(buf,r)} \neq 0 \qquad inst' = \mathtt{pc} \leftarrow \ell + 1 @ \varepsilon}{\langle m, r, buf \cdot inst \cdot buf'\rangle \xrightarrow[execute\ i]{rollback \cdot pc\ \ell+1} \langle m, r, buf \cdot inst'\rangle}$$

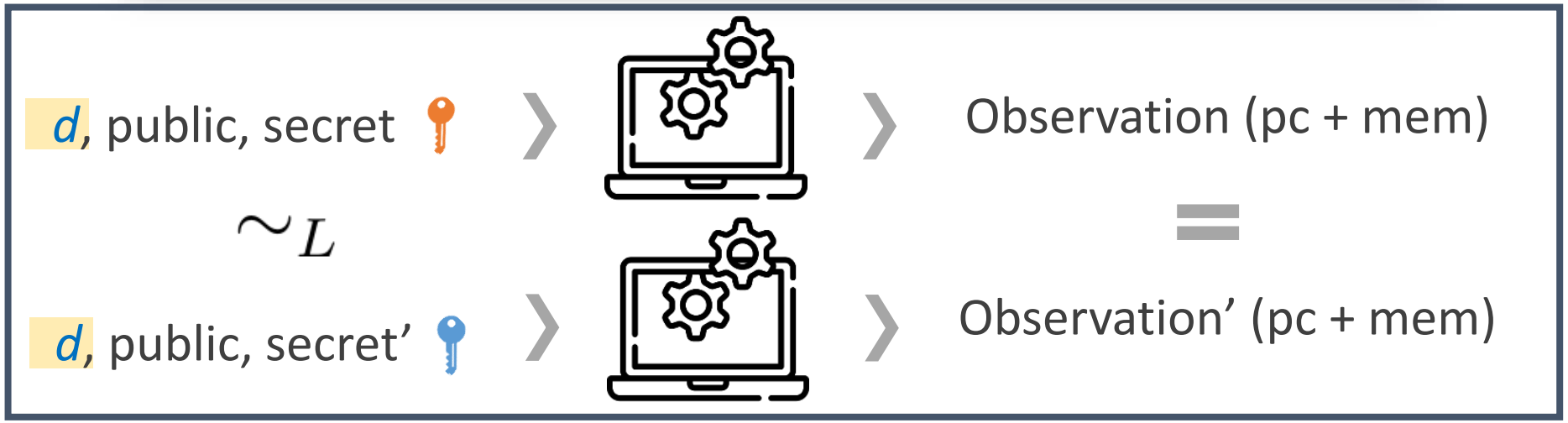
# Define security

**Definition 2** (SCT). *A program $P$ is SCT, if and only if: For any pair of initial configurations $\sigma_0$, $\sigma_0'$, and for any set of directives $d$,*

$$\text{if } \sigma_0 \sim_L \sigma_0' \text{ and } \sigma_0 \xrightarrow[d]{o}{}^{n} \sigma_n$$

$$\text{then}$$

$$\sigma_0' \xrightarrow[d]{o'}{}^{n} \sigma_n' \text{ and } o = o'$$



# Now how do we verify SCT?

# Modelling speculative semantics

**Litmus tests (328 instrutions):**

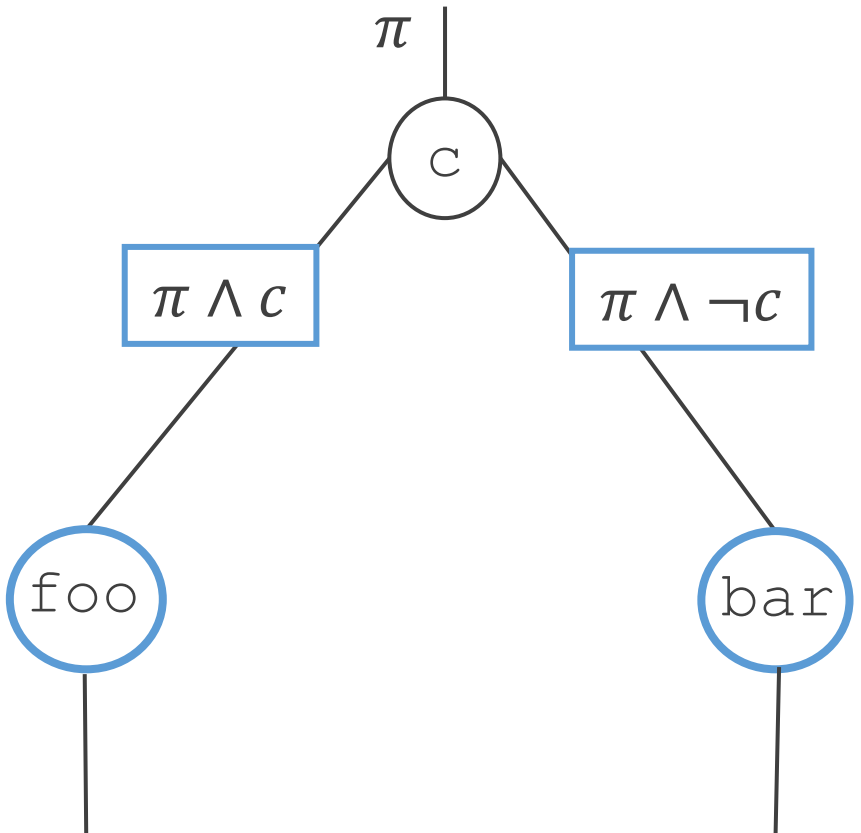
- Sequential semantics
  → **14 paths**

- Speculative semantics
  → **37M paths**


*Modelling all transient paths explicitly is intractable*
*We need to be smarter*

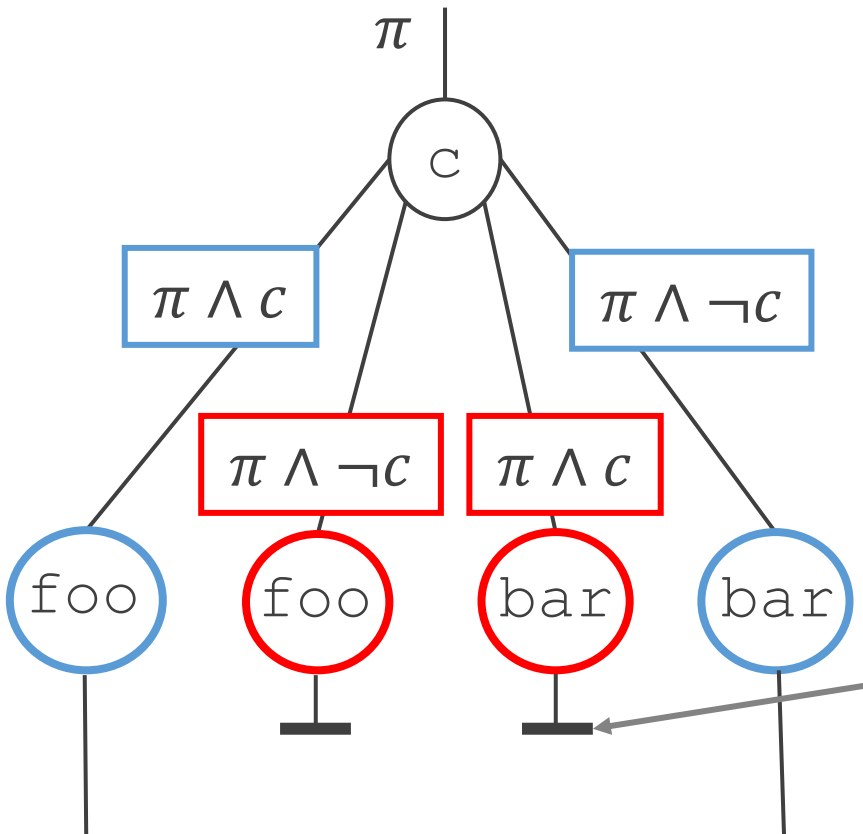
# RelSE for architectural semantics

```
if c
then foo
else bar
```



# RelSE for Spectre-PHT (naive)

```
if c
then foo
else bar
```




Fork into 4 paths:

- *2 sequential paths*
- *+ 2 extra transient path*

On sequential and transient branches:

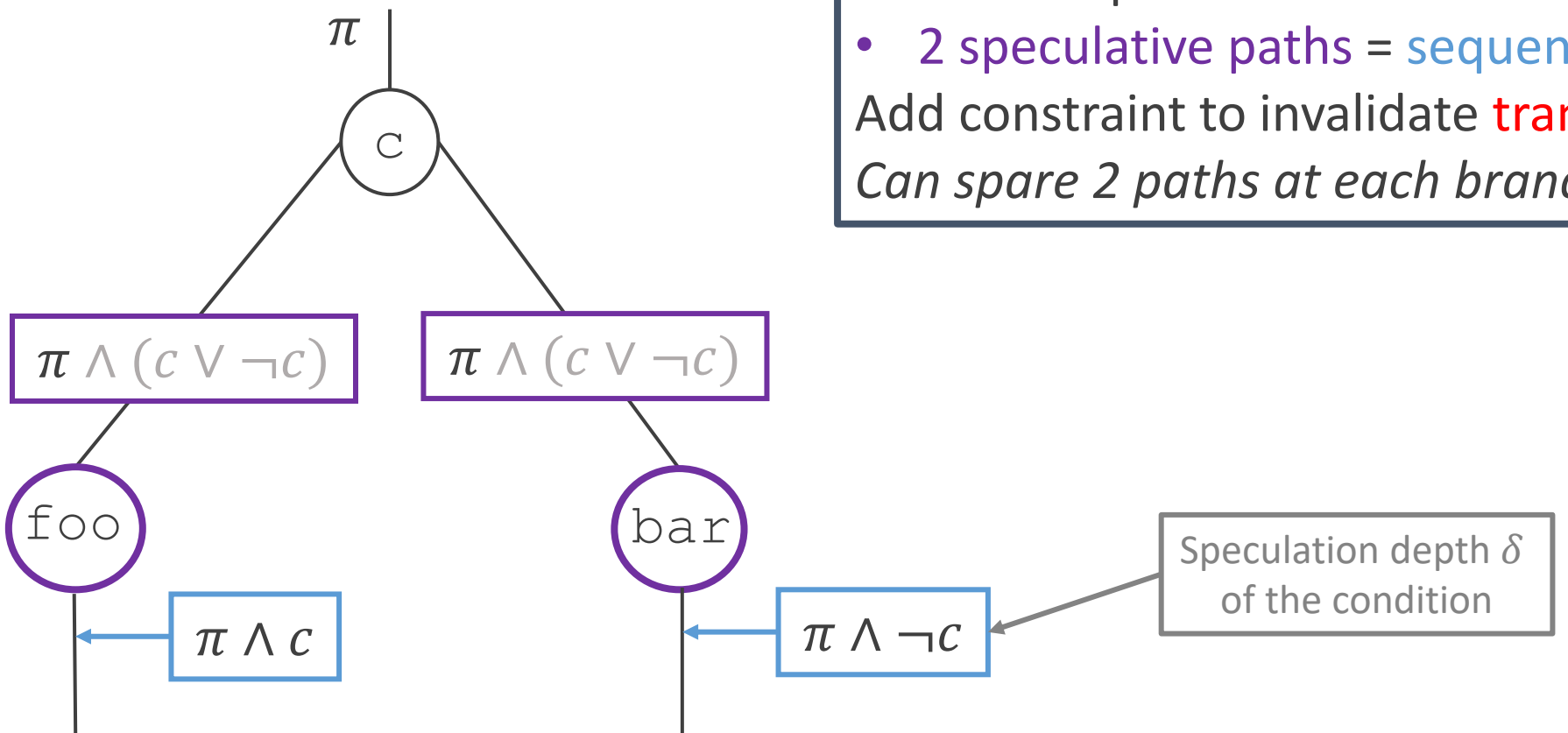
- No secret-dependent branches
- No secret-dependent memory accesses

(e.g. KLEESpectre)

Speculation depth $\delta$ of the condition

# RelSE for Spectre-PHT (but let's be smarter)

```
if c
then foo
else bar
```




Fork into 2 paths:

- 2 speculative paths = sequential ∨ transient

Add constraint to invalidate transient path

*Can spare 2 paths at each branch!*

# And concretely?

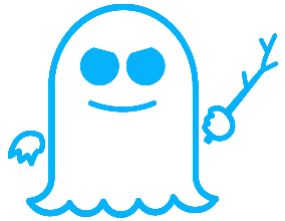
# Verify/optimize Spectre protections

- Find gadgets in crypto [1,2]
- Find attacks combining Spectre variants [2,3]
- Insert Spectre protections smartly [4,5]
- Type system to protect crypto against Spectre [5]
- Find gadgets in the Linux kernel [6]


[1] Cauligi, Sunjay, et al. "Constant-time foundations for the new spectre era." *PLDI'20*
[2] Daniel, Lesly-Ann, Sébastien Bardin, and Tamara Rezk. "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse." *NDSS' 21*
[3] Fabian, Xaver, Marco Guarnieri, and Marco Patrignani. "Automatic Detection of Speculative Execution Combinations." *CCS'22*
[4] Vassena, Marco, et al. "Automatically eliminating speculative leaks from cryptographic code with blade." *POPL'21*
[5] Shivakumar, Basavesh Ammanaghatta, et al. "Typing High-Speed Cryptography against Spectre v1." *SP'23*
[6] Johannesmeyer, Brian, et al. "Kasper: scanning for generalized transient execution gadgets in the linux kernel." *NDSS'22*

# Recap

- Constant-Time is vulnerable against Spectre



- New programming model: SCT

  *Speculative ooo semantics*

$$\sigma \xrightarrow[d]{o} \sigma' \text{ with } \begin{cases} o \in \mathcal{O} & \text{(Observation)} \\ d \in \mathcal{D} & \text{(Directive)} \end{cases}$$

- Harder: need clever tricks to avoid complexity

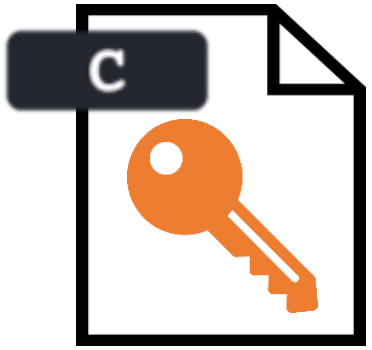

- Yet, formal methods can help optimizing protections and detect bugs!

# PART 3

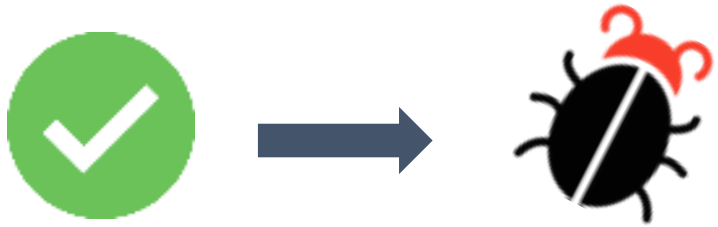
## Fill the gap between models and hardware

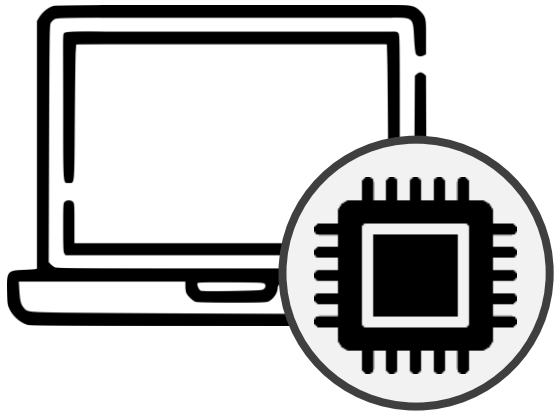

*Or how can we get sound hardware abstractions that can be leveraged by software?*

# First problem: gap model <> HW



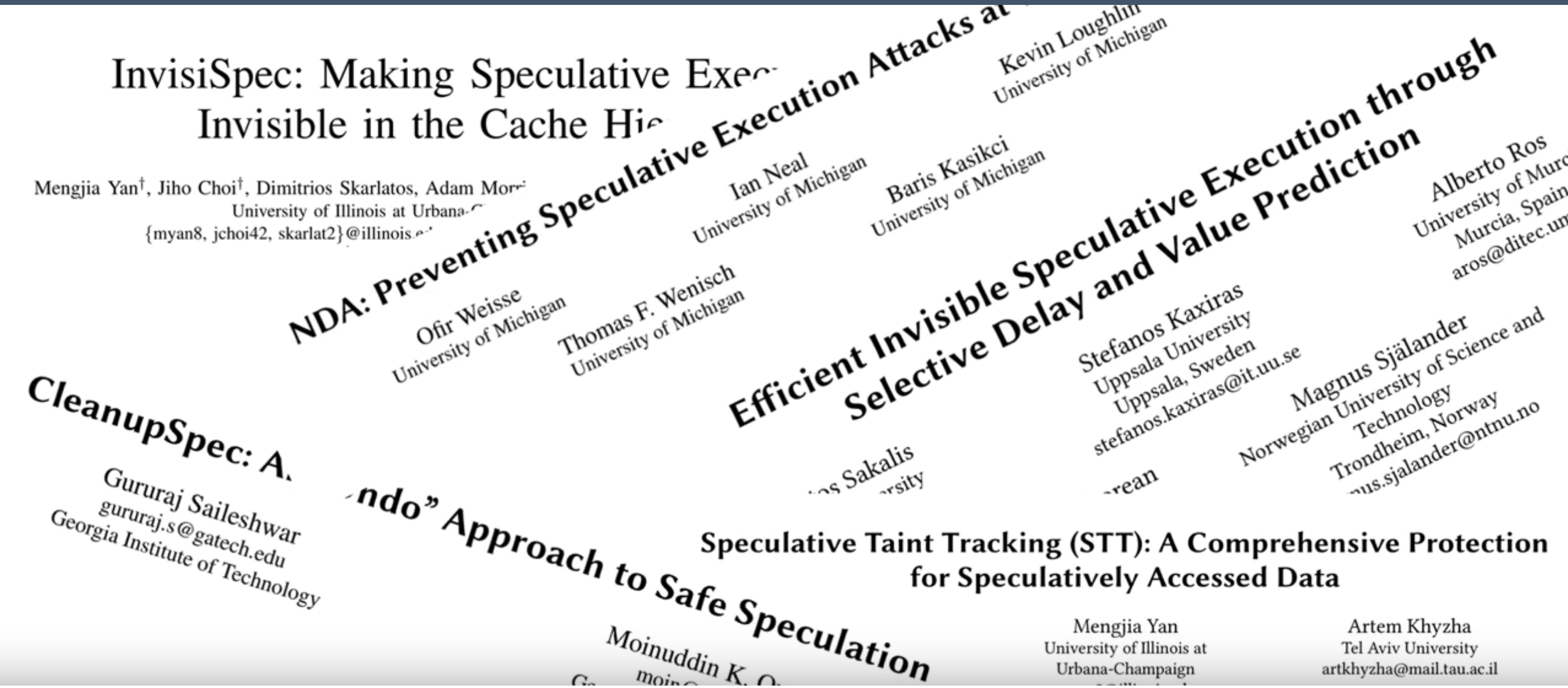

**Software Security Property (e.g. CT/SCT)**



**Actual Microarchitectural Leakage**

# Second problem: many HW defenses

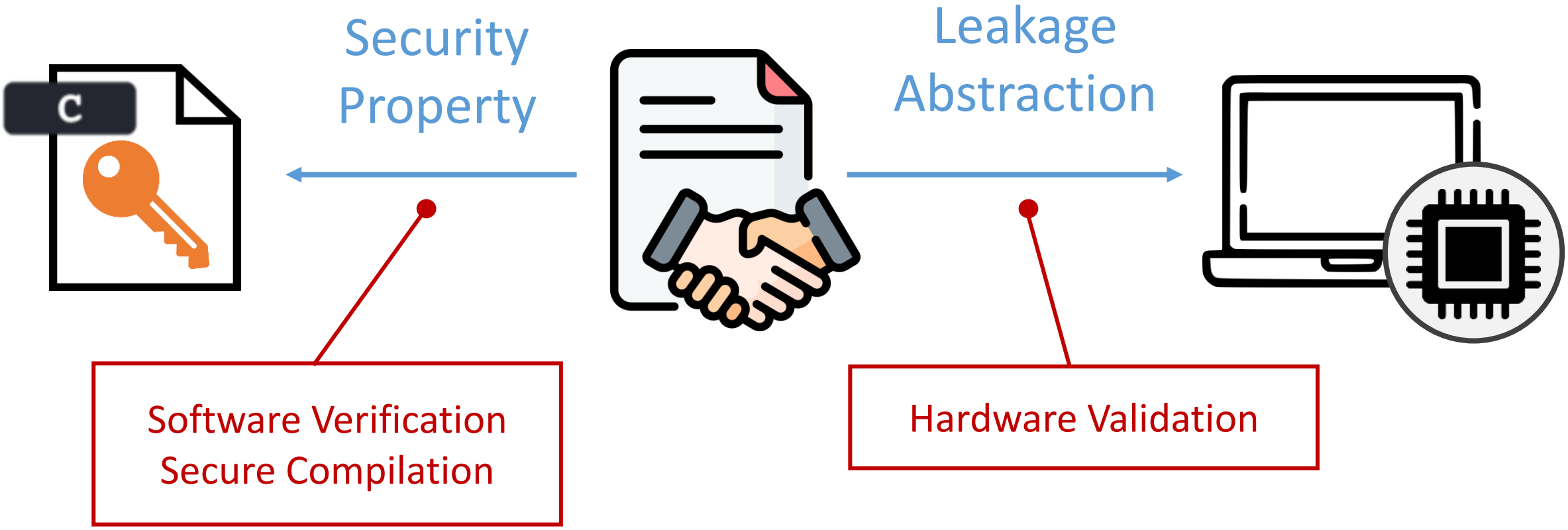


# Second problem: how do we know they work?


- What guarantees?
- How can we program securely?

# Hardware-Software Contracts for Secure Speculation

Marco Guarnieri*, Boris Köpf†, Jan Reineke‡, and Pepe Vila*
*IMDEA Software Institute †Microsoft Research ‡Saarland University



# HW/SW contracts for side-channel-free programs

**Definition.** Contracts specify which program execution a side-channel adversary can distinguish

**Goals.**

- Capture security guarantees of hardware defenses
- Abstracts away hardware details
- Distribute security obligations between software/hardware
- Basis for secure programming

# Contract world

**Contract.** labeled deterministic semantics $\sigma_0 \xrightarrow{l_1} \sigma_1 \xrightarrow{l_2} \dots \xrightarrow{l_n} \sigma_n$

Define a trace of observation produced during execution $[\![p]\!](\sigma_0) = l_1 l_2 \dots l_n$

## Observer mode

- **Constant-time** (ct)
  - Control-flow + memory accesses
- **Architectural observer** (arch)
  - Leaks values of loads

## Execution mode

- **Sequential** (seq)
  - In-order execution
- **Speculative** (spec)
  - Always mispredict branches

# Hardware world

**Hardware states**

$\langle \sigma, \mu \rangle$

**Hardware semantics**

$\langle \sigma, \mu \rangle \Rightarrow \langle \sigma', \mu' \rangle$

**Adversary Model**

Projections of $\mu$

**Hardware observation trace**

$\{\!|p|\!\}(\sigma)$

# Close the gap HW <> contract

**Definition 1** ($\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$). A hardware semantics $\{\!|\cdot|\!\}$ *satisfies a contract* $[\![\cdot]\!]$ if, for all programs $p$ and all initial architectural states $\sigma, \sigma'$, if $[\![p]\!](\sigma) = [\![p]\!](\sigma')$, then $\{\!|p|\!\}(\sigma) = \{\!|p|\!\}(\sigma')$.

*States are indistinguishable in contract semantics*
*Then they should be indistinguishable on HW*

# End-to-end guarantees

**Program noninterference w.r.t to contract**

**Definition 3** ($p \vdash NI(\pi, [\![\cdot]\!])$). Program $p$ is *non-interferent* w.r.t. contract $[\![\cdot]\!]$ and policy $\pi$ if for all initial architectural states $\sigma, \sigma'$: $\sigma \simeq_\pi \sigma' \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma')$.

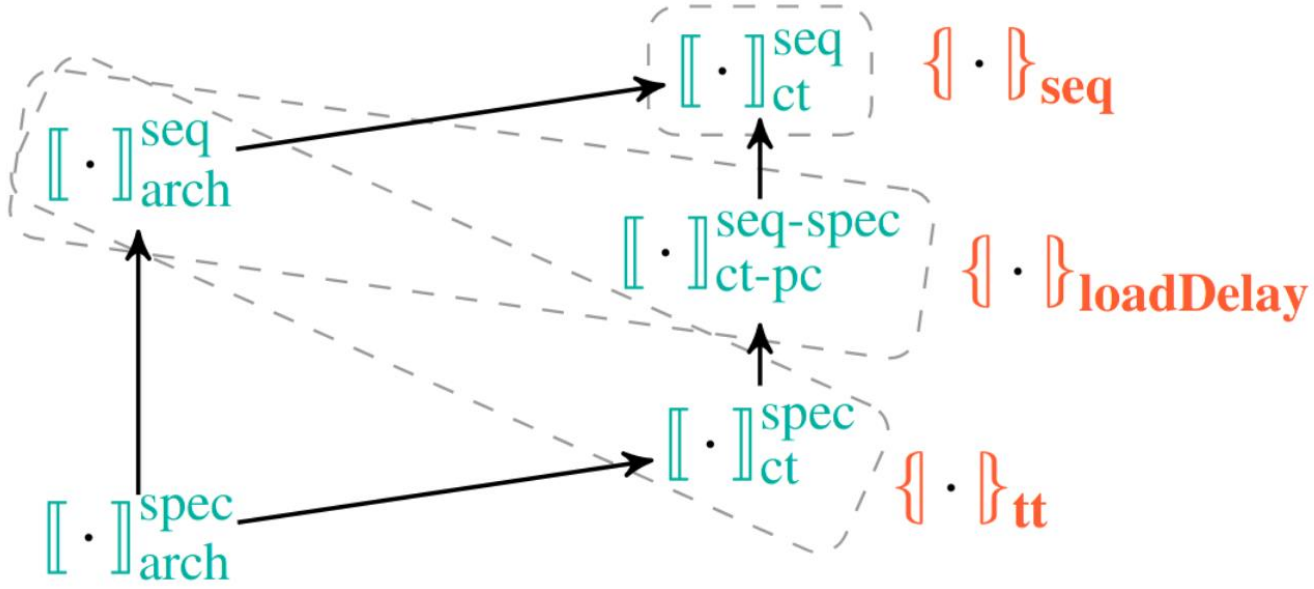
**Proposition 2.** *If* $p \vdash NI(\pi, [\![\cdot]\!])$ *and* $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$, *then* $p \vdash NI(\bar{\pi}, \{\!|\cdot|\!\})$.

*Program security w.r.t. contract gives*
*HW-security on any HW satisfying the contract*

# And concretely?

# Formally study HW countermeasures

- seq: disable all speculation
- loadDelay: delaying all speculative loads
- tt: taint speculative load values and delay computations




*Comparison of hardware countermeasures*

**PROSPECT: Provably Secure Speculation for the Constant-Time Policy**

Lesly-Ann Daniel[1], Marton Bognar[1], Job Noorman[1], Sébastien Bardin[2], Tamara Rezk[3] and Frank Piessens[1]

[1]imec-DistriNet, KU Leuven, 3001 Leuven, Belgium
[2]CEA, List, Université Paris Saclay, France
[3]INRIA, Université Côte d'Azur, Sophia Antipolis, France

- Track and protect secrets during speculative execution
- CT program in ISA semantics ⇒ secure on HW semantics
- Proof based on contract framework

**Revizor: Testing Black-Box CPUs against Speculation Contracts**

Oleksii Oleksenko*
Christof Fetzer
TU Dresden
Dresden, Germany

Boris Köpf
Microsoft Research
Cambridge, UK

**Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing**

Oleksii Oleksenko
*Microsoft Research*

Marco Guarnieri
*IMDEA Software Institute*

Boris Köpf
*Microsoft Research*

Mark Silberstein
*Technion*

- Test CPU against contracts
  - Generate pairs of programs indistinguishable wrt. contract
  - Execute them on CPU, check if they differ
- Rediscover existing Spectre variants
- Discover two new variants
  - Zero-dividend-injection
  - String-comparison overrun (repe, repne)

**Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts**

Zilong Wang
IMDEA Software Institute

Gideon Mohr
Saarland University

Klaus von Gleissenthall
Vrije Universiteit Amsterdam

Jan Reineke
Saarland University

Marco Guarnieri
IMDEA Software Institute

- Verify RTL processor designs against contract (ISA level)
- Applied on 3 RISC-V processors leaking CF, MEM, variable-time instr.
- Small in-order processors, no speculative execution

# Contract-Aware Secure Compilation

Marco Guarnieri
IMDEA Software Institute

Marco Patrignani
Stanford University
CISPA Helmholz Center for Information Security

- Source code shouldn't be tailored to specific HW guarantees
- Contract-Aware Secure COmpilation (CASCO)
  - Compiler parametric wrt. HW/SW contract
  - Make compilers aware of HW security guarantees
  - Leverage these to produce secure code
- *(Still theoretical)*

# Recap

- Gap between model and hardware

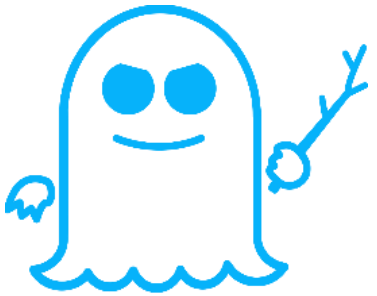


- Hard to reason about HW defenses
- Contract can help formalizing HW leakage and guarantees
- Strong formal basis to reduce the gap!

  *With already strong concrete results*

# Conclusion

- Concrete HW execution leak information
  - HW optimizations do not care for security

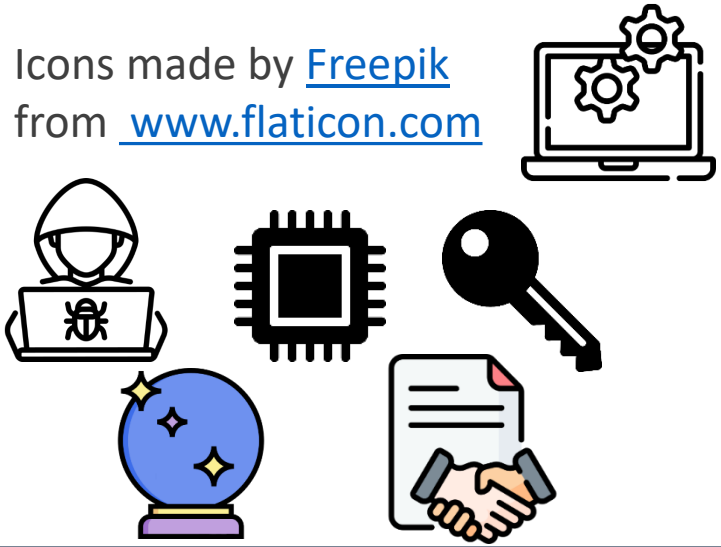


- Formal methods can help
  - Formalize observations & define secure programming models
  - Find bugs / prove that SW is secure
- Still a gap between HW-models
  - HW-SW contracts can help reduce it!
  - Opens exciting research directions!


# Backup

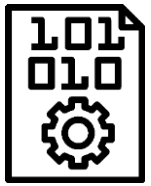
# Credits

Icons made by Freepik from www.flaticon.com



Icons made by bqlqn from www.flaticon.com

Icons made by Becris from www.flaticon.com



Icons made by scrip from www.flaticon.com

From draw.io

# Beyond self-composition: Optimization for symbolic execution

# Relational SE

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic store**

$$p \mapsto < p >$$

$$s \mapsto < s \mid s' >$$

Sharing 👍

# Relational SE

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic store**

Sharing 👍

$$p \mapsto < p >$$

$$s \mapsto < s \mid s' >$$

$$c \mapsto < p \times s - 48 \mid p \times s' - 48 >$$

# Relational SE

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Check CT!

**Symbolic store**

Sharing 👍

$$\mathtt{p} \mapsto < p >$$
$$\mathtt{s} \mapsto < s \mid s' >$$
$$\mathtt{c} \mapsto < p \times s - 48 \mid p \times s' - 48 >$$

# Relational SE

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic store**

$$\begin{aligned} \mathrm{p} &\mapsto < p > \\ \mathrm{s} &\mapsto < s \mid s' > \\ \mathrm{c} &\mapsto < p \times s - 48 \mid p \times s' - 48 > \end{aligned}$$

Sharing 👍

Relational formula: $\mathrm{F}(p, s, s')$

$$\begin{aligned} c &= p \times s - 48 \\ c' &= p \times s' - 48 \end{aligned} \wedge \mathrm{c} = 0 \neq c' = 0$$

Sharing 👍

# Relational SE

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic store**

Sharing

$$\mathrm{p} \mapsto < p >$$
$$\mathrm{s} \mapsto < s \mid s' >$$
$$\mathrm{c} \mapsto < p \times s - 48 \mid p \times s' - 48 >$$

Relational formula: $\mathrm{F}(p, s, s')$

$$c = p \times s - 48$$
$$c' = p \times s' - 48$$
$$\wedge\, \mathrm{c} = 0 \neq c' = 0$$

Sharing

**SMT-Solver**

p = 6
s = 8  s'=1

# Relational SE

```
foo(public p, secret s){
  c := p - 48;
  if(c = 0) error();
  else return s/c;
}
```

**Symbolic store**

$$p \mapsto < p >$$
$$s \mapsto < s \mid s' >$$
$$c \mapsto < p - 48 >$$


[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

# Better approach: Relational SE

```
foo(public p, secret s){
  c := p - 48;
  if(c = 0) error();
  else return s/c;
}
```

Check CT!

**Symbolic store**

$$\mathtt{p} \mapsto < p >$$
$$\mathtt{s} \mapsto < s \mid s' >$$
$$\mathtt{c} \mapsto < p - 48 >$$

[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

# Better approach: Relational SE

```
foo(public p, secret s){
  c := p - 48;
  if(c = 0) error();
  else return s/c;
}
```

Check CT!

**Symbolic store**

$$\mathtt{p} \mapsto < p >$$
$$\mathtt{s} \mapsto < s \mid s' >$$
$$\mathtt{c} \mapsto < p - 48 >$$

Track secrets and spare queries 👍

[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

# Spectre-STL

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

## Sequential execution

```
store a s
store a p
store b q
v = load a
leak(v)
```

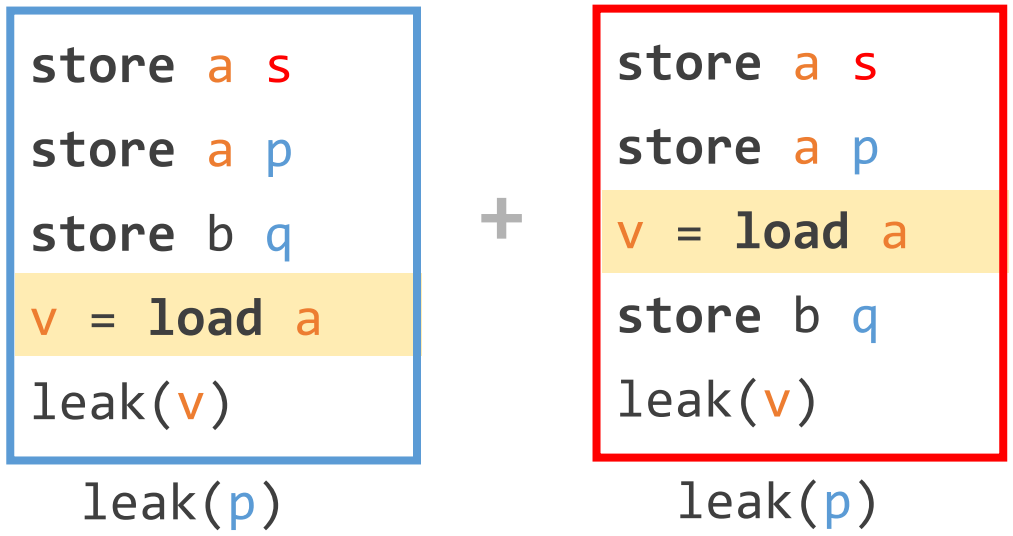
```
leak(p)
```

With s = secret / q and p = public / a ≠ b

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

**Sequential execution** + **Transient Executions**




With s = secret / q and p = public / a ≠ b

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

**Sequential execution** + **Transient Executions**

```
store a s
store a p
store b q
v = load a
leak(v)
```

leak(p)

+

```
store a s
store a p
v = load a
store b q
leak(v)
```

leak(p)

+

```
store a s
v = load a
store a p
store b q
leak(v)
```

leak(s)

With s = secret / q and p = public / a ≠ b

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

**Sequential execution** + **Transient Executions**

```
store a s
store a p
store b q
v = load a
leak(v)
```

leak(p)

+

```
store a s
store a p
v = load a
store b q
leak(v)
```

leak(p)

+

```
store a s
v = load a
store a p
store b q
leak(v)
```

leak(s)

+

```
v = load a
store a s
store a p
store b q
leak(v)
```

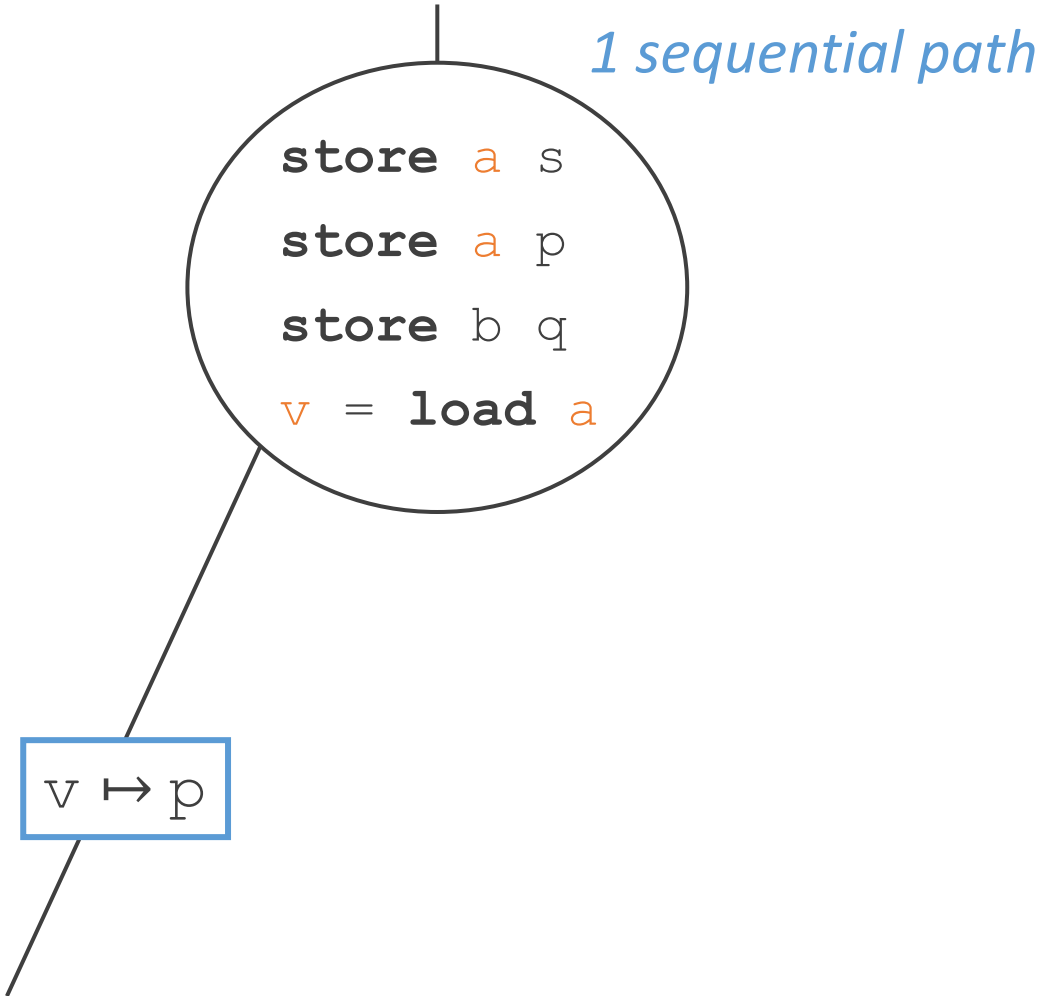
leak(init_mem[a])

With s = secret / q and p = public / a ≠ b

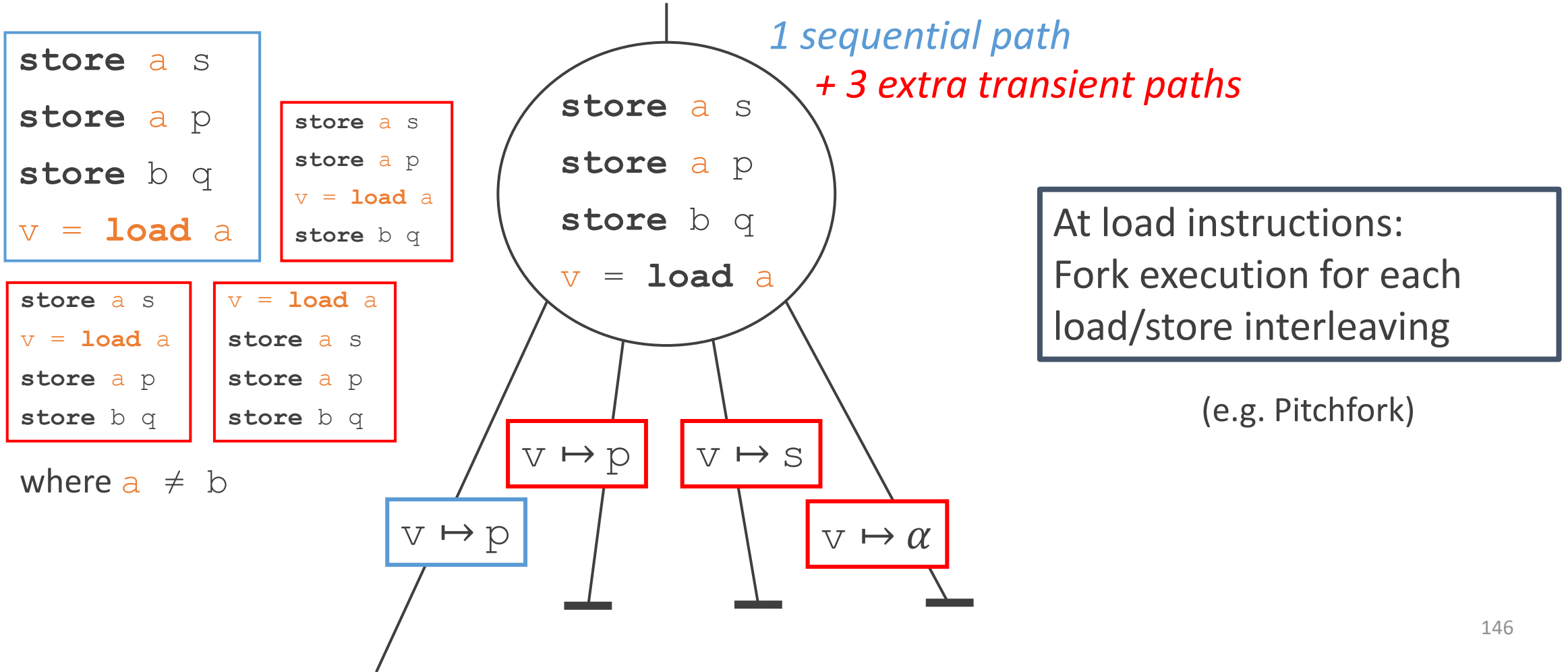
# RelSE for architectural semantics

```
store a s
store a p
store b q
v = load a
```

where $a \neq b$



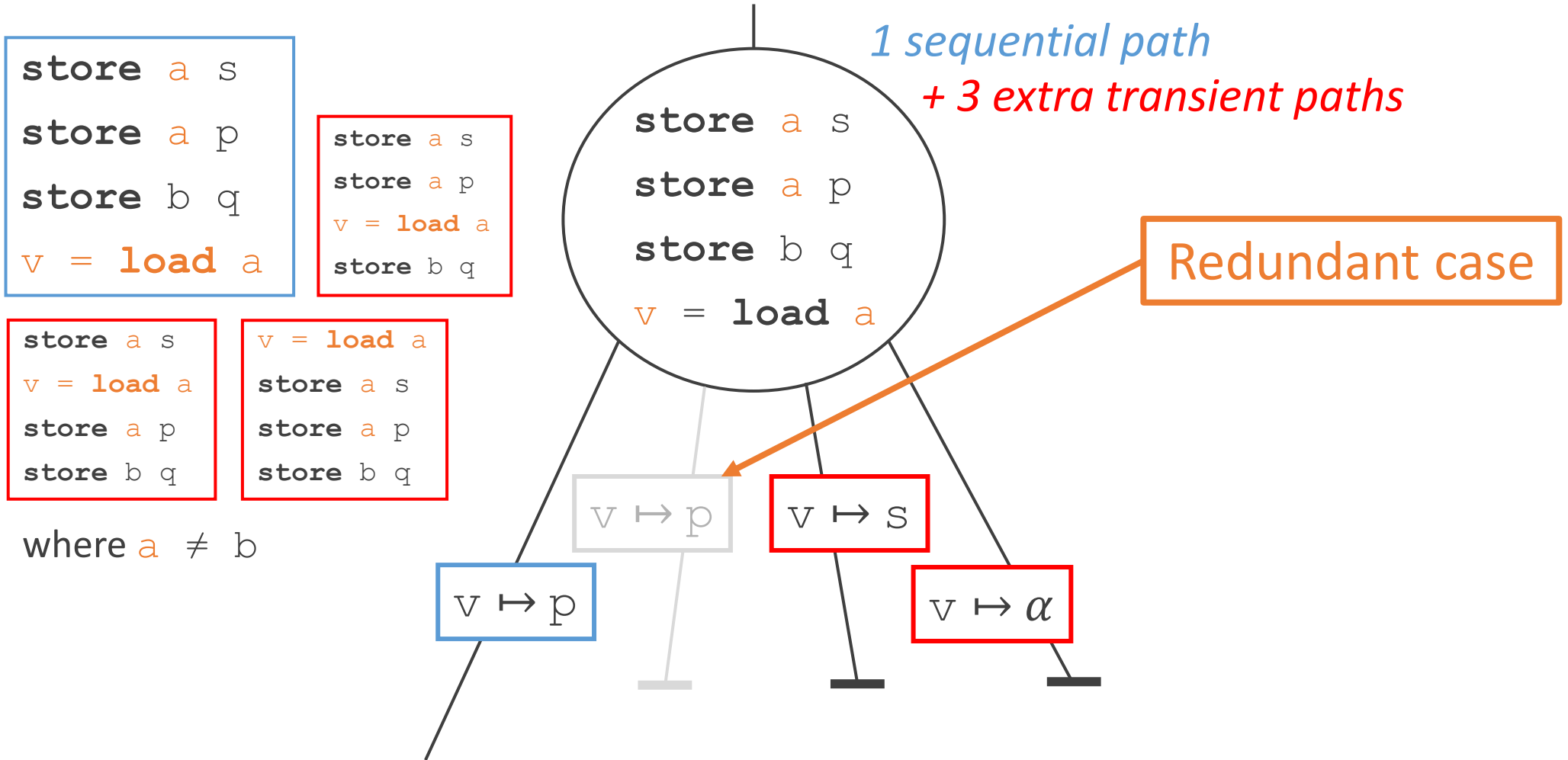
# RelSE for Spectre-PHT (naive)




At load instructions:
Fork execution for each load/store interleaving

(e.g. Pitchfork)

# RelSE for Spectre-STL (but let's be smarter)



# RelSE for Spectre-STL (but let's be smarter)