

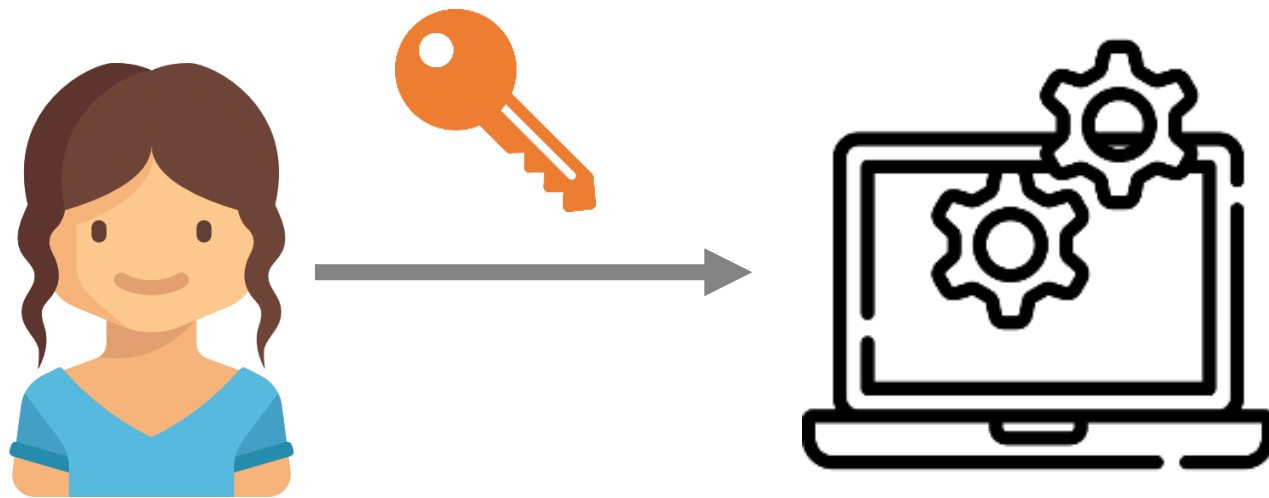
# Beyond constant-time programming: Hardware-software co-designs for microarchitectural security

Au-delà de la programmation en temps-constant:  
Co-conception matérielle-logicielle pour la sécurité microarchitecturale

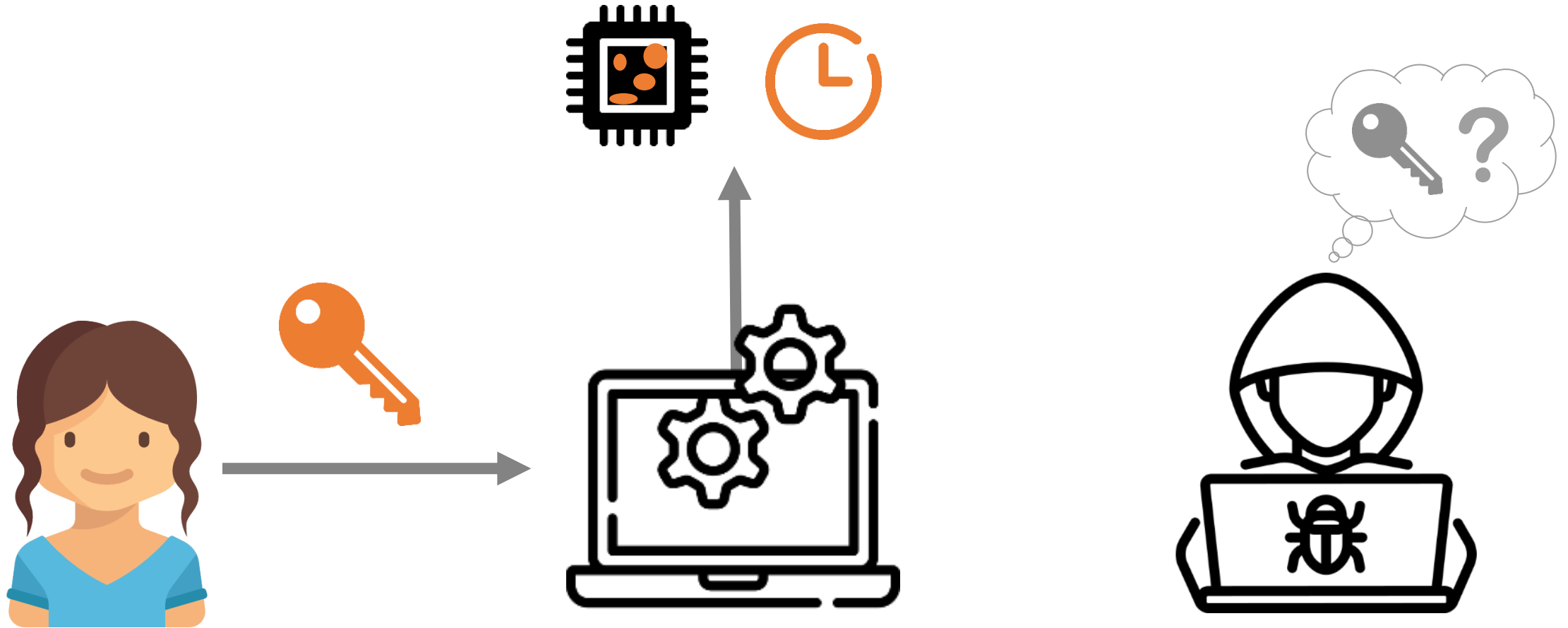
Séminaire Épicure – Rennes  
December 12<sup>th</sup> 2023

**KU LEUVEN**

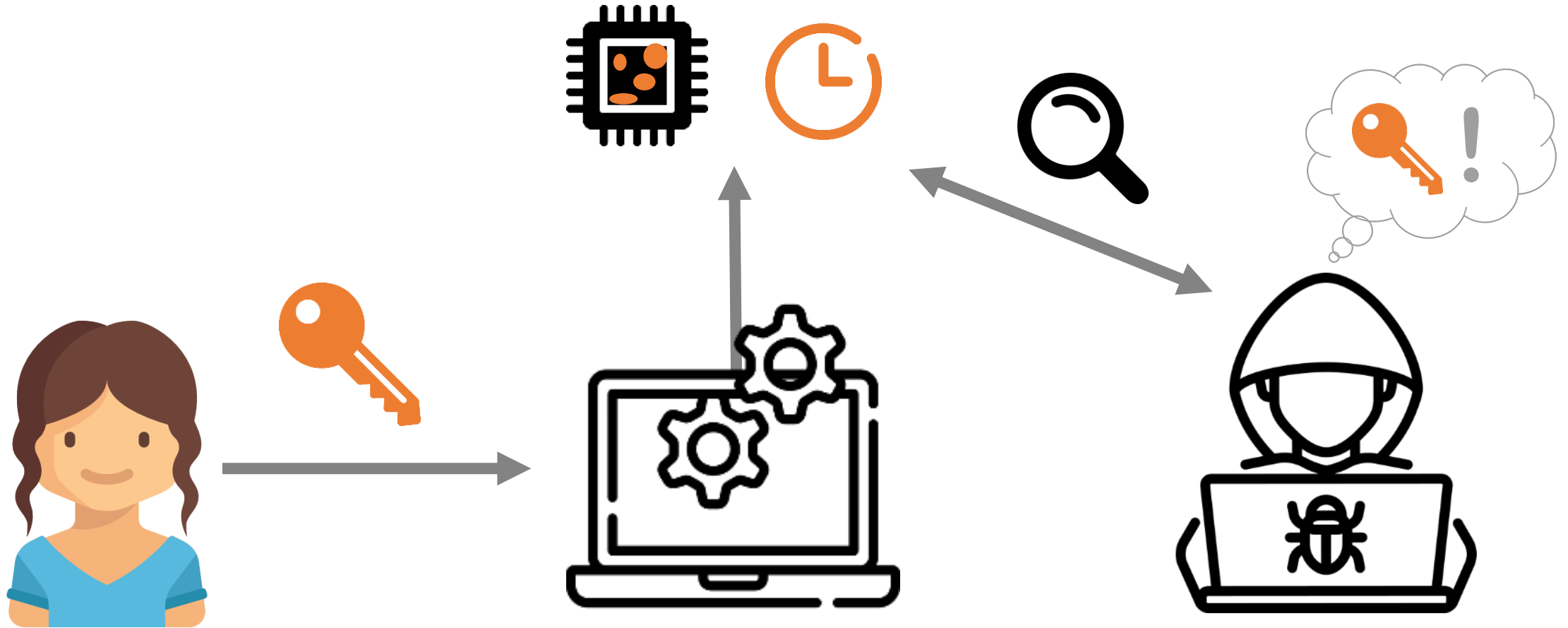
# Programs handle secret data...



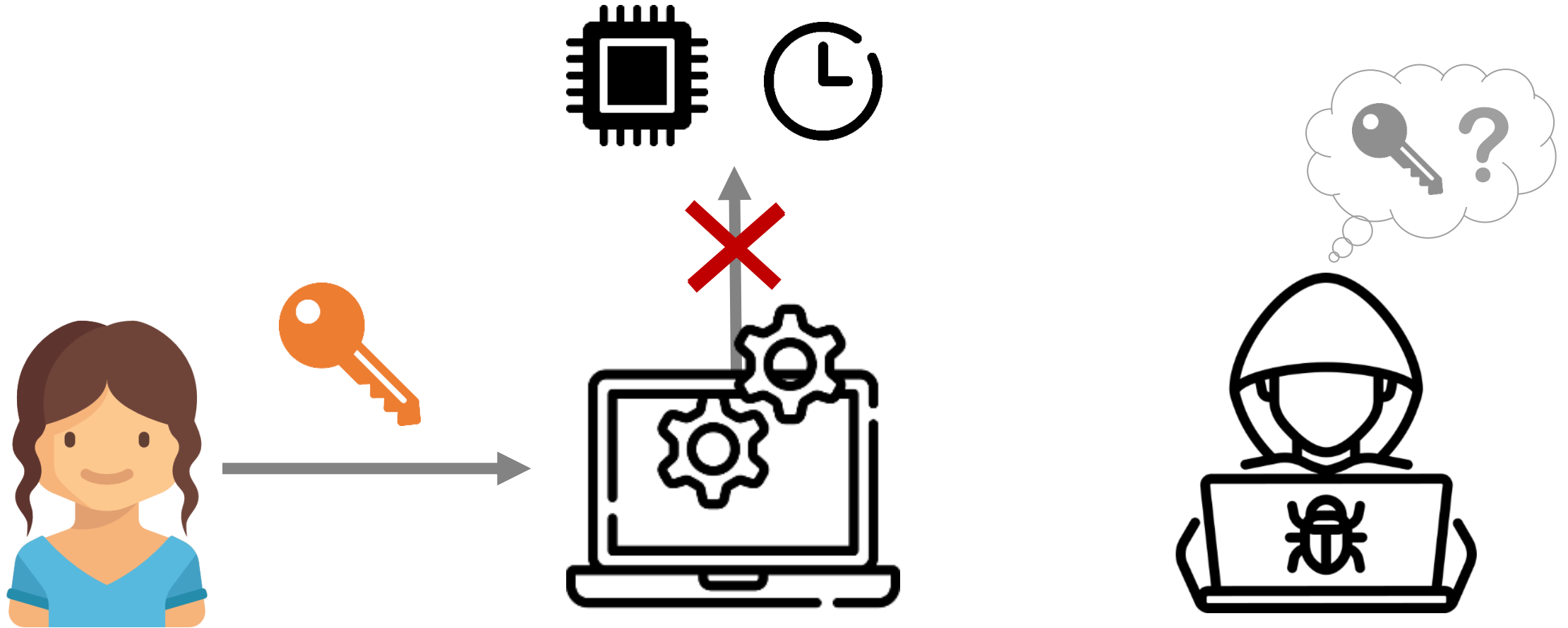
... which can affect timing/microarchitecture



# ... and leak via side-channel attacks



# Solution? Constant-time programming!



# What can influence execution time/microarchitecture?

## Control Flow

```
if secret
```

```
then foo()
```



```
else bar()
```



secret



~~secret~~

# What can influence execution time/microarchitecture?

## Control Flow

```
if secret
```

```
then foo()
```

```
else bar()
```



secret

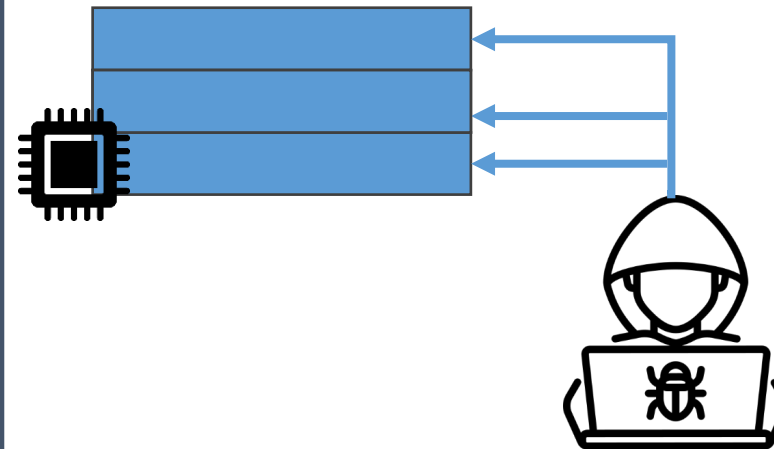


~~secret~~

## Memory Accesses

```
x = buf[secret]
```

### Cache



# What can influence execution time/microarchitecture?

## Control Flow

```
if secret
```

```
then foo()
```

```
else bar()
```



secret

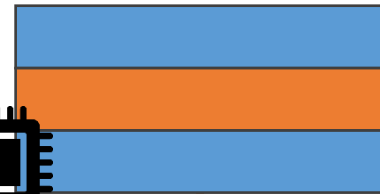
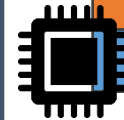


~~secret~~

## Memory Accesses

```
x = buf[secret]
```

Cache

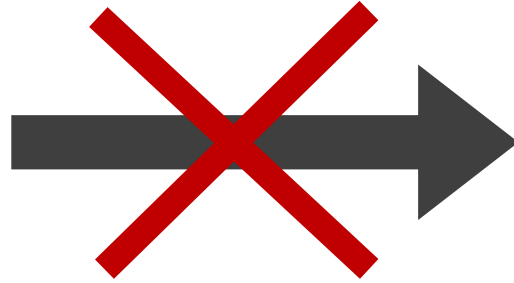


secret





# Solution? Constant-time programming!



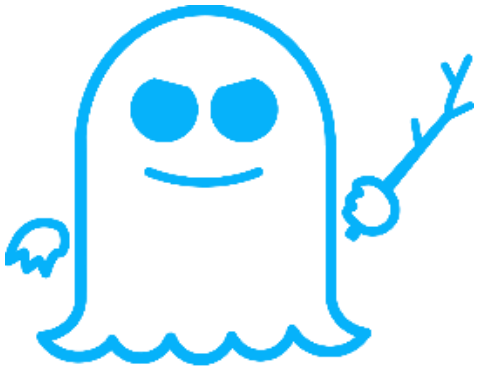
## Leaky instructions

- Control-Flow
- Memory accesses
- Variable-time instr.

- Full software countermeasure
- Plenty of analysis tools: ctgrind, MicroWalk, Binsec/Rel
- De facto standard for crypto: BearSSL, Libsodium, HACL\*, etc.

# Constant-time is not perfect

## Security



Still vulnerable to Spectre

## Efficiency



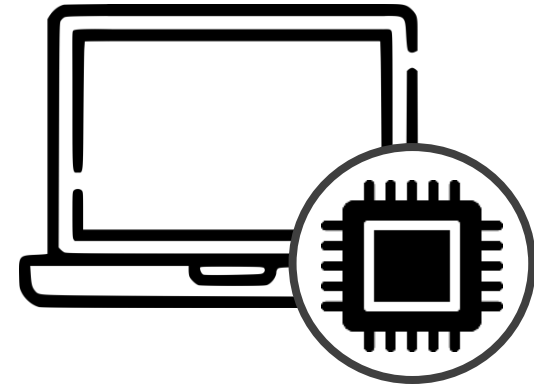
Conservative model  
Affects performance

# End-to-End Solution?



## In Software?

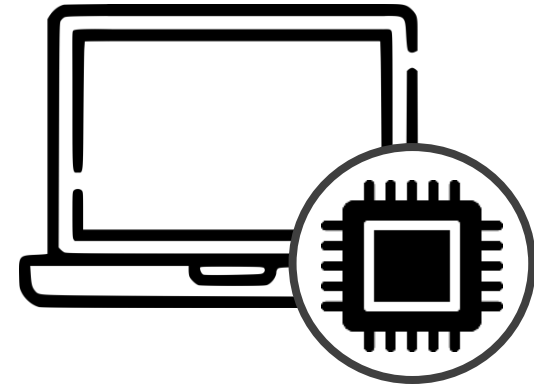
Constant-time Programming  
Speculative Constant-time



## In Hardware?

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

# End-to-End Solution?



**In Software**

Full HW / full SW solutions:

- Partial countermeasures
- Performance

Constant-time Prog  
Speculative Consta

**in Hardware?**

architectural partitioning,  
visible speculation,  
OISA, STT, SPT, ConTeXT, etc.

# End-to-End Solution?



**Hardware-software co-design**

*Best performance/security tradeoff*

# HW/SW Collaboration for End-to-End Security

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

Lesly-Ann Daniel<sup>1</sup>, Marton Bognar<sup>1</sup>, Job Noorman<sup>1</sup>, Sébastien Bardin<sup>2</sup>, Tamara Rezk<sup>3</sup> and Frank Piessens<sup>1</sup>

<sup>1</sup>imec-DistriNet, KU Leuven

<sup>2</sup>CEA, List, Université Paris-Saclay

<sup>3</sup>INRIA, Université Côte d'Azur

### Abstract

We propose PROSPECT, a generic formal processor model providing provably secure speculation for the constant-time policy. It guarantees that the control flow of the program does not depend on secrets, relying on techniques like *dummy execution* (for balancing code) or *conditional execution* (for linearizing code). In the current state-of-practice, the primitives to implement these techniques have to be found in an existing instruction set architecture (ISA) that was not designed a priori to provide them, leading to performance, security, and portability issues. To counter these issues, this paper proposes lightweight hardware extensions for supporting these techniques in a principled way. We propose (1) a novel hardware

## Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs

Hans Winderix  
*imec-DistriNet*  
KU Leuven

Marton Bognar  
*imec-DistriNet*  
KU Leuven

Job Noorman  
*imec-DistriNet*  
KU Leuven

Lesly-Ann Daniel  
*imec-DistriNet*  
KU Leuven

Frank Piessens  
*imec-DistriNet*  
KU Leuven

**Abstract**—The control flow of a program can often be observed through side-channel attacks. Hence, when control flow depends on secrets, attackers can learn information about these secrets. Widely used software-based countermeasures ensure that attacker-observable aspects of the control flow do not depend on secrets, relying on techniques like *dummy execution* (for balancing code) or *conditional execution* (for linearizing code). In the current state-of-practice, the primitives to implement these techniques have to be found in an existing instruction set architecture (ISA) that was not designed a priori to provide them, leading to performance, security, and portability issues. To counter these issues, this paper proposes lightweight hardware extensions for supporting these techniques in a principled way. We propose (1) a novel hardware

benefits compared to other approaches [5], [6]. Second, *linearization* [7]–[10] ensures that control flow does not depend on program secrets at all.

Balancing and linearization are important ingredients in state-of-practice software-based countermeasures (such as *constant-time programming* [11]), as well as in recent research prototypes [5]–[7], [10], [12]. They are based on techniques like *dummy execution* (i.e., using architectural no-ops

32<sup>ND</sup> USENIX  
SECURITY SYMPOSIUM

45th IEEE Symposium on  
Security and Privacy

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

Lesly-Ann Daniel<sup>1</sup>, Marton Bogнар<sup>1</sup>, Job Noorman<sup>1</sup>, Sébastien Bardin<sup>2</sup>, Tamara Rezk<sup>3</sup> and Frank Piessens<sup>1</sup>

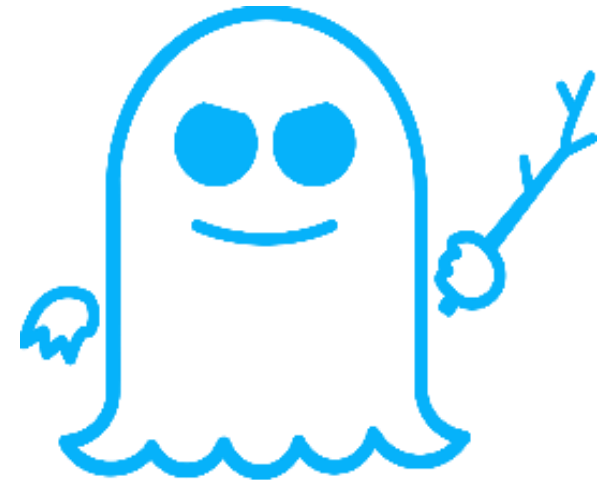
<sup>1</sup>imec-DistriNet, KU Leuven, 3001 Leuven, Belgium

<sup>2</sup>CEA, List, Université Paris Saclay, France

<sup>3</sup>INRIA, Université Côte d'Azur, Sophia Antipolis, France

# Constant-time is vulnerable to Spectre

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





# Constant-time is vulnerable to Spectre

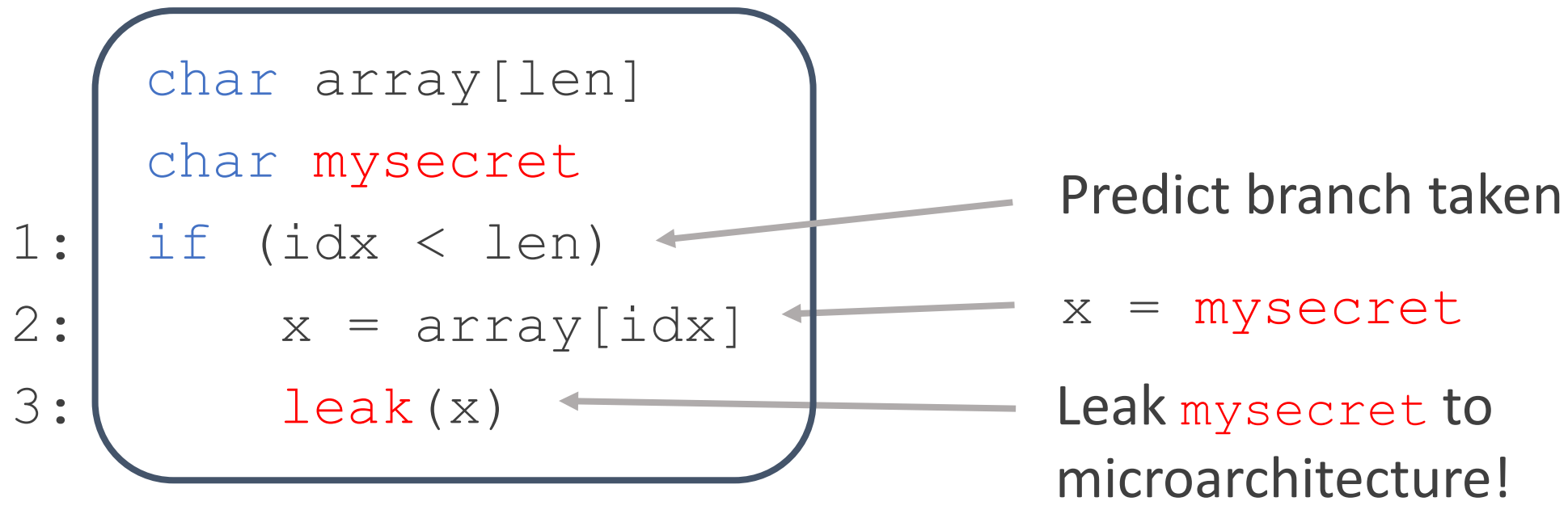
```
char array[len]
char mysecret
1: if (idx < len)
2:     x = array[idx]
3:     leak(x)
```

Consider `idx = len`

Predict branch taken



# Constant-time is vulnerable to Spectre



Consider `idx = len`

# How can I protect my code?

## Constant-Time Foundations for the New Spectre Era

Sunjay Cauligi<sup>†</sup> Craig Disselkoen<sup>†</sup> Klaus v. Gleissenthall<sup>†</sup>  
Dean Tullsen<sup>†</sup> Deian Stefan<sup>†</sup> Tamara Rezk<sup>\*</sup> Gilles Barthe<sup>\*\*</sup>

<sup>†</sup>UC San Diego, USA      <sup>\*</sup>INRIA Sophia Antipolis, France

<sup>♣</sup>MPI for Security and Privacy, Germany      <sup>\*\*</sup>IMDEA Software Institute, Spain

### Speculative constant-time

- Hard to reason about
- New speculation mechanisms?



# We need Secure Speculation for Constant-Time!



Developers should not care about speculations



**Hardware** shall not speculatively leak secrets



But still be efficient and enable **speculation**



**Hardware defense:**

Secure speculation for constant-time!

# Hardware Secrecy Tracking



## Software side

- Label secrets
- Constant-time program

## Hardware side

- Track security labels
- Secrets do not speculatively flow to insecure instructions



### ConTEXT: A Generic Approach for Mitigating Spectre

Michael Schwarz<sup>1</sup>, Moritz Lipp<sup>1</sup>, Claudio Canella<sup>1</sup>, Robert Schilling<sup>1,2</sup>, Florian Kargl<sup>1</sup>, Daniel Gruss<sup>1</sup>  
<sup>1</sup>Graz University of Technology <sup>2</sup>Knight Center for Cyber Security

### SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks

Jacob Fustos

Farzad Farshchi  
University of Kansas

Heechul Yun  
University of Kansas

### Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy

Rutvik Choudhary  
UIUC, USA

Christopher W. Fletcher  
UIUC, USA

Jiyong Yu  
UIUC, USA

Adam Morrison  
Tel Aviv University, Israel

# Illustration with Spectre-v1

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

Consider `idx = len`

# Illustration with Spectre-v1

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

Developer marks secrets



Consider `idx = len`

# Illustration with Spectre-v1

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

Developer marks secrets

Speculative execution



Consider `idx = len`



# Illustration with Spectre-v1

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

Consider `idx = len`

Developer marks secrets

Speculative execution



`x = mysecret : secret`

# Illustration with Spectre-v1

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

Consider `idx = len`

Developer marks secrets

Speculative execution

`x = mysecret` : **secret**

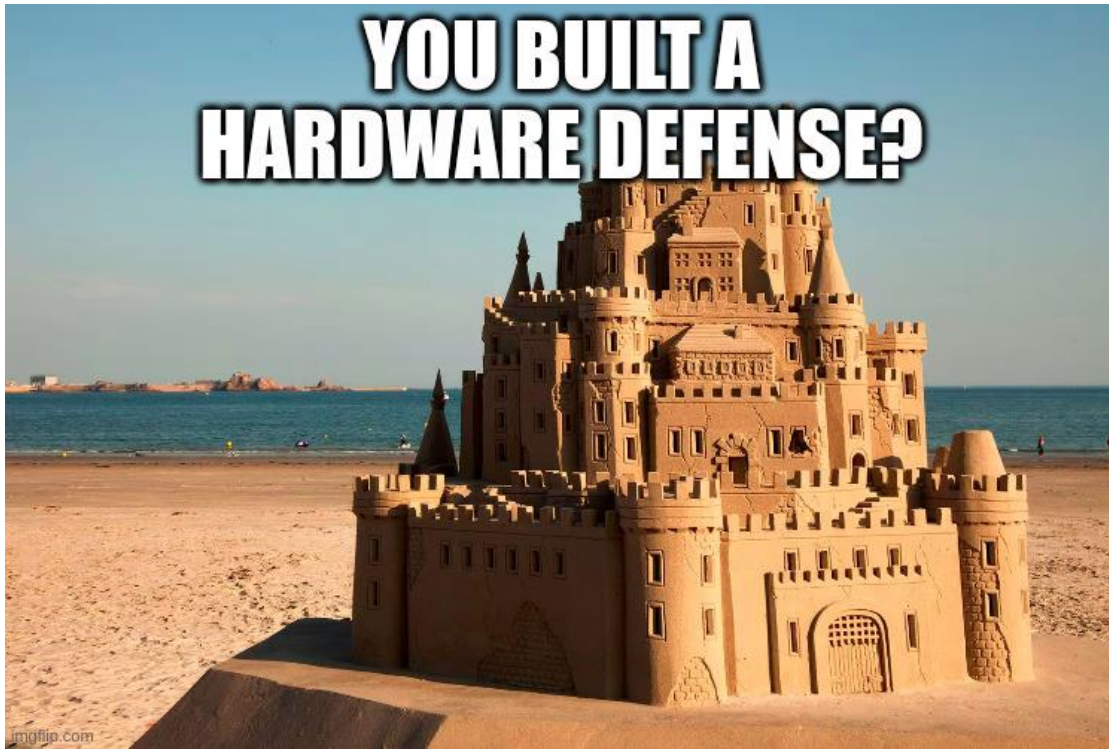
Speculative execution + **secret**

=

`x` *not* forwarded to `leak`



# How do I know that my defense works?



# How do I know that my defense works?

YOU BUILT A  
HARDWARE DEFENSE?

## Hardware-Software Contracts for Secure Speculation

Marco Guarnieri\*, Boris Köpf†, Jan Reineke‡, and Pepe Vila\*

\*IMDEA Software Institute

†Microsoft Research

‡Saarland University

THAT'S CUTE...

# Challenges

Adapt **HW/SW contract** framework to account for

- **All existing** speculation mechanisms (Spectre, LVI)
- **Futuristic** speculation mechanisms (value prediction)
- **Declassification**

# Our contributions

- **ProSpeCT: Formal processor model** with HST
  - **Proof**: constant-time programs do not leak secrets
  - Allows for **declassification**
  - **Generic**: all Spectre variants / LVI
- First to consider **(Load) Value Speculation**
  - **Novel insight**: sometimes need to rollback *correct* speculations for security
- **Implementation** in a RISC-V microarchitecture
  - **First synthesizable** implementation
  - **Evaluation**: hardware cost, performance, annotations



# ProSpeCT: Generic formal processor model for HST

Semantics of **out-of-order speculative** processor with HST

- Abstract microarchitectural context
- Functions *update*, *predict*, *next*

Attacker observations/influence

**All** public values are leaked / influence predictions

Generic/Powerful  
predictors

**Declassify** = write secrets to public memory

- Beware unintentional declassification

# ProSpeCT: Generic formal processor model for HST

Semantics of **out-of-order speculative** processor with HST

- Abstract microarchitectural context
- Functions *update*, *predict*, *next*

Attacker observations/influence

**All** public values are leaked / influence predictions

Generic/Powerful  
predictors

**Declassify** = write secrets to public memory

- Beware unintentional declassification

Security proof



Constant-time programs (ISA semantics)  
do not leak secrets (micro-arch. semantics)



# Load Prediction: Rollback correct executions?

```
secret char mysecret  
1: x = load mysecret  
2: y = x + 4
```

# Load Prediction: Rollback correct executions?

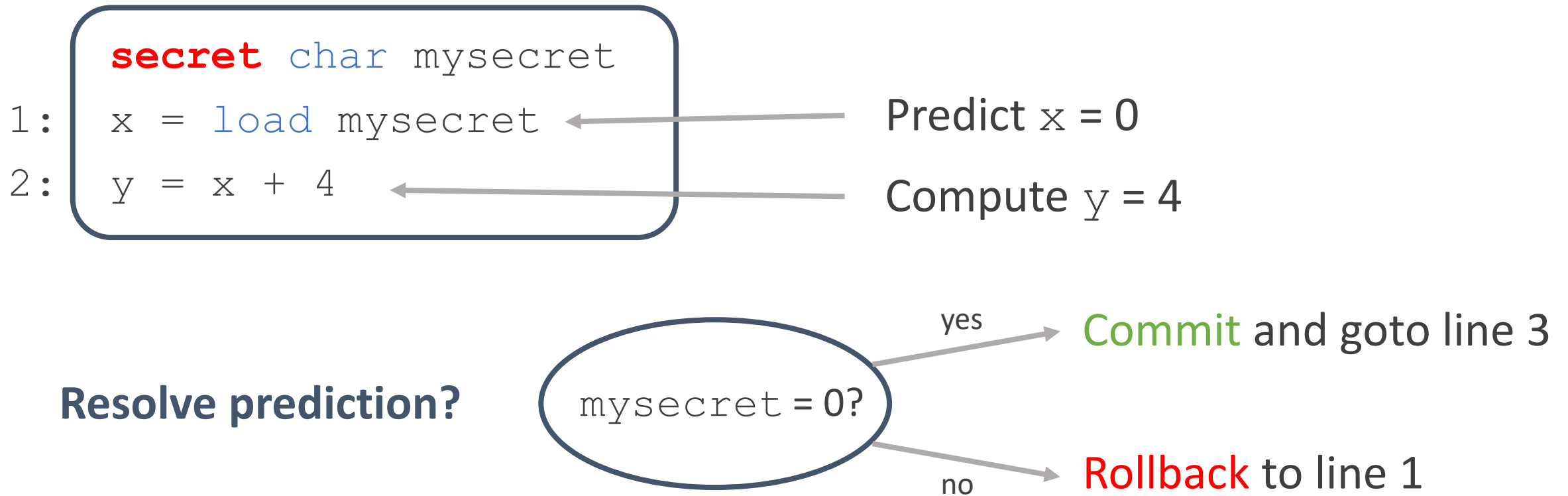
```
secret char mysecret  
1: x = load mysecret  
2: y = x + 4
```

Predict  $x = 0$



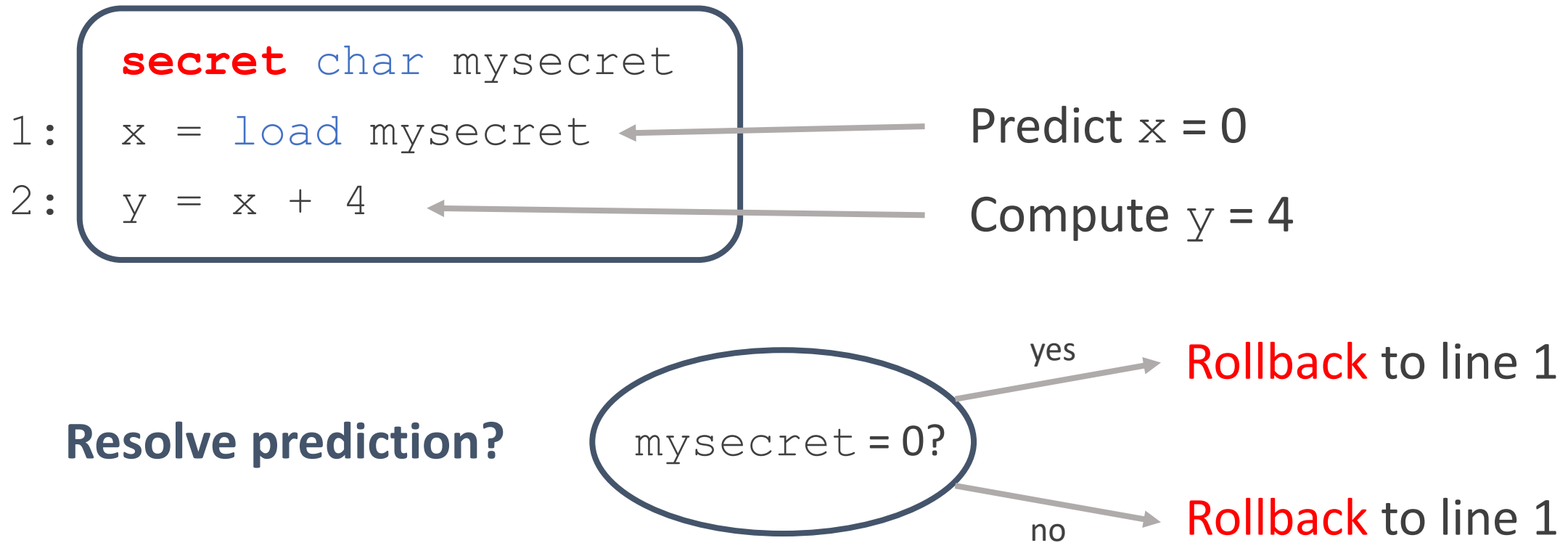
Compute  $y = 4$

# Load Prediction: Rollback correct executions?



*Implicit resolution-based channel!*

# Load Prediction: Rollback correct executions?



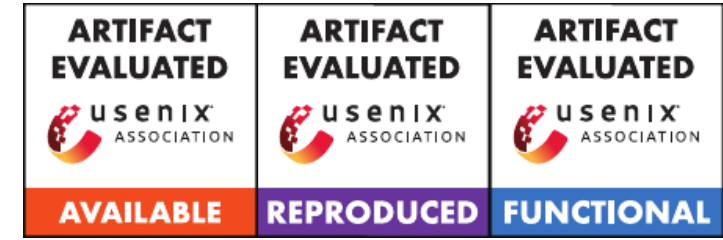
*Solution: always rollback when value is secret*

# Implementation

## Prototype RISC-V implementation

On top of **Proteus** modular RISC-V processor

- Branch target prediction
- Conservative approach
- 2 secret regions defined by CSRs



[github.com/proteus-core/prospect](https://github.com/proteus-core/prospect)

# Evaluation

## Hardware Cost

Synthesized on FPGA

- LUTs: +17%
- Registers: +6%
- Critical path: +2%

## Annotation burden

- 4 primitives (HACL\*)
- Annotate secret
- Ensure no secrets spilled
- Stack public in 3/4 cases
- $\leq 1\text{h}$  / primitive

# Evaluation

## Performance overhead (benchmark from [1])

Speculation/Crypto	25/75	50/50	75/25	90/10
Precise (Key)	0%	0%	0%	0%
Conservative (All)	10%	25%	36%	45%

**No overhead** in software for constant-time code  
when secrets are precisely annotated



[1] Jacob Fustos, Farzad Farshchi, and Heechul Yun. “SpectreGuard: An Efficient Data-Centric Defense Mechanism against Spectre Attacks”. In: DAC. 2019

# Summary



Software informs hardware about secrets



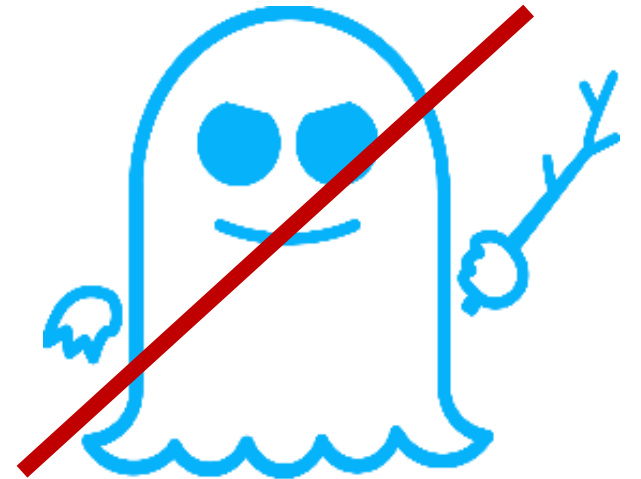
Strong **security** guarantees

*End-to-end security for constant-time programs*



Low overhead

*No software overhead for constant-time code*





# **Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs**

Hans Winderix  
*imec-DistriNet*  
*KU Leuven*

Marton Bognar  
*imec-DistriNet*  
*KU Leuven*

Job Noorman  
*imec-DistriNet*  
*KU Leuven*

Lesly-Ann Daniel  
*imec-DistriNet*  
*KU Leuven*

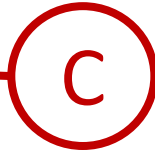
Frank Piessens  
*imec-DistriNet*  
*KU Leuven*

# Secrets can leak via control-flow

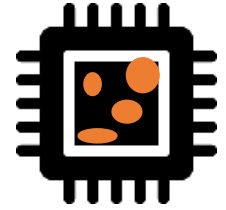
```
if (secret)
    add v a a
    add v v 8
else
    add v a 4
```

# Secrets can leak via control-flow

```
if (secret)
    add v a a
    add v v 8
else
    add v a 4
```

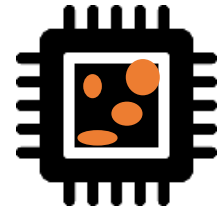
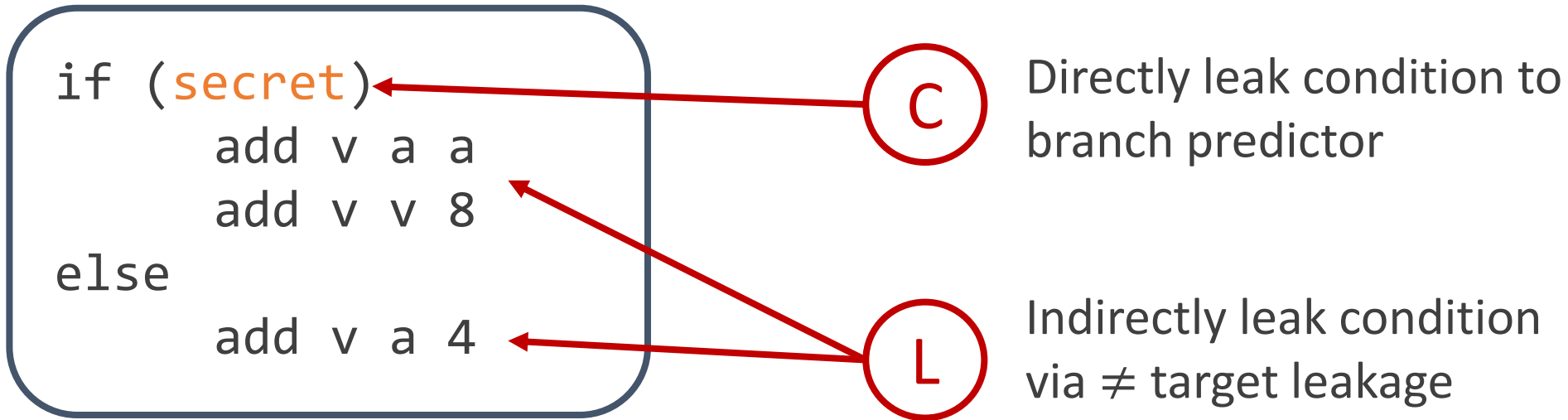


Directly leak condition to  
branch predictor



Conservative leakage model / high-end platforms

# Secrets can leak via control-flow



**C** Conservative leakage model / high-end platforms

**L** Liberal leakage model / small microcontrollers

# Control-Flow Leakage Mitigations?

## Linearization (CT)

```
c = (secret ≠ 0)
add v1 a a
add v1 v1 8
add v2 a 4
v = select c v1 v2
```

(C) + (L)

## Balancing

```
if (secret)
    add v a a
    add v v 8
else
    add v a 4
    add v v 0
```

(L)

# Control-Flow Leakage Mitigations?

## Linearization (CT)

```
c = (secret ≠ 0)
add v1 a a
add v1 v1 8
add v2 a 4
v = select c v1 v2
```

(C) + (L)

## Balancing

```
if (secret)
    add v a a
    add v v 8
else
    add v a 4
    add v v 0
```

(L)

## Issues

- Portability (microcontrollers ≠ servers)
- Performance (extra instructions, registers)
- Security (no security guarantees)

# Goal



**Hardware support and small ISA extension**

*Efficient* and *principled*

control-flow **linearization** and **balancing**

- **Portability:** linearization **(C) + (L)** and balancing **(L)**
- **Performance:** improve over std. linearization/balancing
- **Security:** leakage contract drives secure software development

# Contributions

- Mimic Execution
  - HW **primitive** for mimicking instructions
- Architectural Mimicry (AMi)
  - **Instructions** to control mimic execution
- Programming models
  - **Secure/correct** balancing/linearization with AMi
- Implementation in RISC-V
  - **Evaluation**: hardware cost, performance



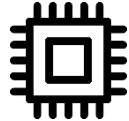



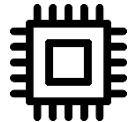
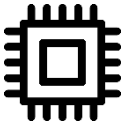

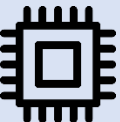


# Mimic execution



2 processor modes  
**standard / mimic**

5 qualifiers  
to control mimic execution

Qualifiers	Standard	Mimic
Activating <b>a.inst</b>	activate mimic mode	
Standard <b>s.inst</b>		
Mimic <b>m.inst</b>		
Persistent <b>p.inst</b>		
Ghost <b>g.inst</b>		

# AMi for Linearization

C

## Insecure Code

```
bnz secret end  
    load v a  
    add v v 1  
end:
```


## AMi

```
a.bnz secret end  
    load v a  
    add v v 1  
end:
```

# AMi for Linearization



## Insecure Code




```
bnz secret end
  load v a
  add v v 1
end:
```

secret = 0

*jmp secret≠0*  
*Load a*  
*add*

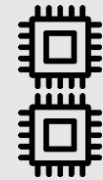
## AMi



```
a.bnz secret end
  load v a
  add v v 1
end:
```

secret = 0

*jmp*  
*Load a*  
*add*



# AMi for Linearization

C

## Insecure Code

```
bnz secret end
  load v a
  add v v 1
end:
```

secret = 0

```
bnz secret end
  load v a
  add v v 1
end:
```

secret ≠ 0

*jmp secret≠0*  
*Load a*  
*add*

*jmp secret≠0*

## AMi

```
a.bnz secret end
  load v a
  add v v 1
end:
```

secret = 0

```
a.bnz secret end
  load v a
  add v v 1
end:
```

secret ≠ 0

*jmp*  
*Load a*  
*add*

*jmp*  
*Load a*  
*add*

v not modified

# AMi for Linearization

C

## Insecure Code

```
bnz secret end
  load v a
  add v v 1
end:
```

secret = 0

```
bnz secret end
  load v a
  add v v 1
end:
```

secret ≠ 0

*jmp secret≠0*  
*Load a*  
*add*

≠

*jmp secret≠0*

## AMi

```
a.bnz secret end
  load v a
  add v v 1
end:
```

secret = 0

```
a.bnz secret end
  load v a
  add v v 1
end:
```

secret ≠ 0

*jmp*  
*Load a*  
*add*

=

*jmp*  
*Load a*  
*add*

*v not modified*

# Maintain Correctness?

```
a.bnz c end  
add v v 1  
add a a 4  
p.store v a  
end
```

## Correctness:

No effect on live state  
in mimic mode



# Maintain Correctness?

```
a.bnz c end
```

```
add v v 1
```

```
add a a 4
```

```
p.store v a
```

```
end
```



**Correctness:**

No effect on live state  
in mimic mode



**Breaks correctness**

$c \neq 0$

# Maintain Correctness?

```
a.bnz c end
```

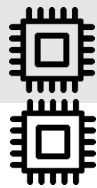
```
add v v 1
```

```
add a a 4
```

```
g.load t a
```

```
p.store t a
```

```
end
```



**Correctness:**

No effect on live state  
in mimic mode

**Correct**



$c \neq 0$



# Enforce Security?

```
a.bnz c end  
add v v 1  
add a a 4  
g.load t a  
p.store t a  
end
```

## Security:

Leakage independent  
of processor mode



# Enforce Security?

**a.**bnz c end

add v v 1

add a a 4

**g.**load t a

**p.**store t a

end



*add*

*add*

*Load 0*

*store 0*

$a = 0$      $c \neq 0$

**a.**bnz c end

add v v 1

add a a 4

**g.**load t a

**p.**store t a

end



*add*

*add*

*Load 4*

*store 4*

$a = 0$      $c = 0$

**Security violation!**

# Enforce Security?

**a.**bnz c end

add v v 1

**p.**add a a 4

**g.**load t a

**p.**store t a

end



*add*

*add*

*Load 4*

*store 4*

$a = 0$      $c \neq 0$

**a.**bnz c end

add v v 1

**p.**add a a 4

**g.**load t a

**p.**store t a

end



*add*

*add*

*Load 4*

*store 4*

$a = 0$      $c = 0$

**Secure**

+correct assuming a, t are not live

# Formalization

- Operational ISA semantics for AMi
  - Instrumented with leakage
- Definition well-behaved activating regions
  - Proof (under conditions) activating regions are well-behaved
  - Nested and recursive activation
- Definitions correct / secure programming model
  - Linearization
  - Balancing

# Implementation

## Prototype 32-bit RISC-V implementation on Proteus

- In order & out-of-order
- Mimic instr = no update register file
- Mode-independent stalling
- Constant-time branch (no prediction)
- 2-bit for instruction qualifiers + 3 CSR for store processor mode



# Evaluation

## Benchmark

11 programs from [1]

4 configurations:

Balancing (B) with/wo AMi

Linearization (L) Molnar/AMi

## Results

- Security: tests
- Hardware: max 26% LUTS/FF, 0% CP
- Binary size: +19% → +0% (L)\*
- Performance: +48% → +19% (L)\*

\*No change for balancing

[1] H. Winderix, J. T. Mühlberg, and F. Piessens, “Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks,” in EuroS&P, 2021.

# Summary



Principled **linearization** and **balancing**



**Security**-oriented ISA extension

*Can be leveraged to write side-channels free sw*



Accelerate CT code

*-60% overhead of linearized code*



# A step back



RISC-V open standard ISA

→ HW-SW co-design for security



- Proteus: extensible RISC-V processor

- Security extensions

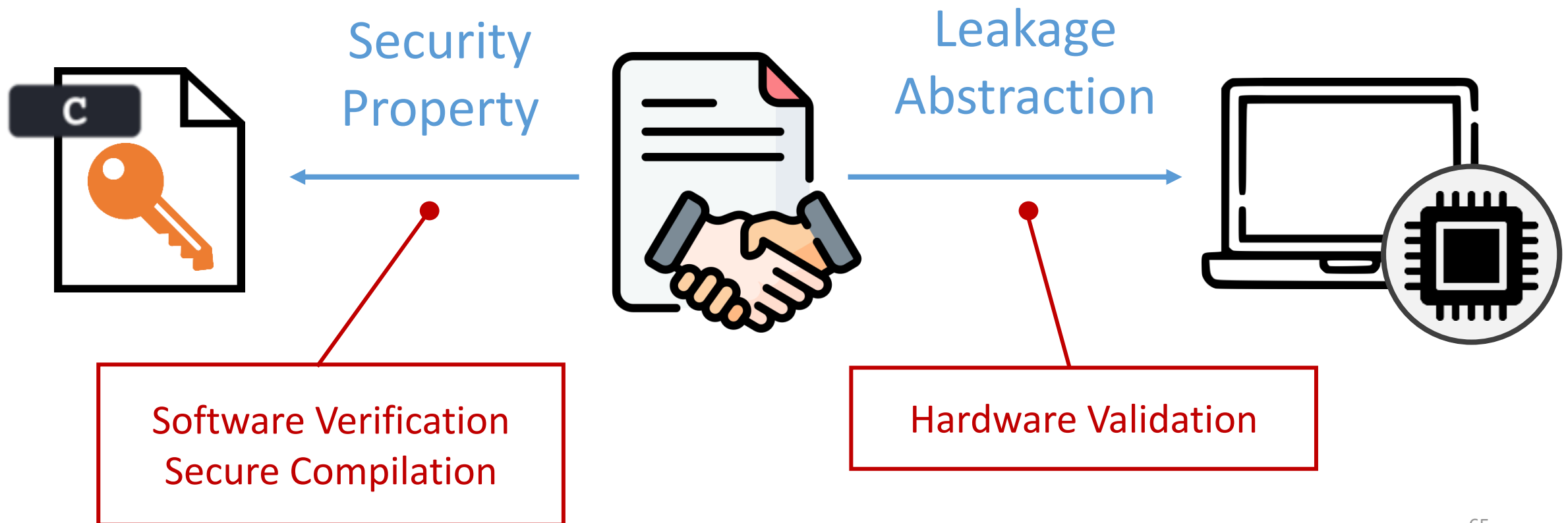
- ProSpeCT
- AMi
- CHERI
- ...



# Hardware-Software Contracts for Secure Speculation

Marco Guarnieri\*, Boris Köpf†, Jan Reineke‡, and Pepe Vila\*

\**IMDEA Software Institute*    †*Microsoft Research*    ‡*Saarland University*



# How to ease adoption of HW-SW co-designs?

Need infrastructure around HW-SW contracts

- **Compilation** support (LLVM / Jasmin)  
*Support larger code, more realistic performance evaluation*
- **Validate HW** implementation (fuzzing / verification)  
*Reduce gap between model and implementation*

# Conclusion

## ProSpeCT



Software informs hardware about secrets



Strong **security** guarantees

*End-to-end security for constant-time programs*



Low overhead

*No software overhead for constant-time code*



## Architectural Mimicry



Principled **linearization** and **balancing**



**Security**-oriented ISA extension

*Can be leveraged to write side-channels free sw*



Accelerate CT code

*-60% overhead of linearized code*



**Goal:** end-to-end security & performance

Backup

# AMi for Balancing



## Insecure Code

```
bnz secret else      jmp  
    add v a a        add  
    add v v 8        add  
    j end            jmp  
else:  
    add v a 4        add  
  
end:
```

# AMi for Balancing L

## Insecure Code

```
bnz secret else      jmp
    add v a a
    add v v 8
    j end
else:
    add v a 4
end:
```

add  
add  
jmp


add

## AMi Balancing

```
bnz secret else      jmp
    add v a a
    add v v 8
    j end
else:
    add v a 4
    m.add v v 8
    j end
end:
```

add  
add  
jmp

add  
add  
jmp

m.add v v 8  


# Hardware Costs

	LUT	Flip-Flops	Critical path
In-order	+19.5%	+22.9%	<b>+0.6%</b>
Out-of-order	+26.3%	+9.2%	<b>-1.0%</b>

Hardware cost overhead of AMi (synthesized on an FPGA)

# Binary Size

	Baseline size (bytes)	Balanced		Linearized	
		No AMi	AMi	Molnar	AMi
Min	132	+0%	+0%	+8%	-6%
Max	500	+41%	+41%	<b>+92%</b>	<b>+2%</b>
Mean	321	+8%	+7%	<b>+19%</b>	<b>+0%</b>

Binary size overhead compared to insecure baseline



# Execution time

	Balanced (in order)		Linearized (in order)		Linearized (ooo)	
	No AMi	AMi	Molnar	AMi	Molnar	AMi
Min	+6%	+6%	+9%	-11%	-5%	-11%
Max	+143%	+143%	+275%	+69%	+233%	+77%
Mean	+59%	+59%	<b>+57%</b>	<b>+24%</b>	<b>+48%</b>	<b>+19%</b>

Execution time overhead compared to insecure baseline (cycles)