

# Architectural Mimicry

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

Hans Winderix  
*imec-DistriNet*  
*KU Leuven*

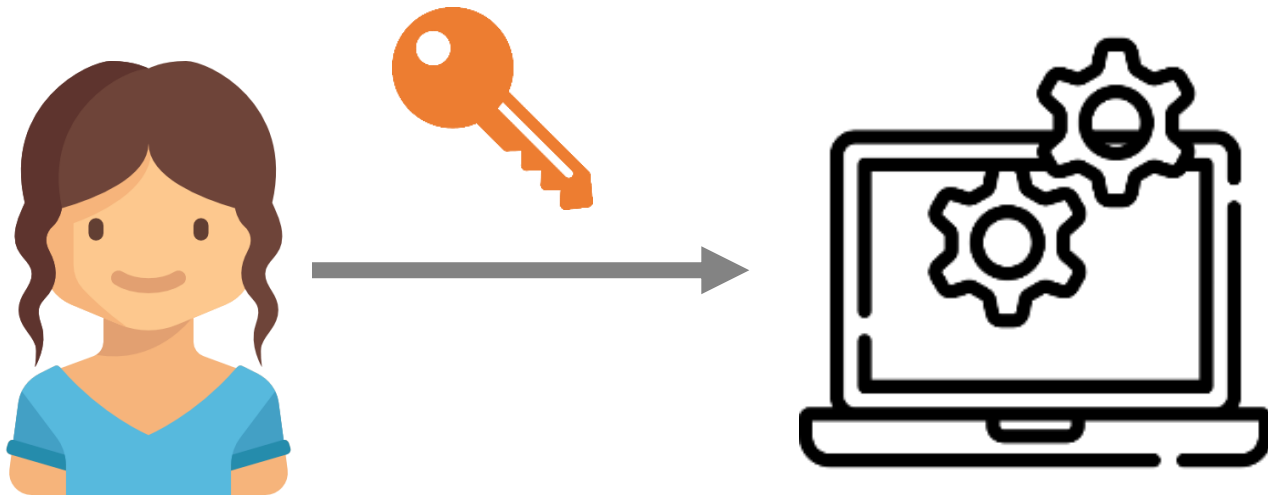
Marton Bognar  
*imec-DistriNet*  
*KU Leuven*

Job Noorman  
*imec-DistriNet*  
*KU Leuven*

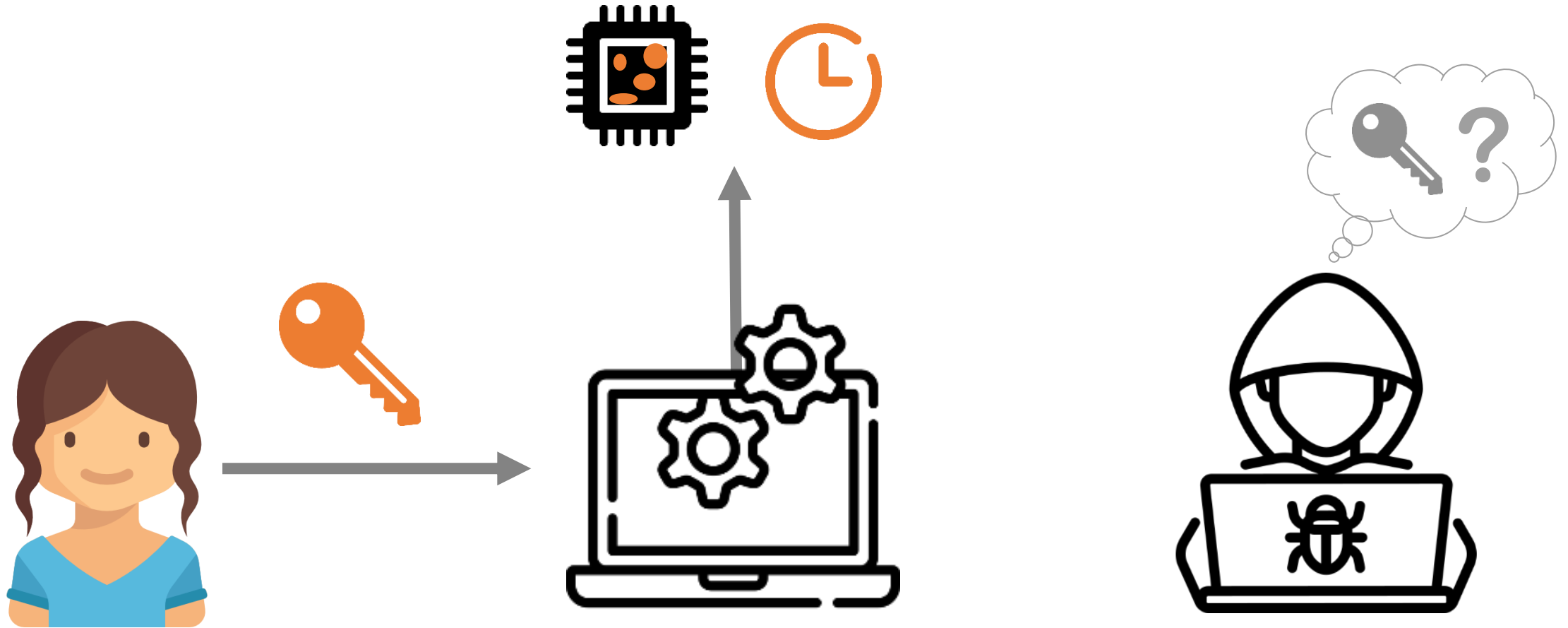
Lesly-Ann Daniel  
*imec-DistriNet*  
*KU Leuven*

Frank Piessens  
*imec-DistriNet*  
*KU Leuven*

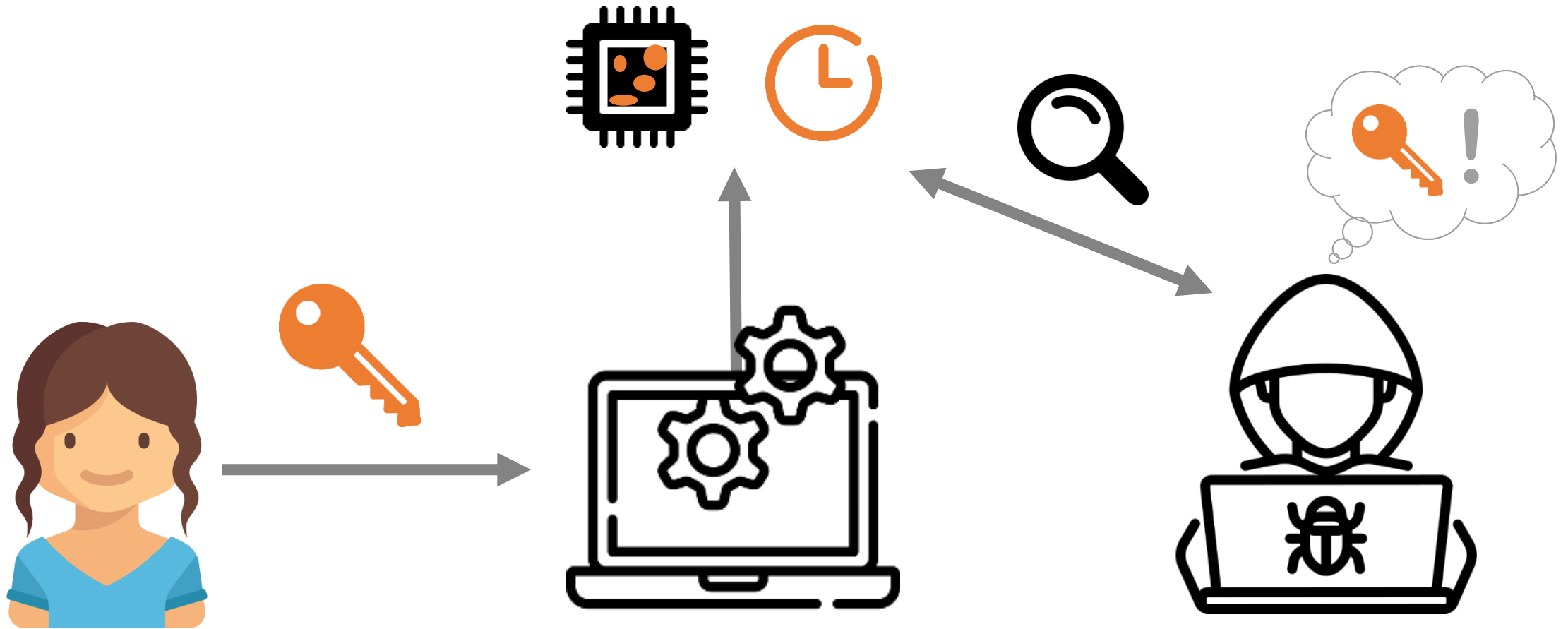
# Programs handle secret data...



... which can affect timing/microarchitecture



# ... and leak via side-channel attacks

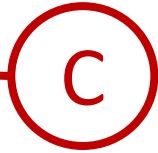


# Secrets can leak via control-flow

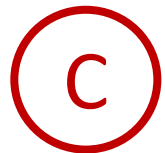
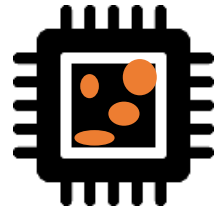
```
if (secret)
    add v a a
    add v v 8
else
    add v a 4
```

# 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

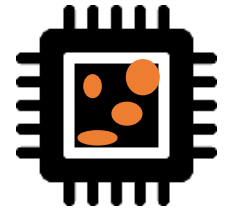
```
if (secret)
  add v a a
  add v v 8
else
  add v a 4
```

C

Directly leak condition to branch predictor

L

Indirectly leak condition via  $\neq$  target leakage



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

# 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




## 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 Balancing

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



# AMi for Linearization



## Insecure Code

```
bnz secret else      jmp secret≠0
    add v a a        add
    add v v 8        add
    j end            jmp
else:
    add v a 4        add
end:
```

# AMi for Linearization



## Insecure Code

```
bnz secret else      jmp secret≠0
    add v a a         add
    add v v 8         add
    j end            jmp
else:
    add v a 4         add
end:
```

## Linearization

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



# AMi for Linearization

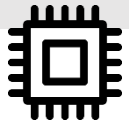



## Insecure Code

```
bnz secret else      jmp secret≠0
    add v a a         add
    add v v 8         add
    j end            jmp
else:
    add v a 4         add
end:
```

## Linearization

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



when secret ≠ 0

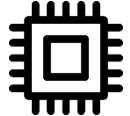

# AMi for Linearization



## Insecure Code

```
bnz secret else      jmp secret≠0
    add v a a         add
    add v v 8         add
    j end            jmp
else:
    add v a 4         add
end:
```

## Linearization

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

when secret = 0

# 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**



$c \neq 0$   
(branch taken)

# Maintain Correctness?

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

$c \neq 0$   
(branch taken)

**Correctness:**

No effect on live state  
in mimic mode

**Correct**



# 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
```

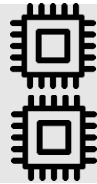
```
add v v 1
```

```
add a a 4
```

```
g.load t a
```

```
p.store t a
```

```
end
```



*add*

*add*

*Load 0*

*store 0*

$a = 0$     $c \neq 0$   
(branch taken)

```
a.bnz c end
```

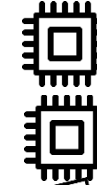
```
add v v 1
```

```
add a a 4
```

```
g.load t a
```

```
p.store t a
```

```
end
```



*add*

*add*

*Load 4*

*store 4*

$a = 0$     $c = 0$   
(branch non-taken)

**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*

$a = 0$     $c \neq 0$   
(branch taken)

```
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*

$a = 0$     $c = 0$   
(branch non-taken)

**Secure**

+correct assuming a is 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

# Evaluation

## Benchmark

11 programs from [1]

4 configurations:

Balancing (B) with/wo AMi

Linearization (L) Molnar/AMi

## Research question

- Security: tests
- Hardware
- Binary size
- Performance

[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.

# Implementation

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

- In order & out-of-order
- Mimic instruction = not update register file
- Mode-independent stalling
- Constant-time branch (no prediction)
- 2-bit for instruction qualifiers + 3 CSR for store processor mode



# Hardware Costs

	LUT	Flip-Flops	Critical path
<b>In-order</b>	+19.5%	+22.9%	<b>+0.6%</b>
<b>Out-of-order</b>	+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
<b>Min</b>	+6%	+6%	+9%	-11%	-5%	-11%
<b>Max</b>	+143%	+143%	+275%	+69%	+233%	+77%
<b>Mean</b>	+59%	+59%	<b>+57%</b>	<b>+24%</b>	<b>+48%</b>	<b>+19%</b>

Execution time overhead compared to insecure baseline (cycles)

# Conclusion



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*



# Future Work?

## 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*



Backup