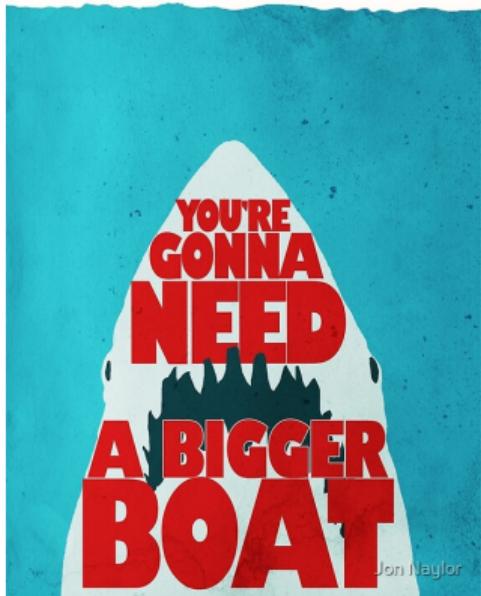


# Lessons from Capsizing SGX Enclave Programs

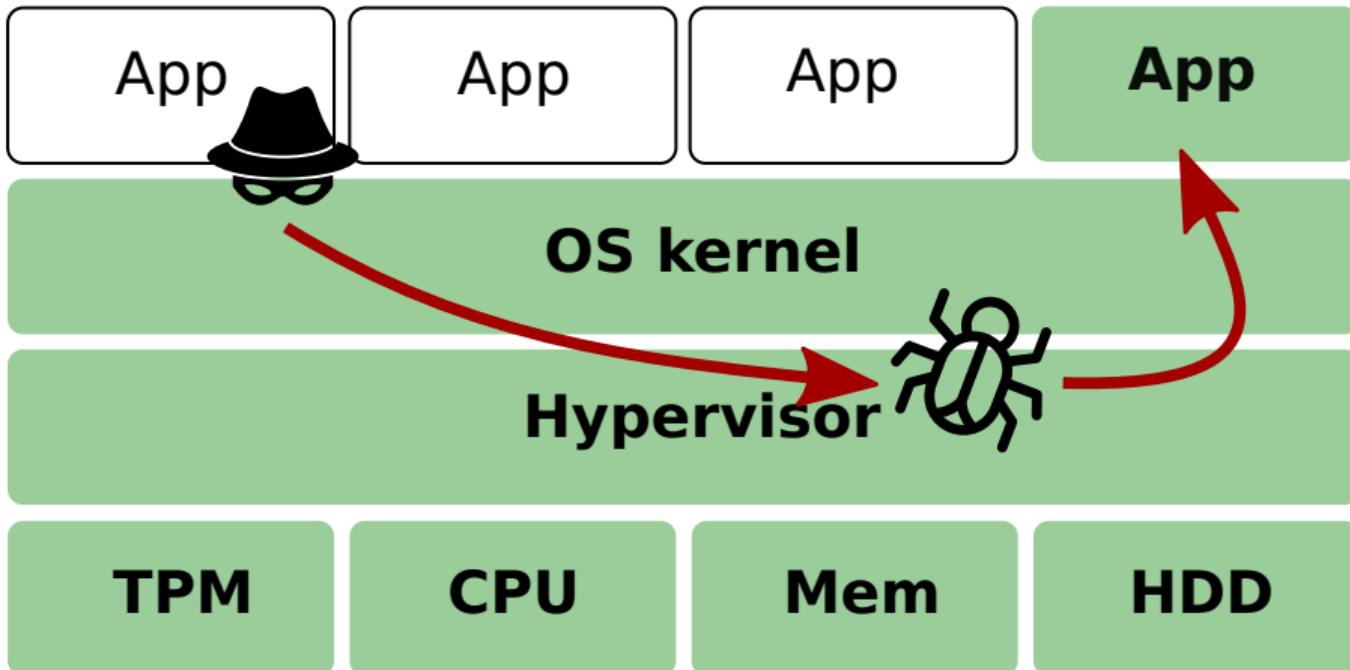
Jo Van Bulck

BINSEC Webinar (online), February 10, 2022

🏡 imec-DistriNet, KU Leuven 📩 jo.vanbulck@cs.kuleuven.be 🐦 jovanbulck

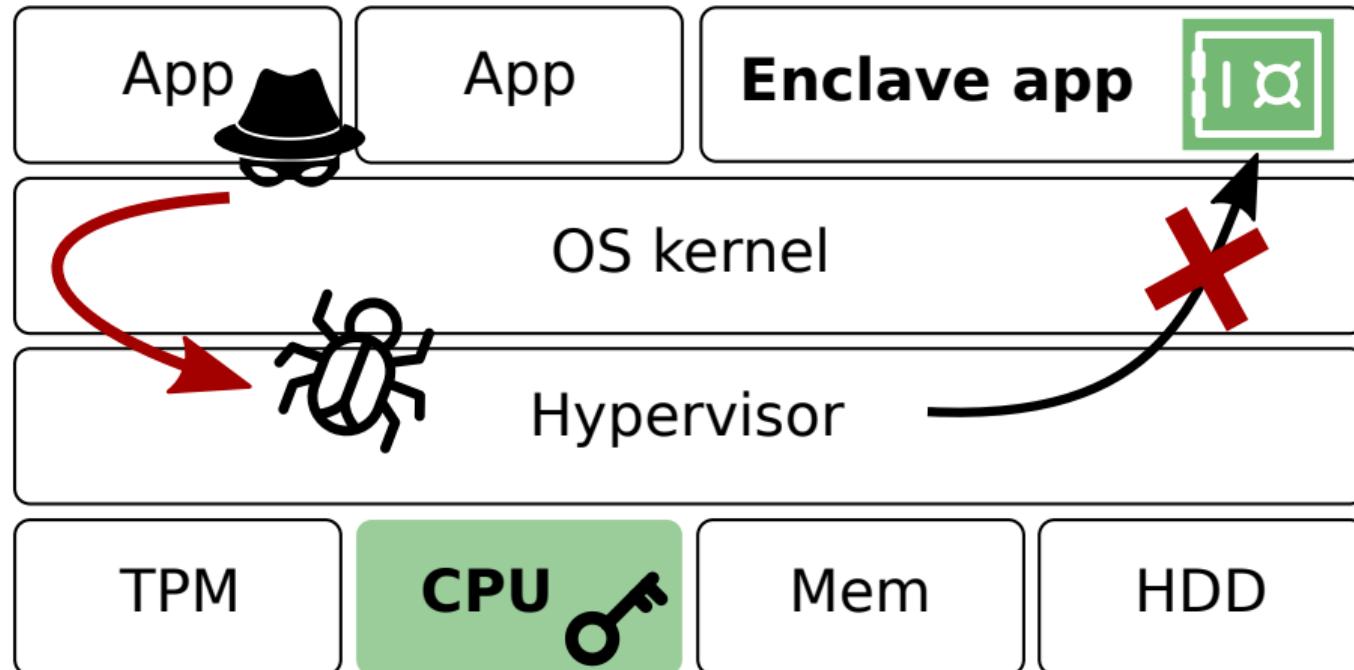


# Enclaved execution: Reducing attack surface



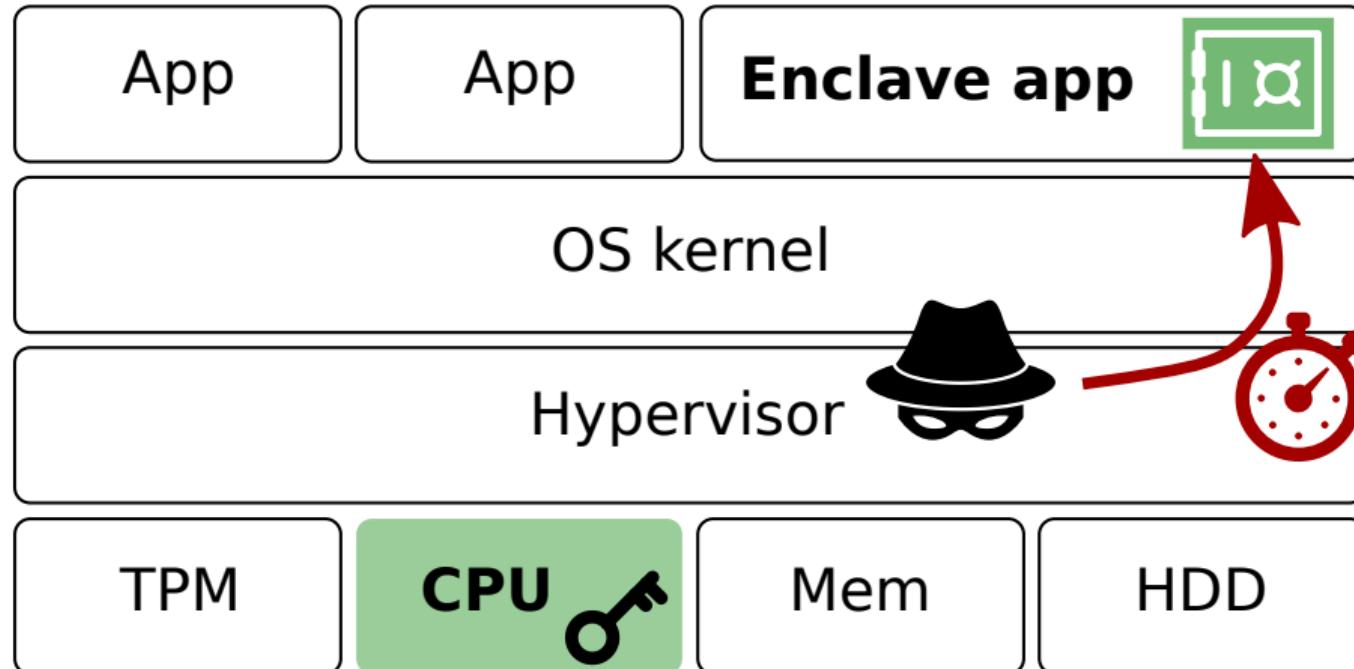
Traditional **layered designs**: large **trusted computing base**

# Enclaved execution: Reducing attack surface



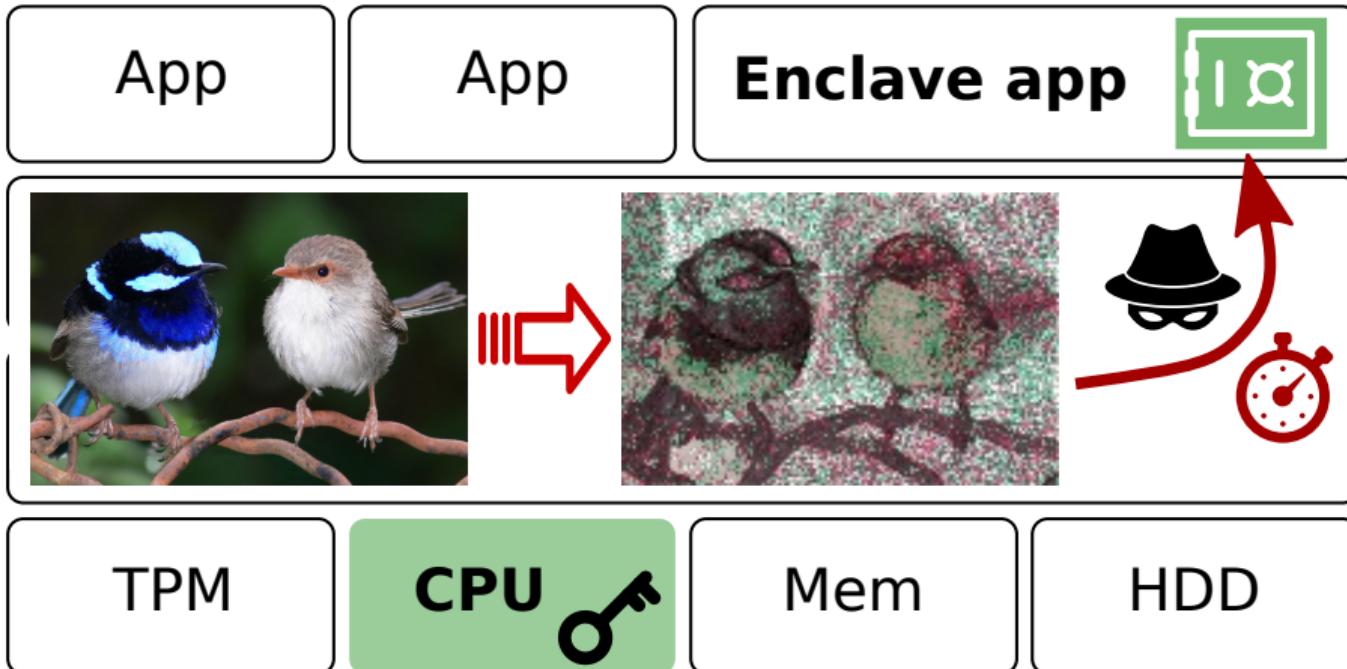
Intel SGX promise: hardware-level **isolation** and attestation

# Enclaved execution: Privileged side-channel attacks



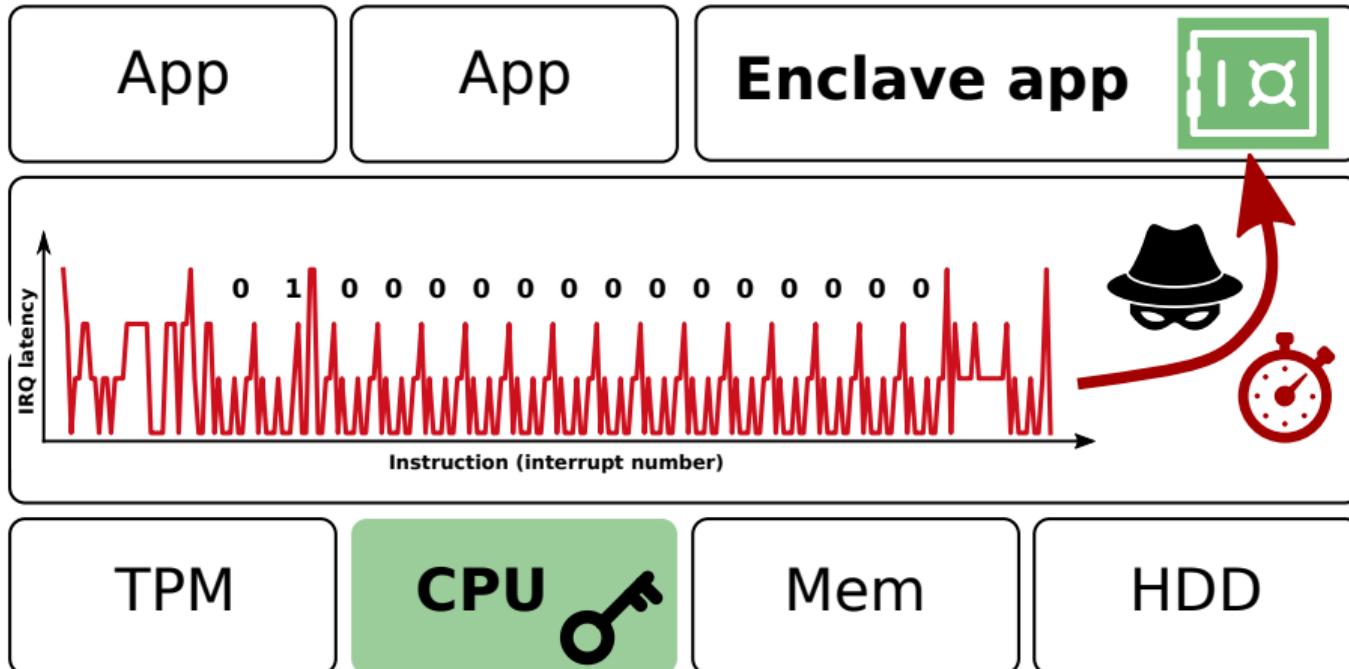
**Game-changer:** Untrusted OS → new class of powerful **side channels!**

# Enclaved execution: Privileged side-channel attacks



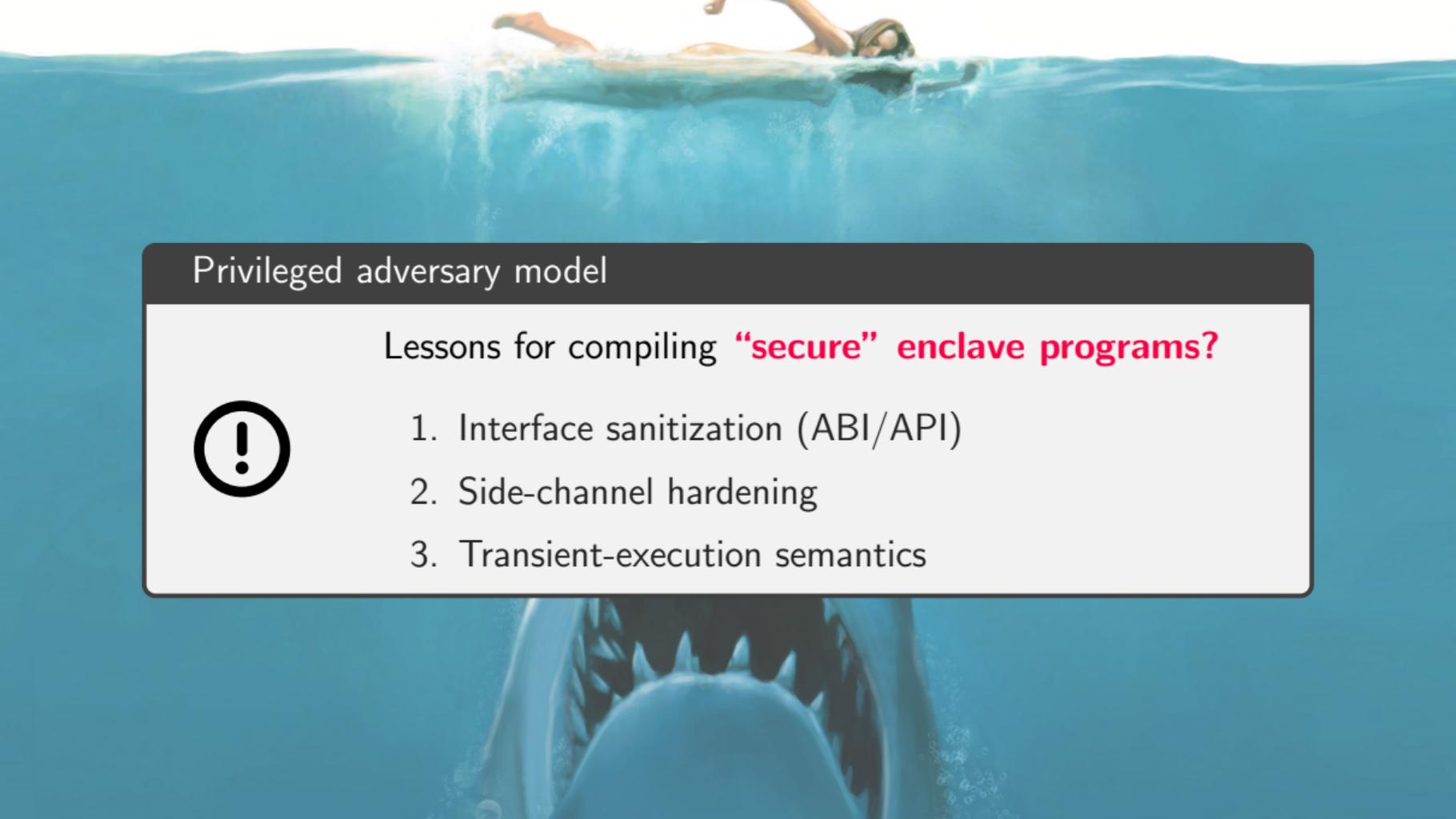
Xu et al. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems", S&P 2015

## Enclaved execution: Privileged side-channel attacks



Van Bulck et al. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic", CCS 2018



The background image shows a person swimming in clear blue water, with the surface line visible. Below the water, a large shark's mouth is open, showing its sharp teeth, creating a sense of danger and focus on security. 

## Privileged adversary model

Lessons for compiling “**secure” enclave programs?**



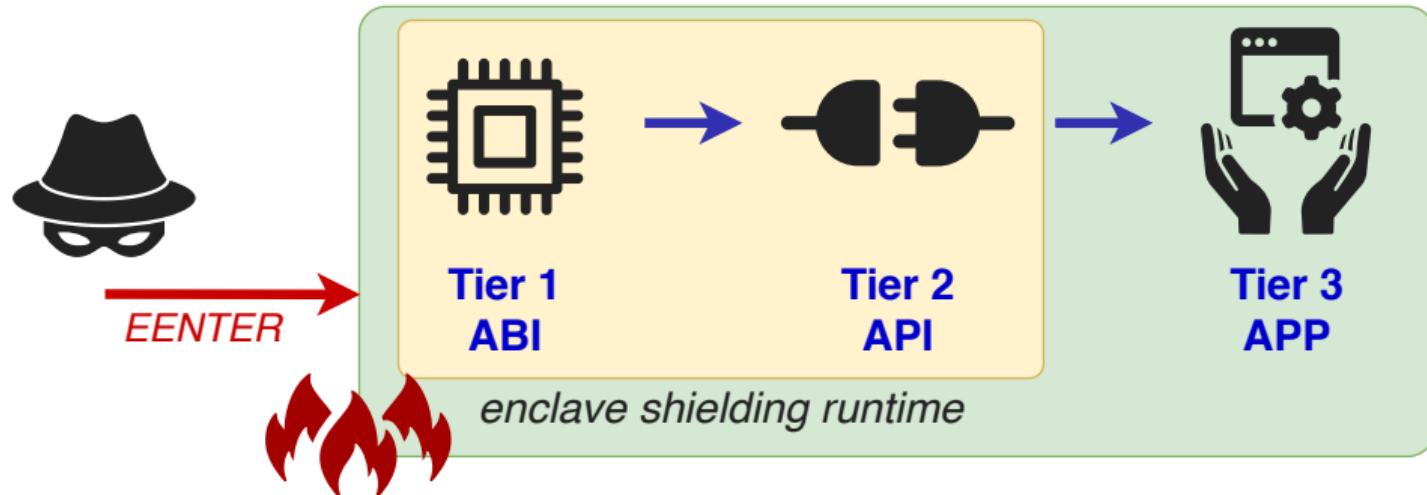
1. Interface sanitization (ABI/API)
2. Side-channel hardening
3. Transient-execution semantics



## Challenge 1: ABI sanitization

---

# Enclave shielding responsibilities

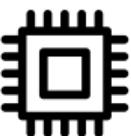


Van Bulck et al. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes", CCS 2019.

# Tier1: Establishing a trustworthy enclave ABI



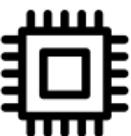
- ~> Attacker controls **CPU register contents** on enclave entry/exit
- ↔ **Compiler** expects well-behaved **calling convention** (e.g., stack)



# Tier1: Establishing a trustworthy enclave ABI



- ~> Attacker controls **CPU register contents** on enclave entry/exit
- ↔ **Compiler** expects well-behaved **calling convention** (e.g., stack)



- ⇒ Need to **initialize CPU registers** on entry and **scrub** before exit!
-

# Summary: ABI-level attack surface

Vulnerability \ Runtime	SGX-SDK	OpenEnclave	Graphene	SGX-LKL	Rust-EDP	Asylo	Keystone	Sancus
Tier1 (ABI)	#1 Entry status flags sanitization ★	★	○	●	○	●	○	○
	#2 Floating-point register sanitization ★	★	○	★	★	●	★	○
	#3 Entry stack pointer restore ○	○	★	●	○	○	○	★
	#4 Exit register leakage ○	○	○	★	○	○	○	○



Relatively understood, but special care for **stack pointer + status register + FPU**

Van Bulck et al. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes", CCS 2019.

Alder et al. "Faulty Point Unit: ABI Poisoning Attacks on Intel SGX", ACSAC 2020.

# Summary: ABI-level attack surface

Vulnerability \ Runtime	SGX-SDK	OpenEnclave	Graphene	SGX-LKL	Rust-EDP	Asylo	Keystone	Sancus	
Tier1 (ABI)	#1 Entry status flags sanitization	★	★	○	●	○	●	○	○
	#2 Floating-point register sanitization	★	★	○	★	★	●	★	○
	#3 Entry stack pointer restore	○	○	★	●	○	○	○	★
	#4 Exit register leakage	○	○	○	★	○	○	○	○

x86 CISC (Intel SGX)

RISC



Attack surface **complex x86 ABI** (Intel SGX) >> simpler **RISC** designs

# x86 string instructions: Direction Flag (DF) operation



- x86 rep string instructions to speed up streamed memory operations

---

```
1 /* memset(buf, 0x0, 100) */
2 for (int i=0; i < 100; i++)
3     buf[i] = 0x0;
```

---



---

```
1 lea rdi, buf
2 mov al, 0x0
3 mov ecx, 100
4 rep stos [rdi], al
```

---

# x86 string instructions: Direction Flag (DF) operation



- [x86 rep string instructions](#) to speed up streamed memory operations
- Default operate **left-to-right**

---

```
1 /* memset(buf, 0x0, 100) */
2 for (int i=0; i < 100; i++)
3     buf[i] = 0x0;
```

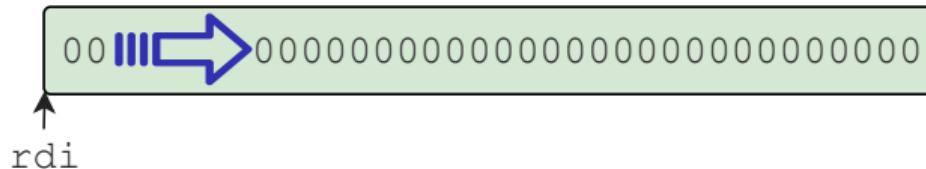
---



---

```
1 lea rdi, buf
2 mov al, 0x0
3 mov ecx, 100
4 rep stos [rdi], al
```

---



# x86 string instructions: Direction Flag (DF) operation



- [x86 rep string instructions](#) to speed up streamed memory operations
- Default operate **left-to-right**, unless software sets *RFLAGS.DF=1*

---

```
1 /* memset(buf, 0x0, 100) */
2 for (int i=0; i < 100; i++)
3     buf[i] = 0x0;
```

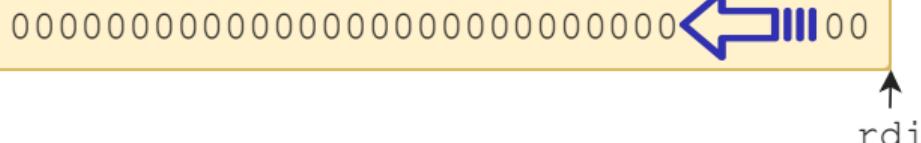
---



---

```
1 lea rdi, buf+100
2 mov al, 0x0
3 mov ecx, 100
4 std ; set direction flag
5 rep stos [rdi], al
```

---



# WHAT COULD POSSIBLY



# GO WRONG?

# SGX-DF: Inverting enclaved string memory operations

## x86 System-V ABI

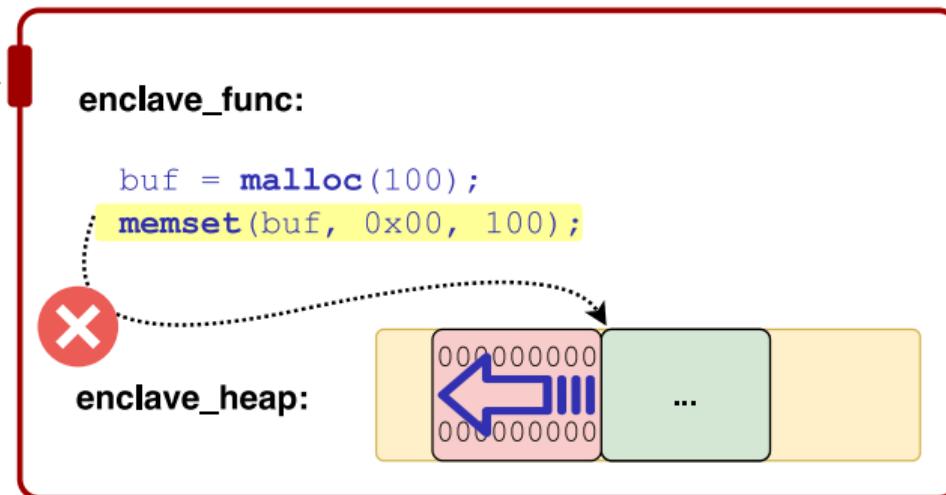
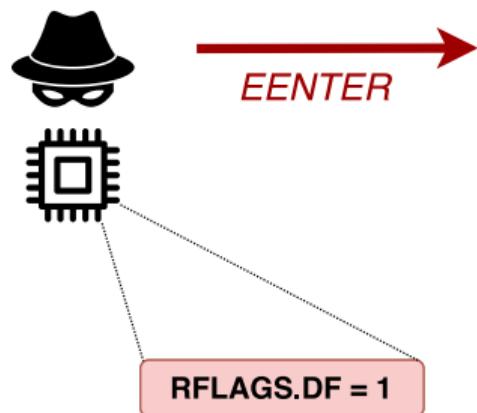


<sup>8</sup> The direction flag DF in the %rFLAGS register must be clear (set to “forward” direction) on function entry and return. Other user flags have no specified role in the standard calling sequence and are *not* preserved across calls.

# SGX-DF: Inverting enclaved string memory operations



Enclave heap **memory corruption**: right-to-left...



# Guardian: symbolic validation of orderliness in SGX enclaves

Pedro Antonino<sup>1</sup>, Wojciech Aleksander Wołoszyn<sup>1,2</sup>, and A. W. Roscoe<sup>1,3,4</sup>

<sup>1</sup> The Blockhouse Technology Limited, Oxford, UK

<sup>2</sup> Mathematical Institute, University of Oxford, Oxford, UK

<sup>3</sup> University College Oxford Blockchain Research Centre, Oxford, UK

<sup>4</sup> Department of Computer Science, University of Oxford, Oxford, UK

{pedro, wojciech}@tbtl.com, awroscoe@gmail.com



## Challenge 2: Constant-time code

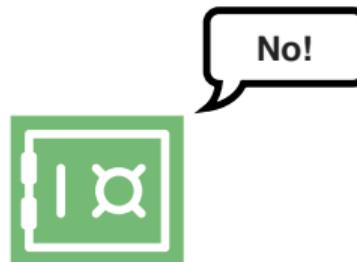
---

## Case study: Comparing a secret password

password

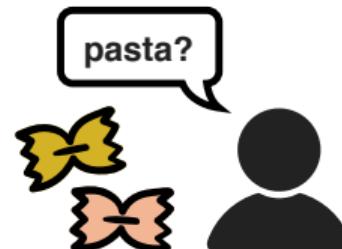
# Case study: Comparing a secret password

password

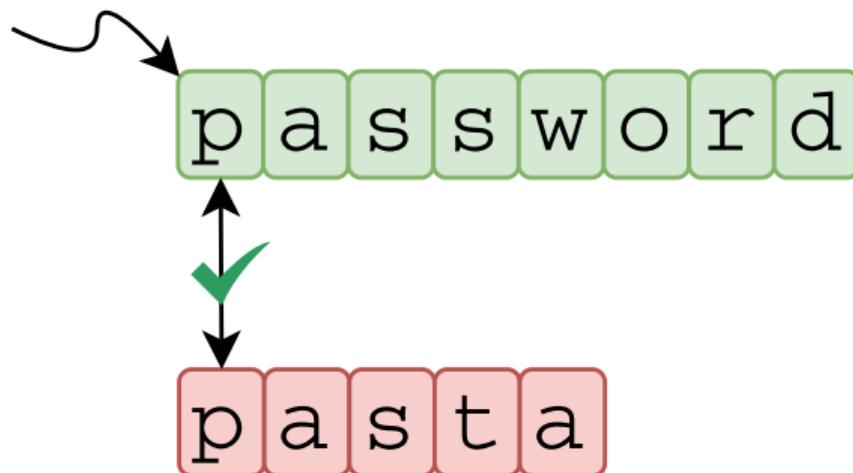


No!

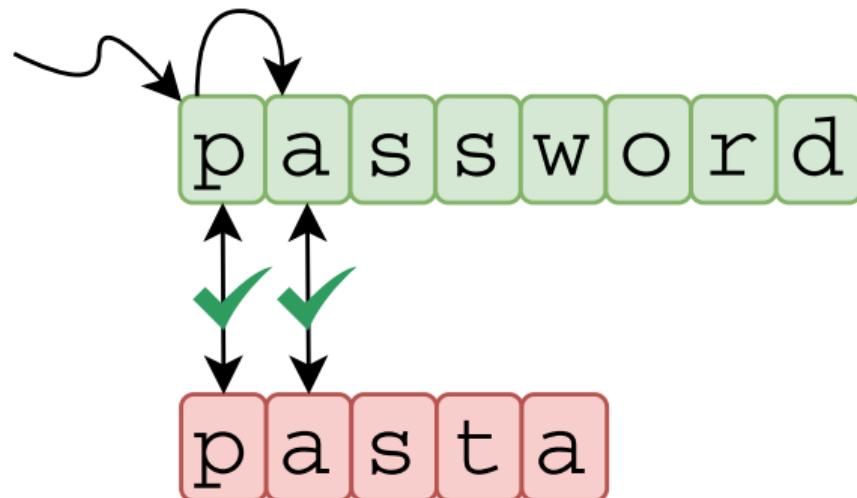
pasta



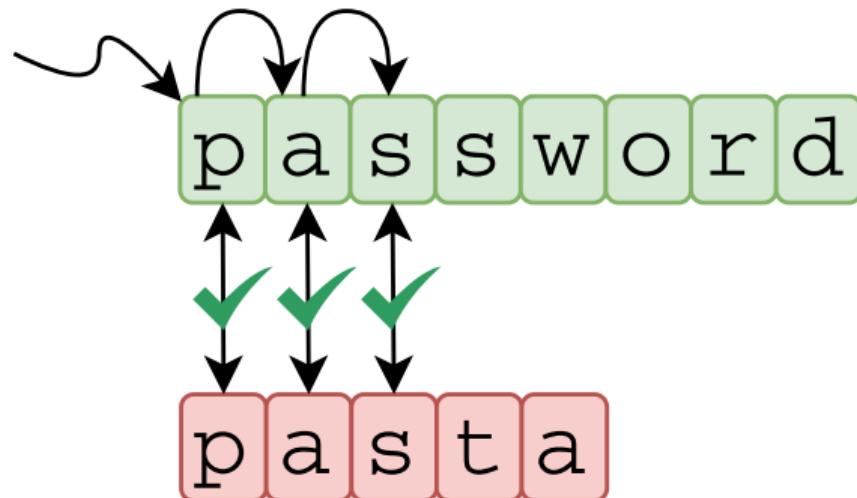
## Case study: Comparing a secret password



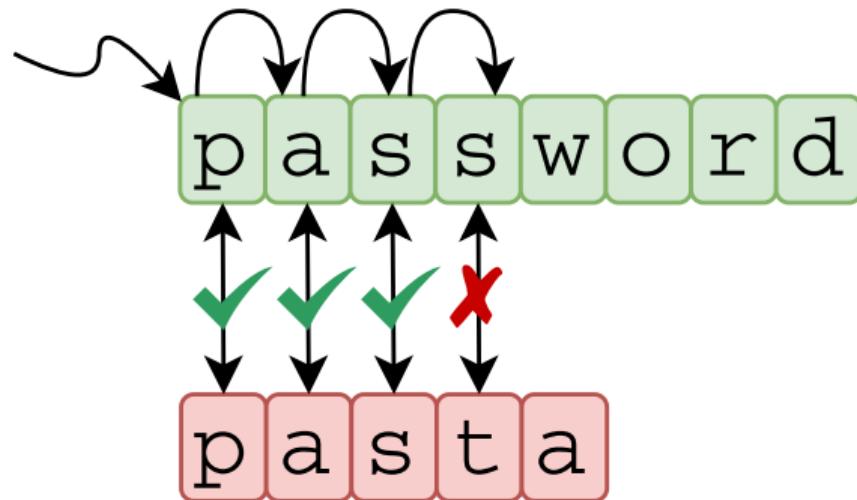
## Case study: Comparing a secret password



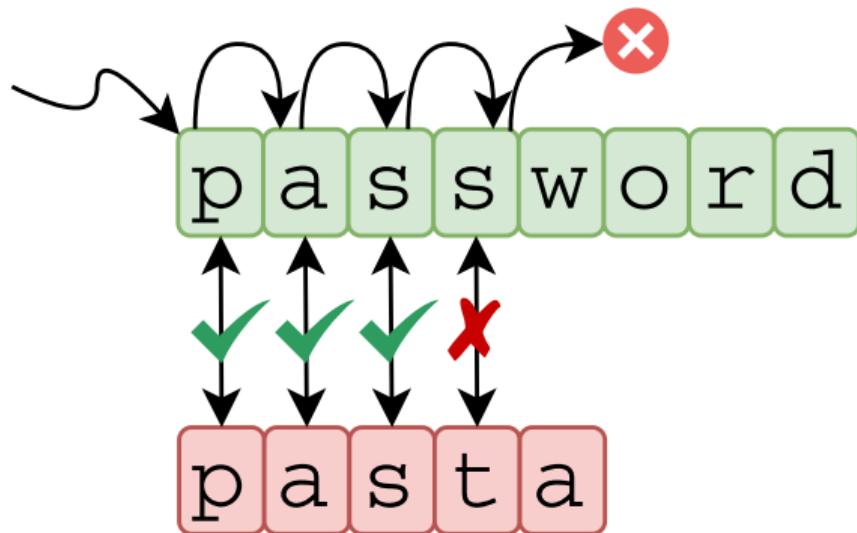
## Case study: Comparing a secret password



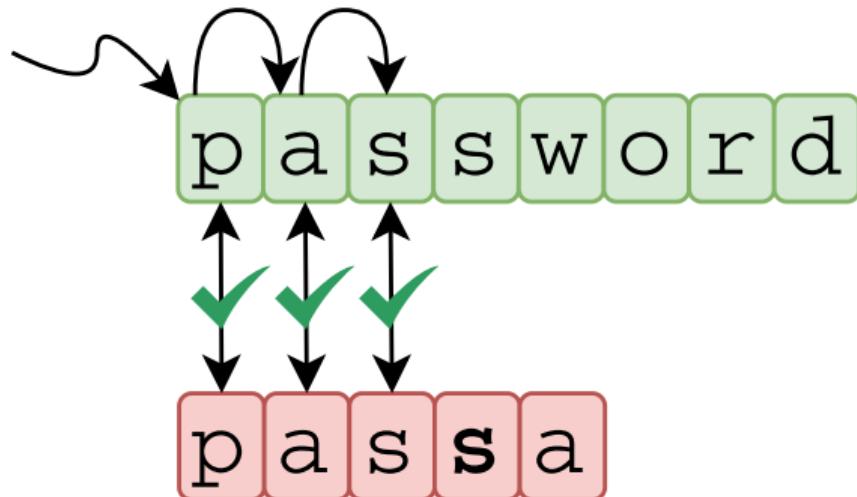
## Case study: Comparing a secret password



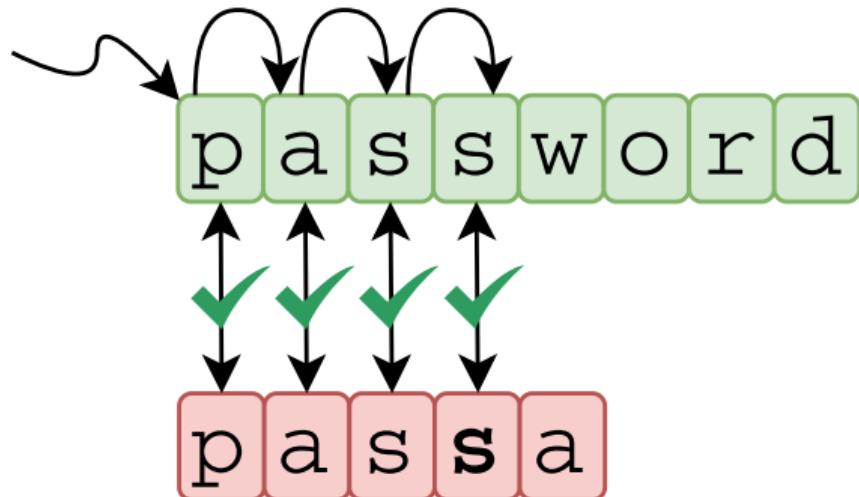
## Case study: Comparing a secret password



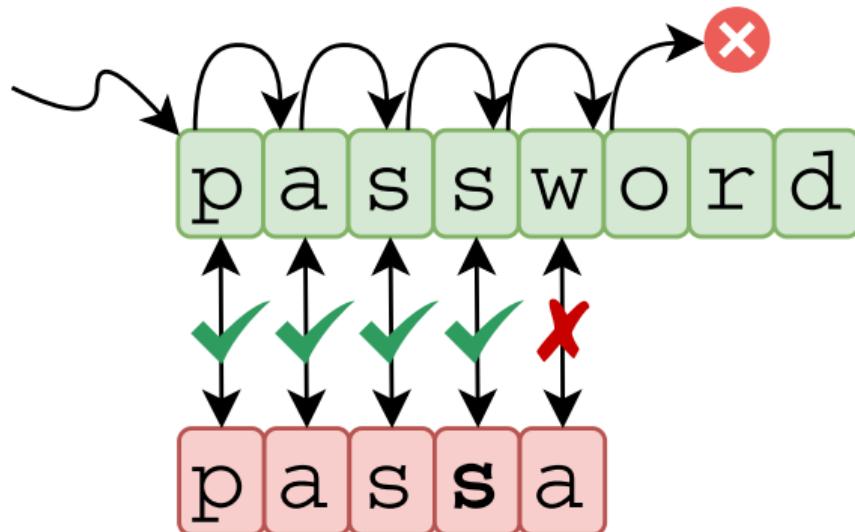
## Case study: Comparing a secret password



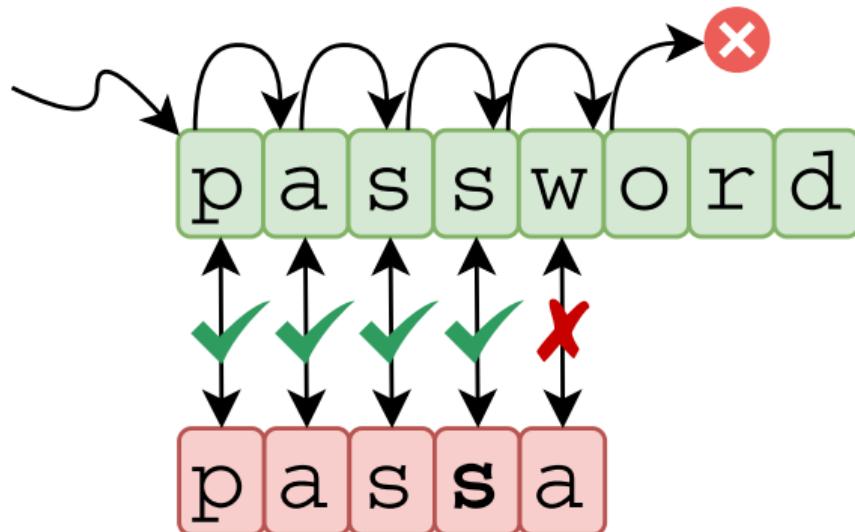
## Case study: Comparing a secret password



## Case study: Comparing a secret password

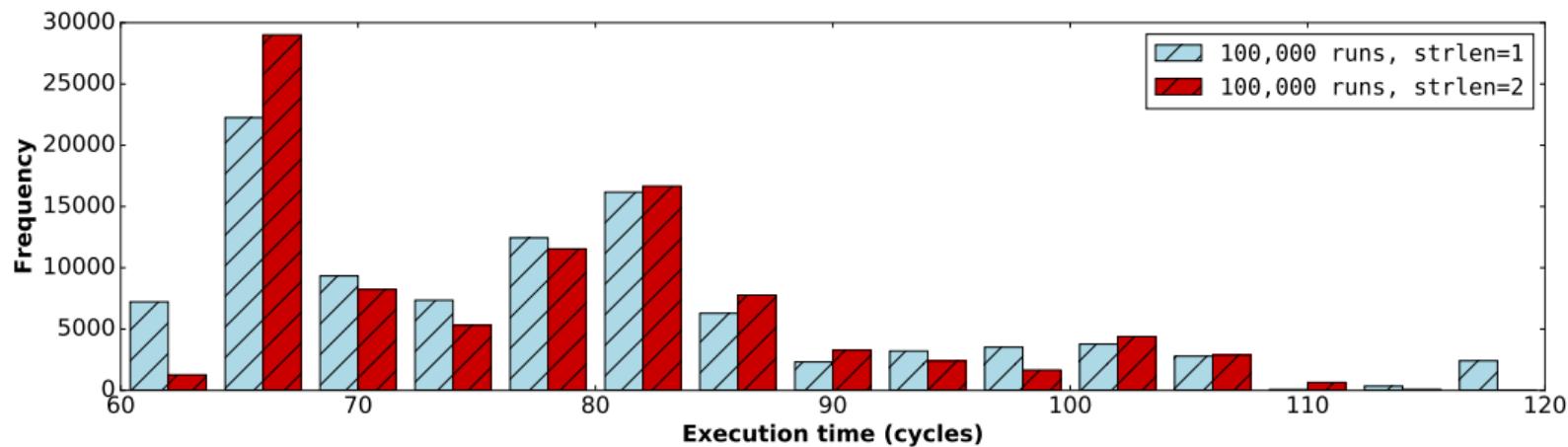


## Case study: Comparing a secret password



Overall **execution time** reveals correctness of individual password bytes!

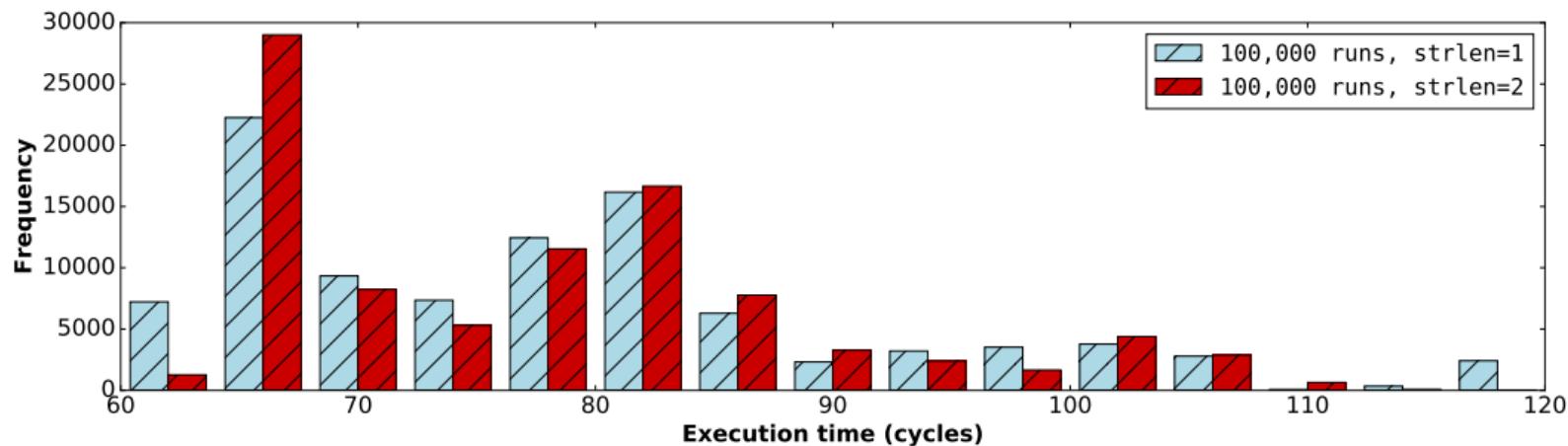
# Building the side-channel oracle with execution timing?



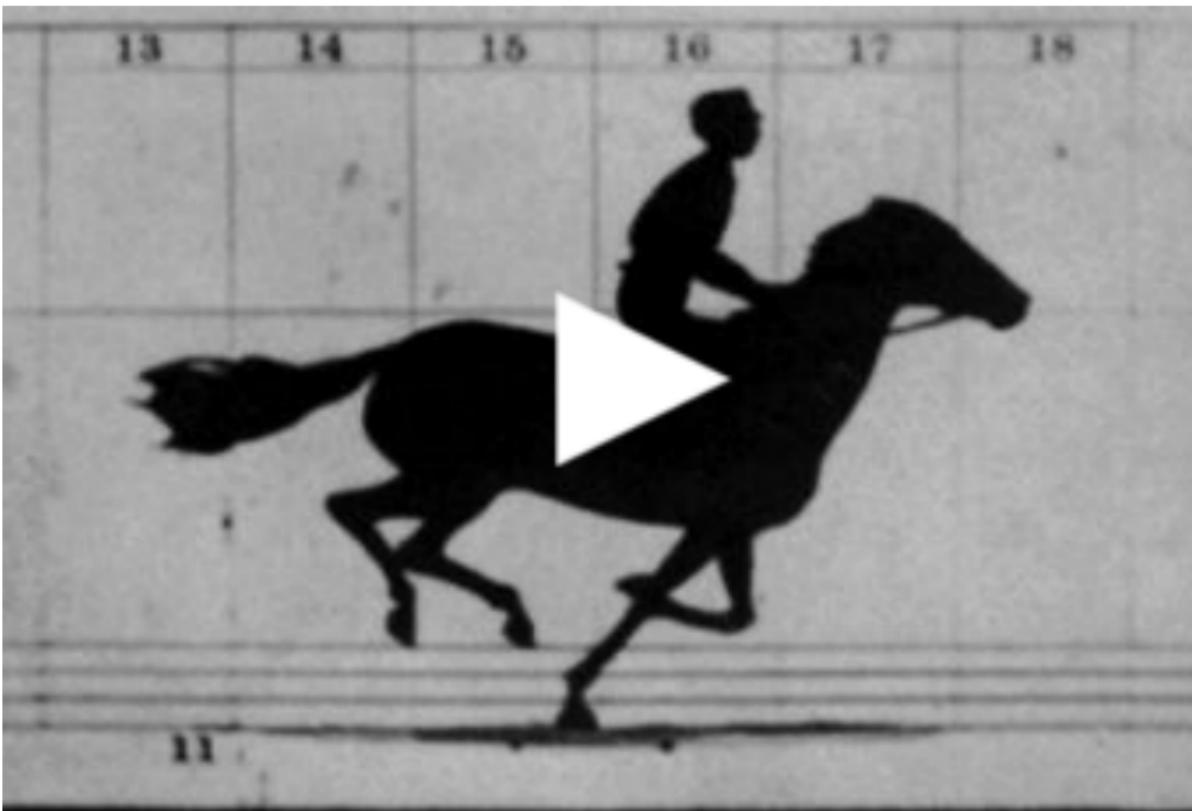
# Building the side-channel oracle with execution timing?



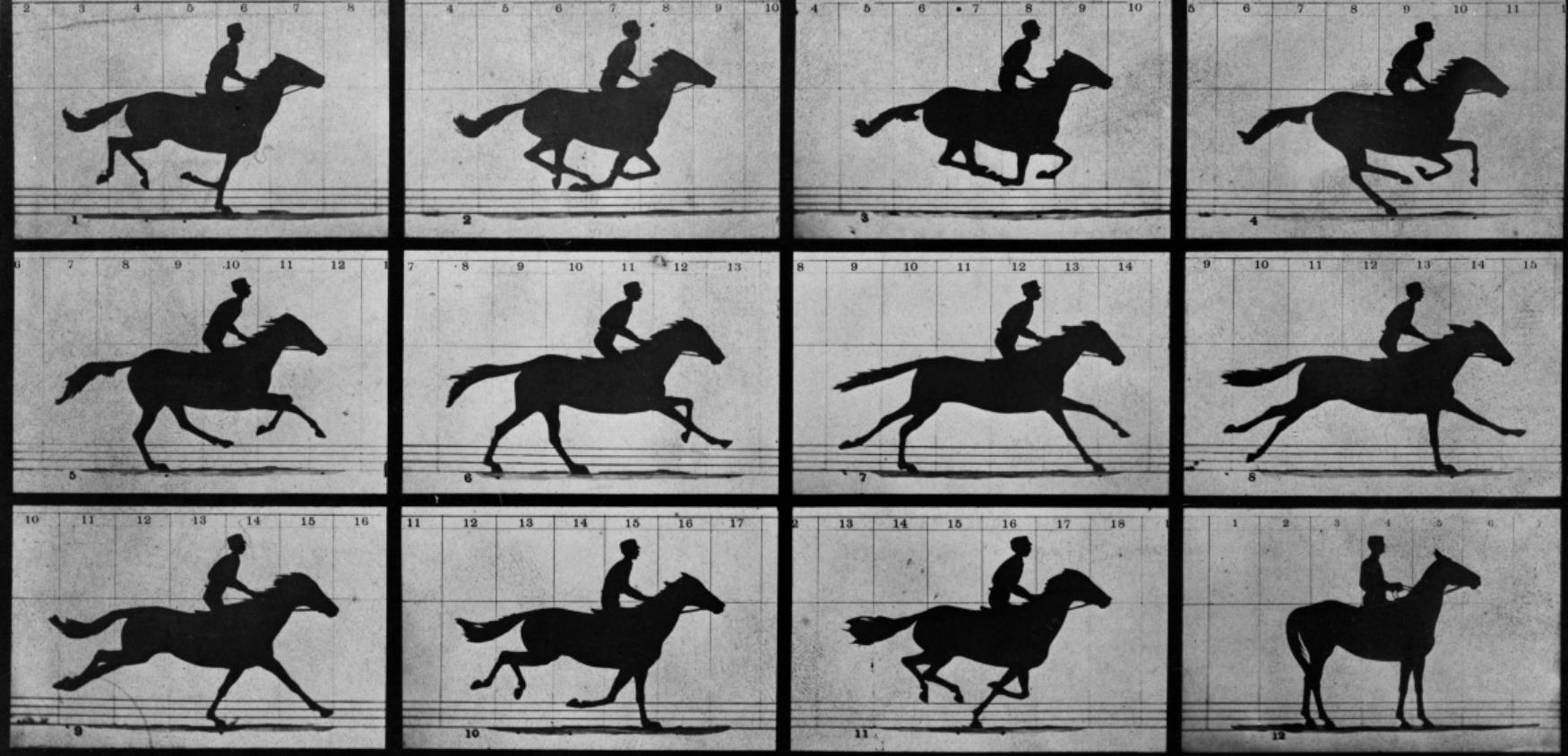
**Too noisy:** modern x86 processors are lightning fast...



## Analogy: Studying galloping horse dynamics



[https://en.wikipedia.org/wiki/Sallie\\_Gardner\\_at\\_a\\_Gallop](https://en.wikipedia.org/wiki/Sallie_Gardner_at_a_Gallop)



Copyright, 1878, by MUYBRIDGE.

MORSE'S Gallery, 417 Montgomery St., San Francisco.

## THE HORSE IN MOTION.

Illustrated by  
MUYBRIDGE

"SALLIE GARDNER," owned by LELAND STANFORD; running at a 1.40 gait over the Palo Alto track, 19th June, 1878.

The sequence of the photographs was taken at the rate of one exposure every  $\frac{1}{10}$  second, or 10 exposures per second. The 12 photographs, therefore, illustrate successive positions of the horse during a single step, or gait, at a uniform speed.

AUTOMATIC ELECTRO-PHOTOGRAPH.

## SGX-Step: Executing enclaves one instruction at a time



# SGX-Step



<https://github.com/jovanbulck/sgx-step>



Unwatch ▾

27



Star

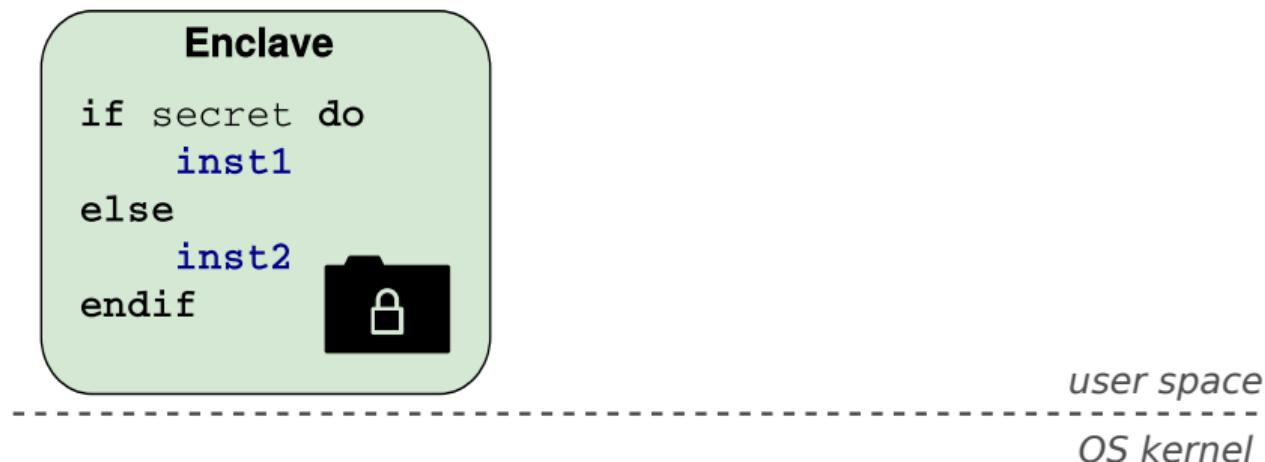
312



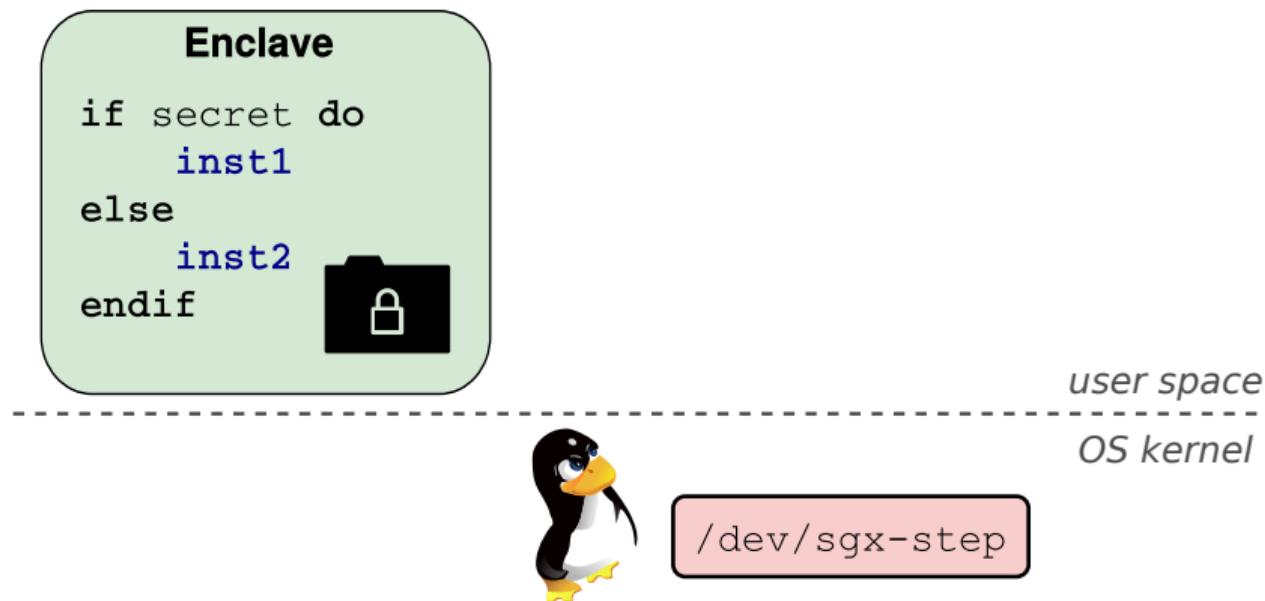
Fork

63

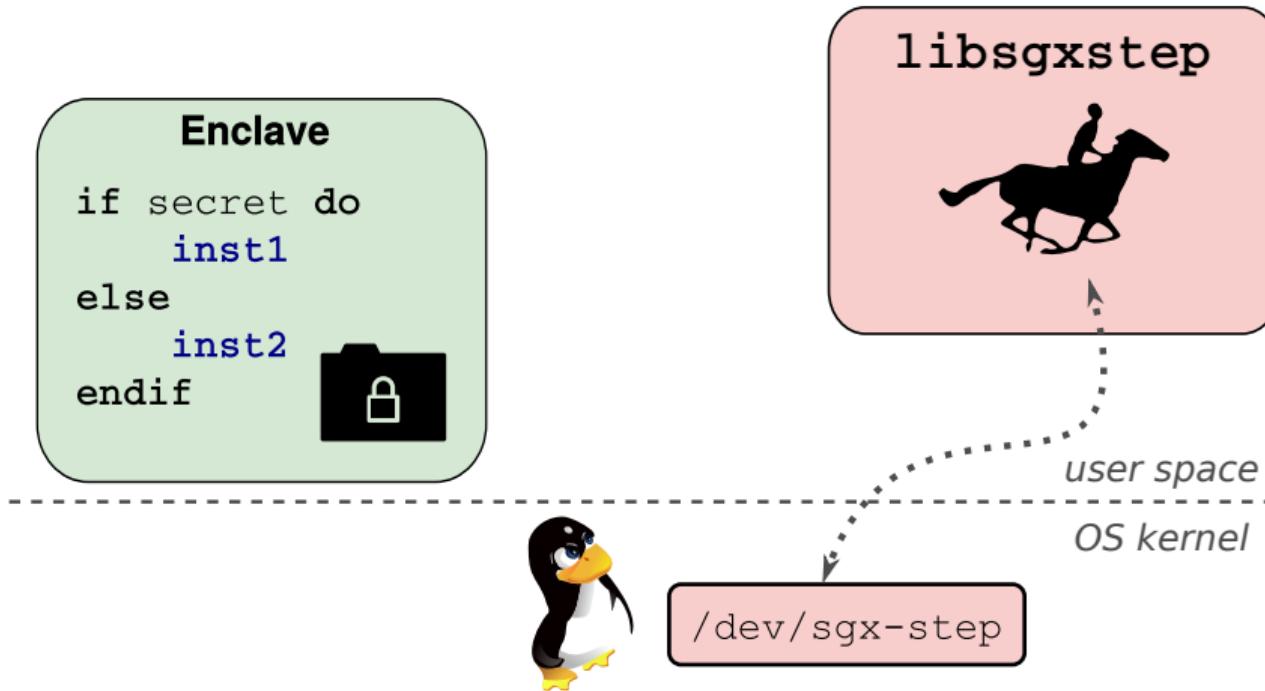
# SGX-Step: Executing enclaves one instruction at a time



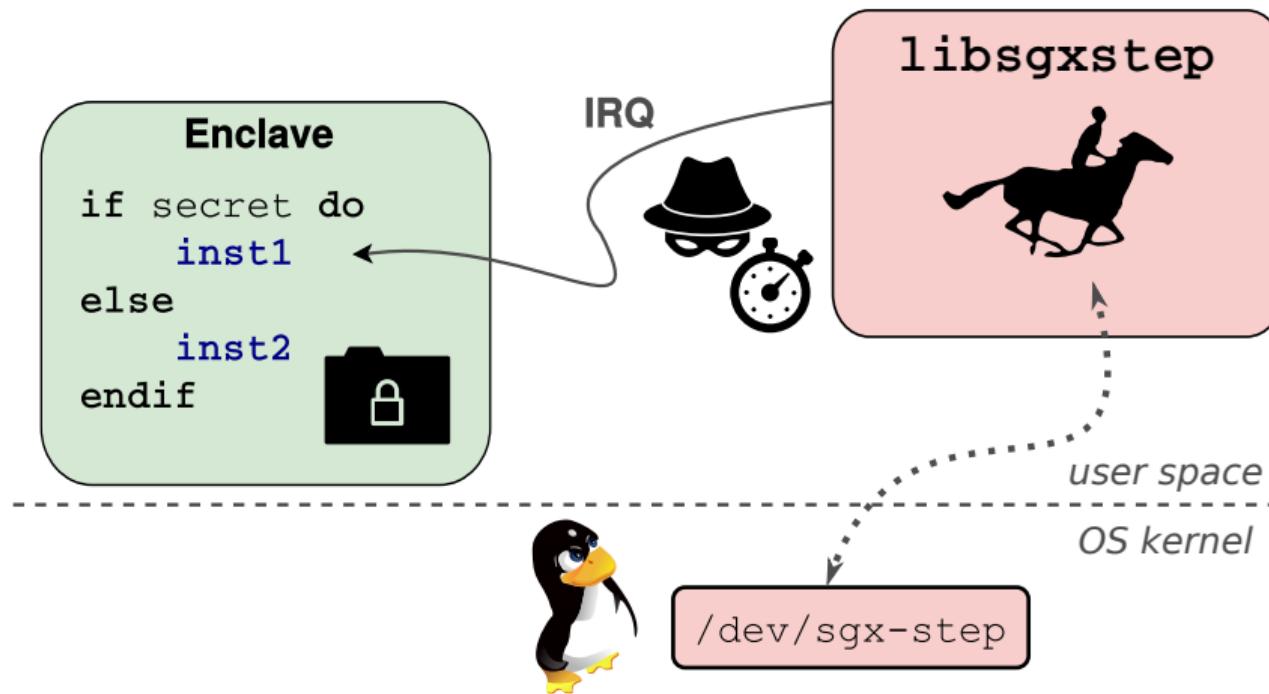
# SGX-Step: Executing enclaves one instruction at a time



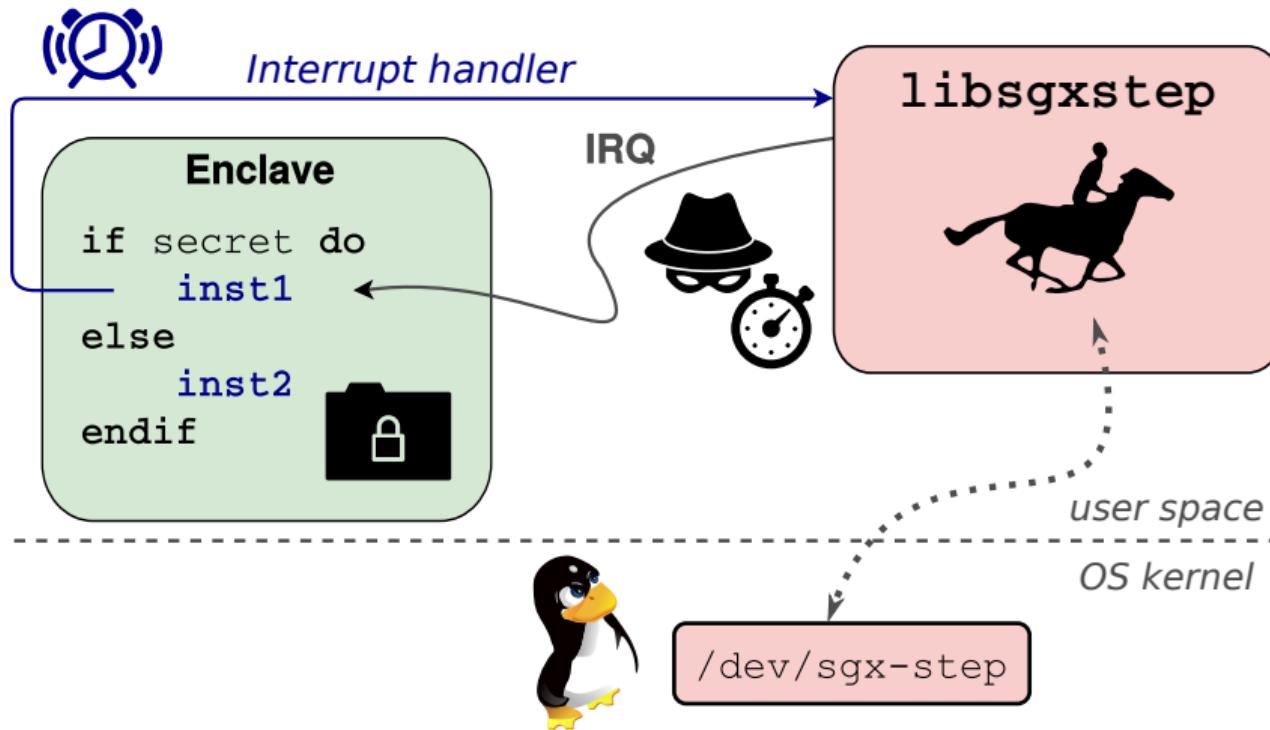
# SGX-Step: Executing enclaves one instruction at a time



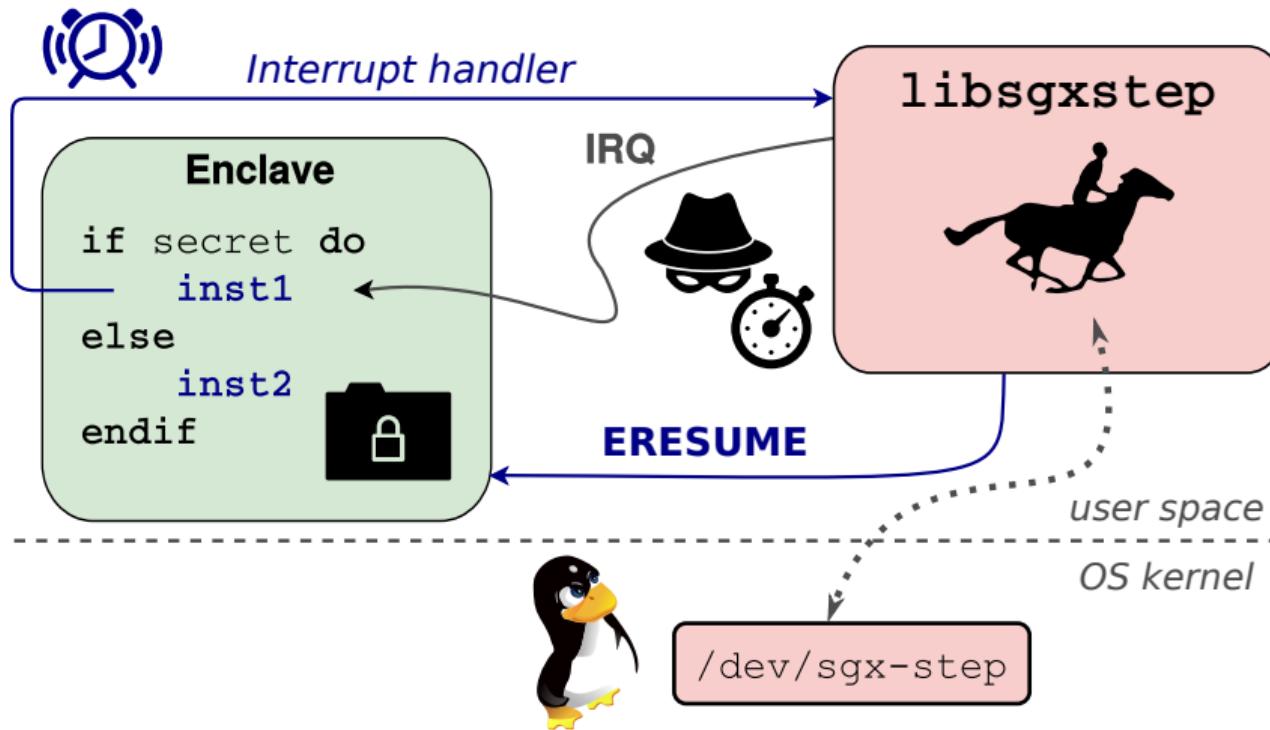
# SGX-Step: Executing enclaves one instruction at a time



# SGX-Step: Executing enclaves one instruction at a time



# SGX-Step: Executing enclaves one instruction at a time



# Demo: Building a deterministic password oracle with SGX-Step

```
[idt.c] DTR.base=0xfffffe0000000000/size=4095 (256 entries)
[idt.c] established user space IDT mapping at 0x7f7ff8e9a000
[idt.c] installed asm IRQ handler at 10:0x56312d19b000
[idt.c] IDT[ 45] @0x7f7ff8e9a2d0 = 0x56312d19b000 (seg sel 0x10); p=1; dpl=3; type=14; ist=0
[file.c] reading buffer from '/dev/cpu/1/msr' (size=8)
[apic.c] established local memory mapping for APIC_BASE=0xfee00000 at 0x7f7ff8e99000
[apic.c] APIC_ID=2000000; LVTT=400ec; TDCR=0
[apic.c] APIC timer one-shot mode with division 2 (lvtt=2d/tocr=0)
```

```
-----  
[main.c] recovering password length  
-----
```

```
[attacker] steps=15; guess='*****'  
[attacker] found pwd len = 6
```

```
-----  
[main.c] recovering password bytes  
-----
```

```
[attacker] steps=35; guess='SECRET' --> SUCCESS
```

```
[apic.c] Restored APIC_LVTT=400ec/TDCR=0
[file.c] writing buffer to '/dev/cpu/1/msr' (size=8)
[main.c] all done; counted 2260/2183 IRQs (AEP/IDT)
jo@breuer:~/sgx-step-demo$ █
```

# **ALL YOUR PASSWORDS**



# **ARE BELONG TO US**

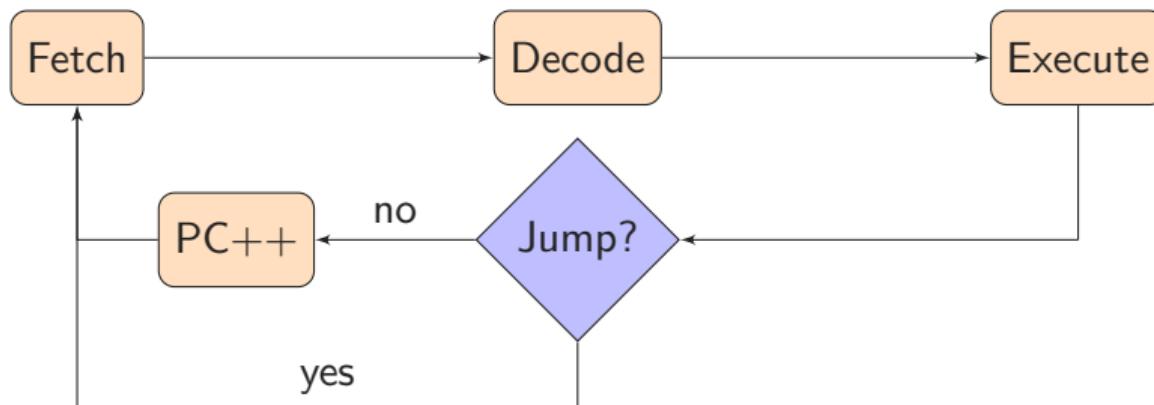
# SGX-Step: Enabling a new line of high-precision enclave attacks

Yr	Attack	Temporal resolution	APIC		PTE		Desc		Drv
			IRQ	IPI	#PF	A/D	PPN	GDT	
'15	Ctrl channel	~ Page	○	○	●	○	○	○	● ✓
'16	AsyncShock	~ Page	○	○	●	○	○	○	- ⚡
'17	CacheZoom	X > 1	●	○	○	○	○	○	✓ ⚡
'17	Hahnel et al.	X 0 - > 1	●	○	○	○	○	●	✓
'17	BranchShadow	X 5 - 50	●	○	○	○	○	○	✗ ⚡
'17	Stealthy PTE	~ Page	○	●	○	●	○	●	✓ ⚡
'17	DarkROP	~ Page	○	○	●	○	○	○	✓ ⚡
'17	SGX-Step	✓ 0 - 1	●	○	●	●	○	○	✓ ⚡
'18	Off-limits	✓ 0 - 1	●	○	●	○	○	●	✓ ⚡
'18	Single-trace RSA	~ Page	○	○	●	○	○	○	✓ ⚡
'18	Foreshadow	✓ 0 - 1	●	○	●	○	●	○	✓ ⚡
'18	SgxPectre	~ Page	○	○	●	○	○	○	✓ ⚡
'18	CacheQuote	X > 1	●	○	○	○	○	○	✓ ⚡
'18	SGXlinger	X > 1	●	○	○	○	○	○	✗ ⚡
'18	Nemesis	✓ 1	●	○	●	●	○	●	✓ ⚡

Yr	Attack	Temporal resolution	APIC		PTE		Desc		Drv
			IRQ	IPI	#PF	A/D	PPN	GDT	
'19	Spoiler	✓ 1	●	○	○	●	○	○	● ✓ ⚡
'19	ZombieLoad	✓ 0 - 1	●	○	●	●	○	○	● ✓ ⚡
'19	Tale of 2 worlds	✓ 1	●	○	●	●	○	○	● ✓ ⚡
'19	MicroScope	~ 0 - Page	○	○	●	○	○	○	✗ ⚡
'20	Bluethunder	✓ 1	●	○	○	○	○	○	● ✓ ⚡
'20	Big troubles	~ Page	○	○	●	○	○	○	○ ✓ ⚡
'20	Viral primitive	✓ 1	●	○	●	●	○	○	● ✓ ⚡
'20	CopyCat	✓ 1	●	○	●	●	○	○	● ✓ ⚡
'20	LVI	✓ 1	●	○	●	●	●	○	● ✓ ⚡
'20	A to Z	~ Page	○	○	●	○	○	○	○ ✓ ⚡
'20	Frontal	✓ 1	●	○	●	●	○	○	● ✓ ⚡
'20	CrossTalk	✓ 1	●	○	●	○	○	○	● ✓ ⚡
'20	Online template	~ Page	○	○	●	○	○	○	○ ✓ ⚡
'20	Déjà Vu NSS	~ Page	○	○	●	○	○	○	○ ✓ ⚡

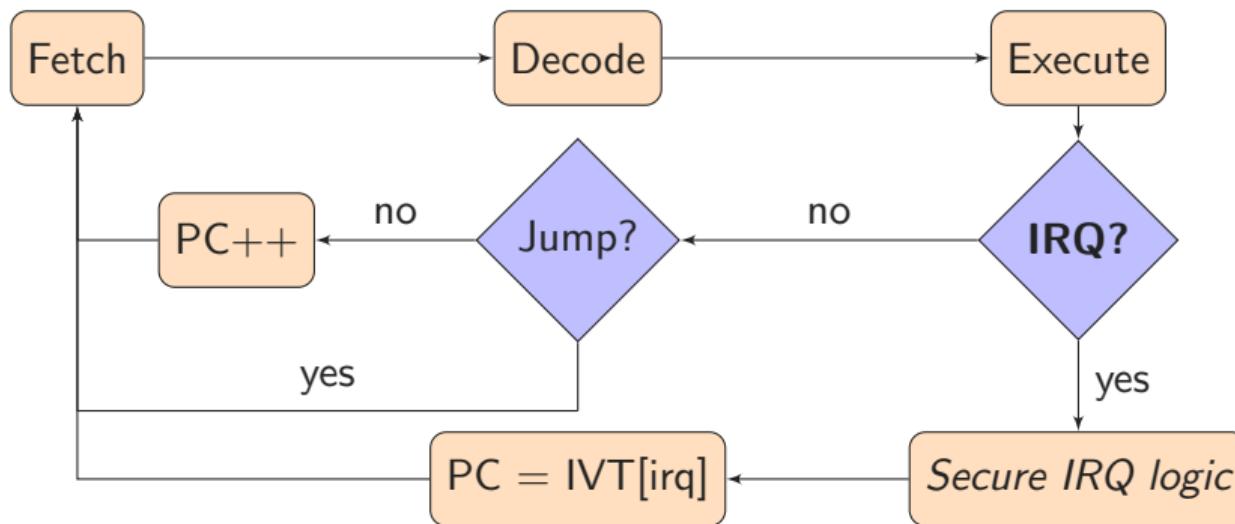
# Back to basics: Fetch-decode-execute

**Elementary CPU behavior:** Stored program computer



# Back to basics: Fetch-decode-execute

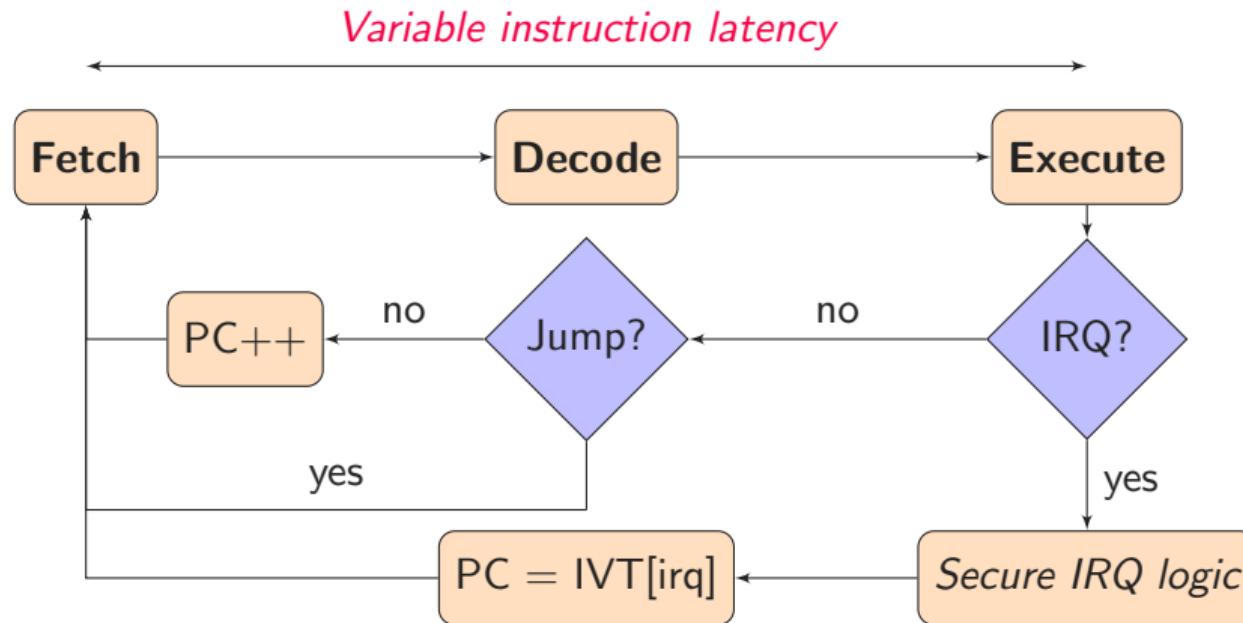
**Interrupts:** Asynchronous events, handled on instruction retirement



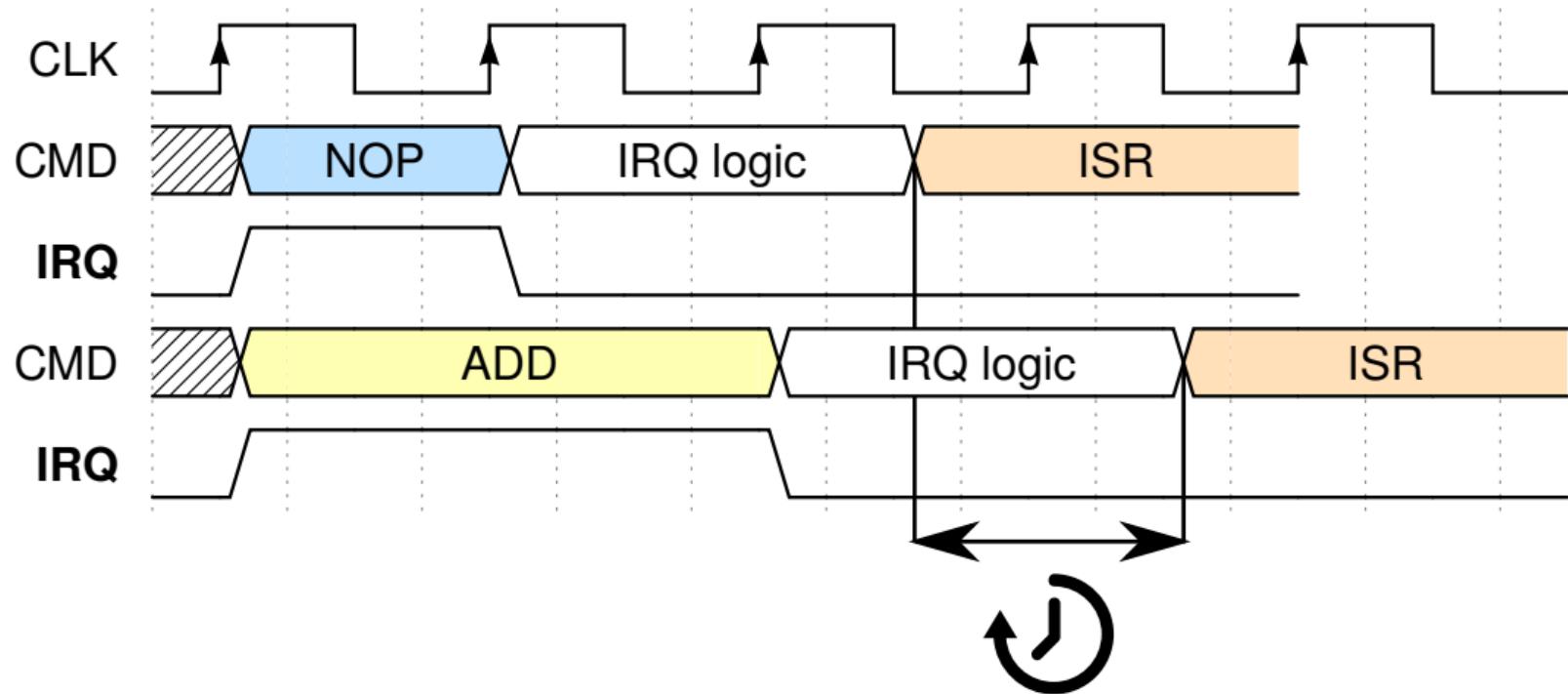
# Back to basics: Fetch-decode-execute



**Timing leak:** IRQ response time depends on current instruction(!)



# Wait a cycle: Interrupt latency as a side channel

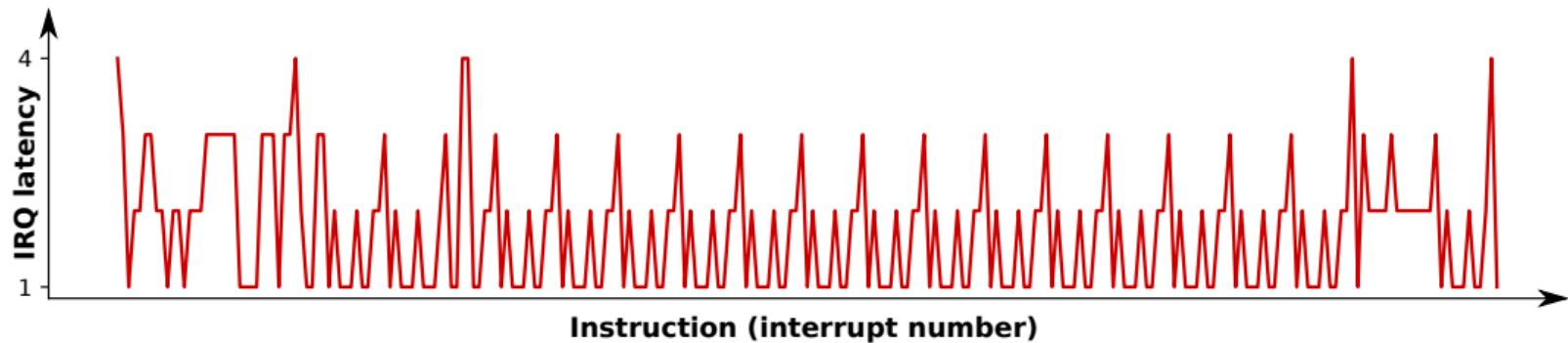


# TIMING LEAKS



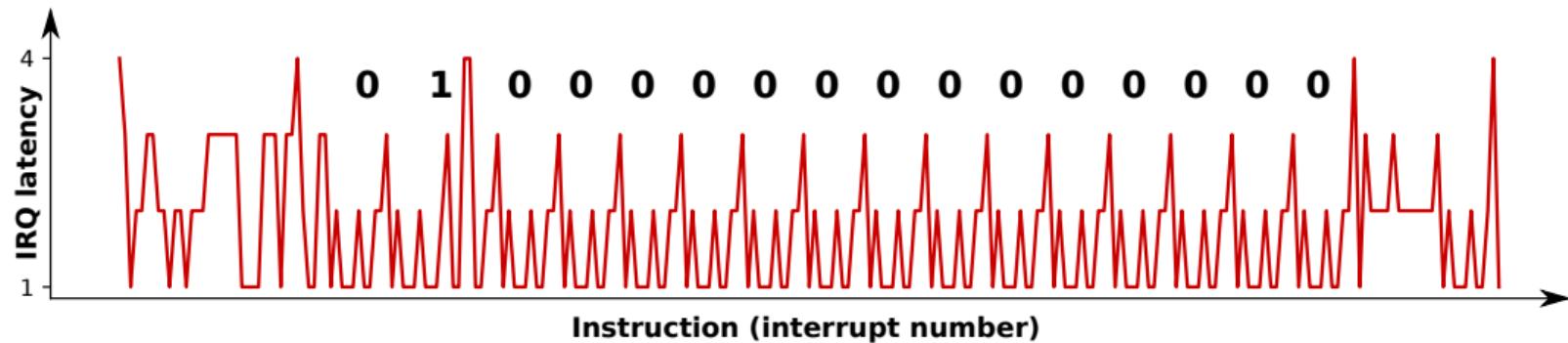
# EVERYWHERE

# Nemesis attack: Inferring key strokes from Sancus enclaves



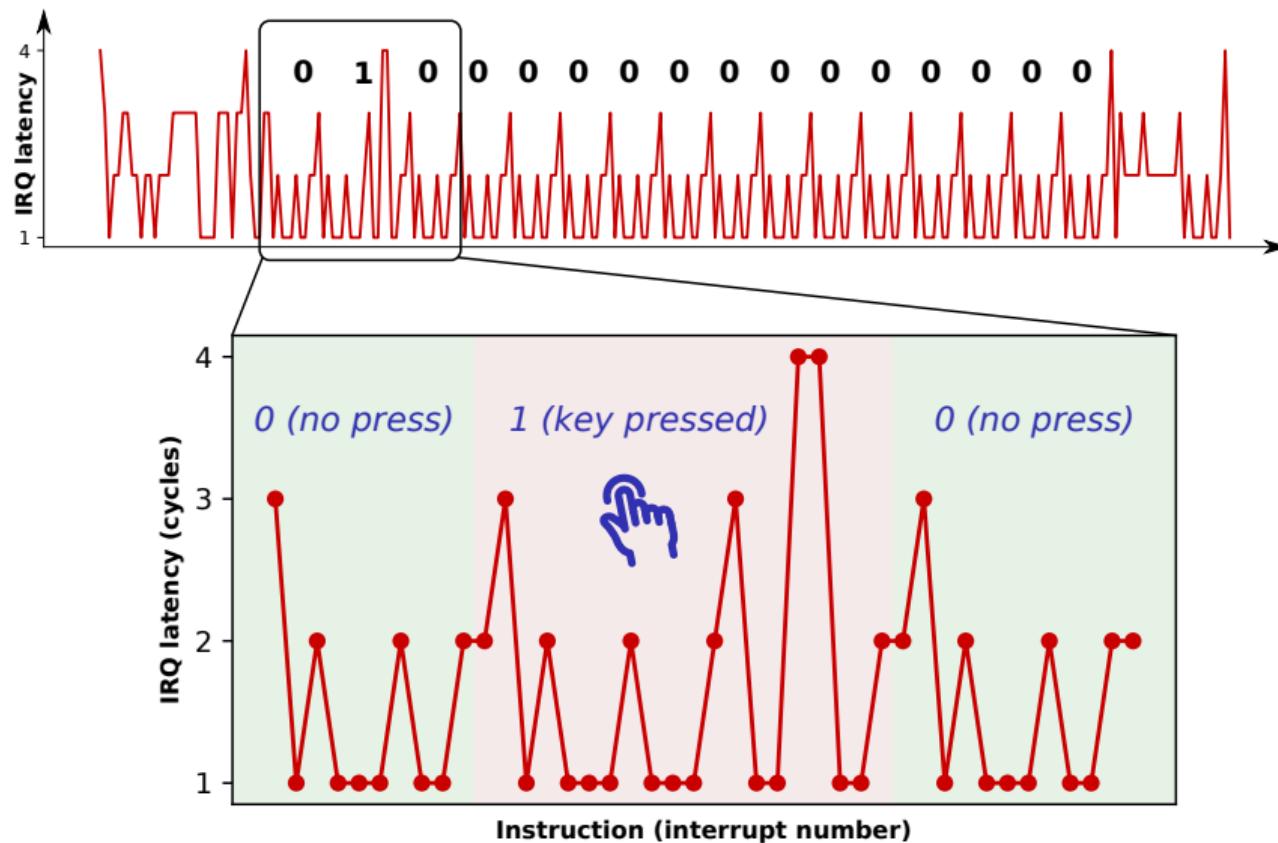
**Enclave x-ray:** Start-to-end trace enclaved execution

## Nemesis attack: Inferring key strokes from Sancus enclaves



## **Enclave x-ray:** Keymap bit traversal (ground truth)

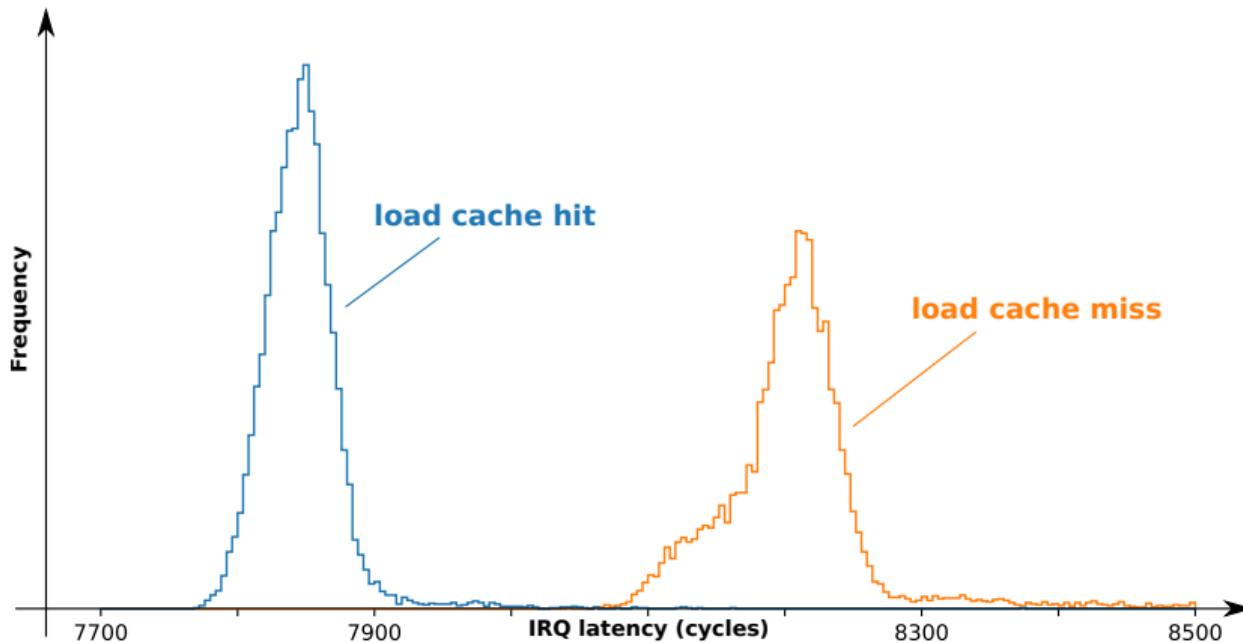
# Nemesis attack: Inferring key strokes from Sancus enclaves



# Intel SGX microbenchmarks: Measuring x86 cache misses



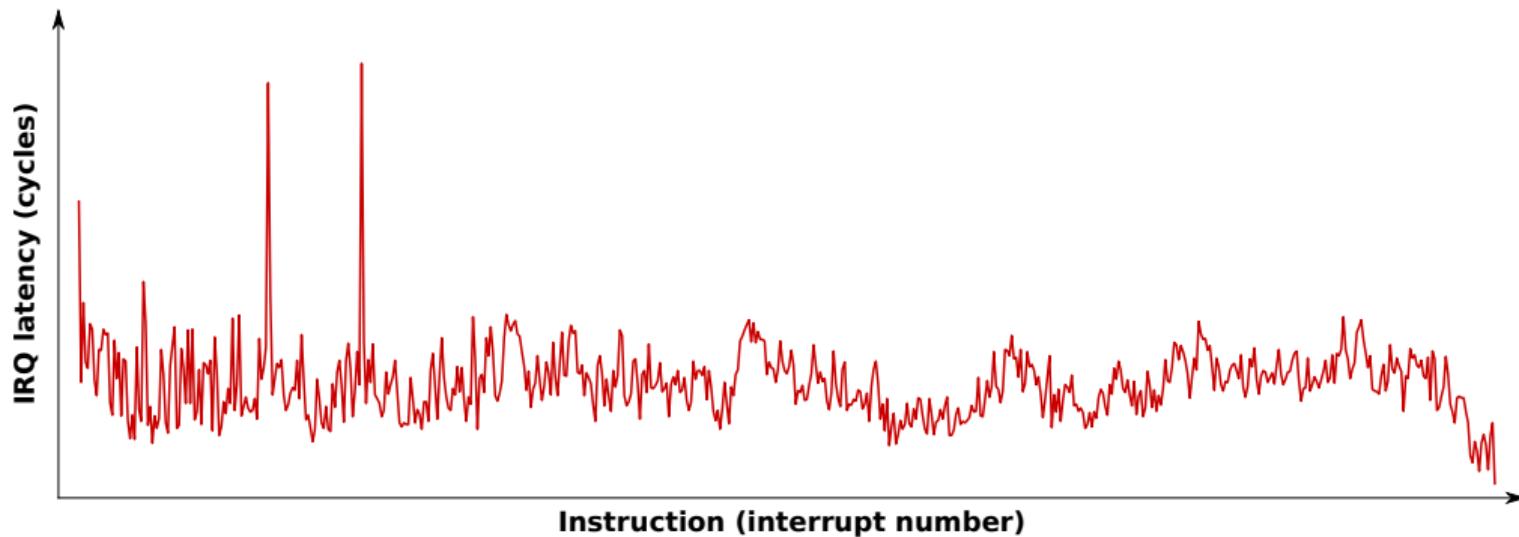
Timing leak: reconstruct *microarchitectural state*



# Single-stepping Intel SGX enclaves in practice



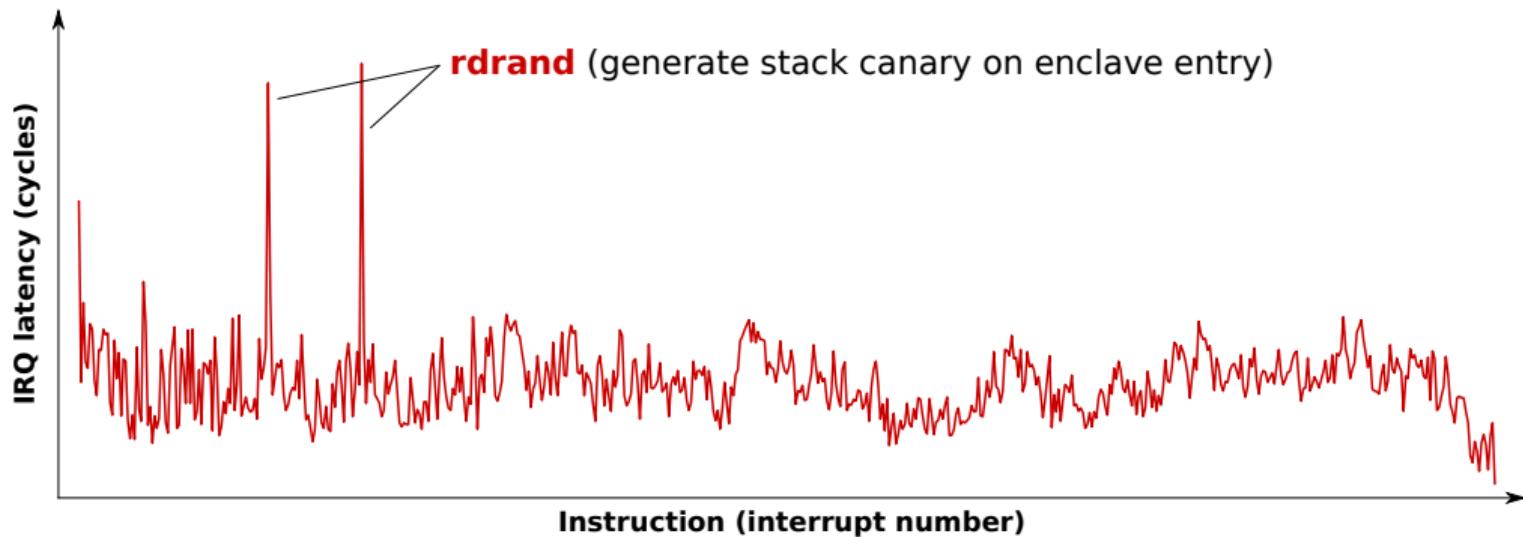
**Enclave x-ray:** Start-to-end trace enclaved execution



# Single-stepping Intel SGX enclaves in practice



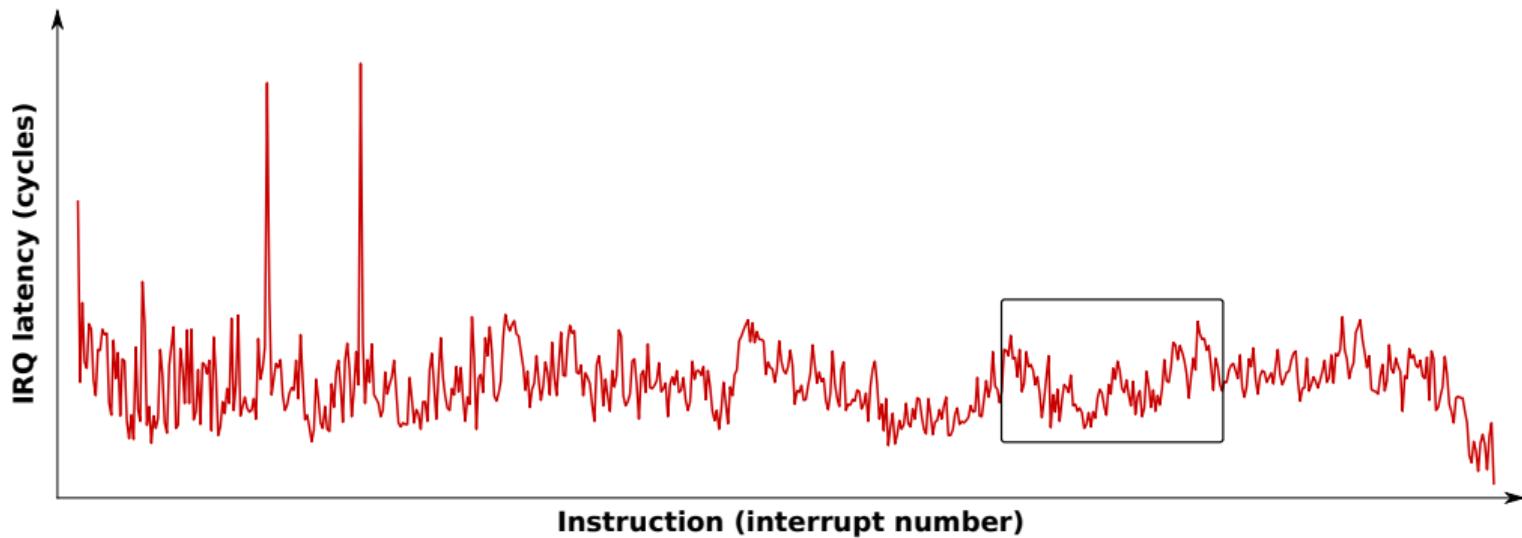
**Enclave x-ray:** Spotting **high-latency** instructions



# Single-stepping Intel SGX enclaves in practice

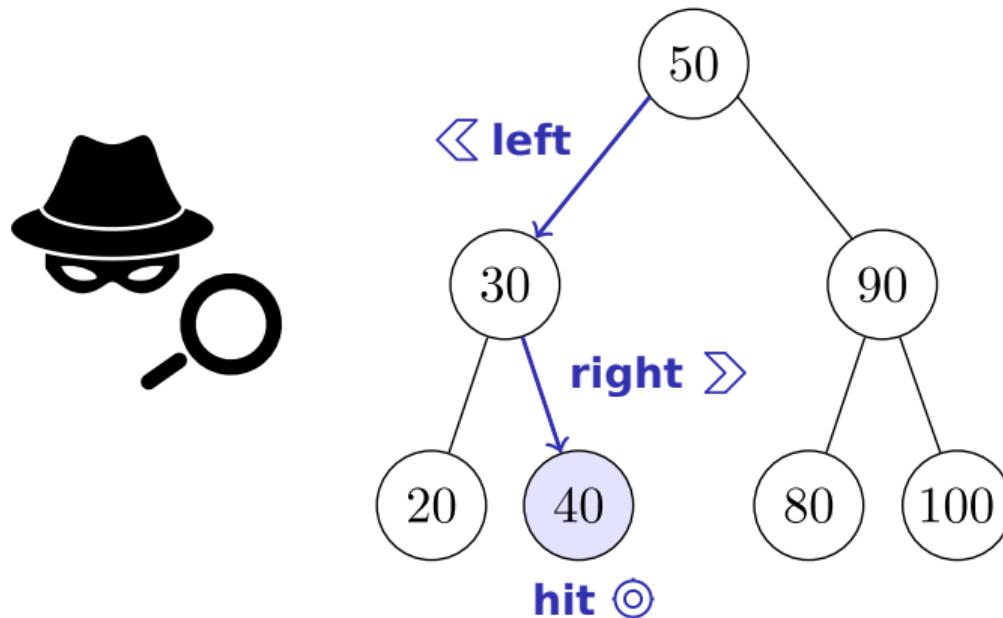


**Enclave x-ray:** Zooming in on `bsearch` function



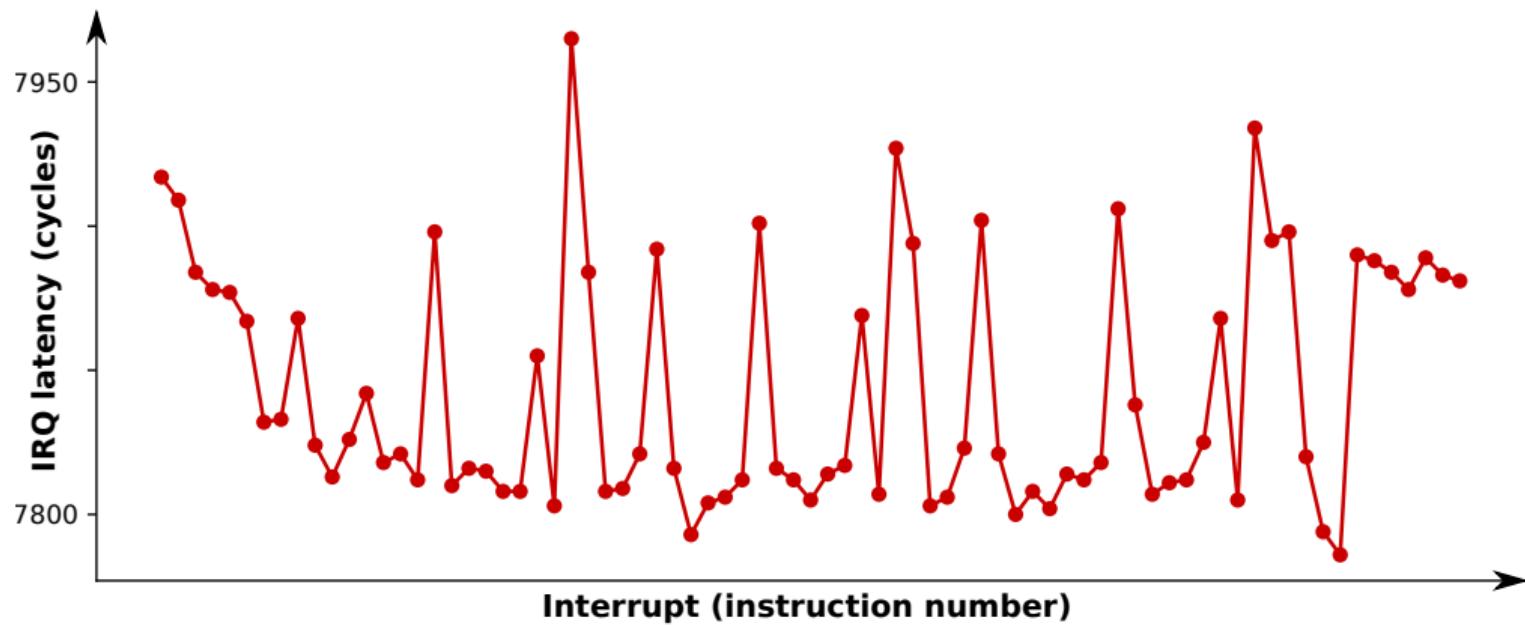
# De-anonymizing SGX enclave lookups with interrupt latency

**Adversary:** Infer **secret lookup** in known sequence (e.g., DNA)



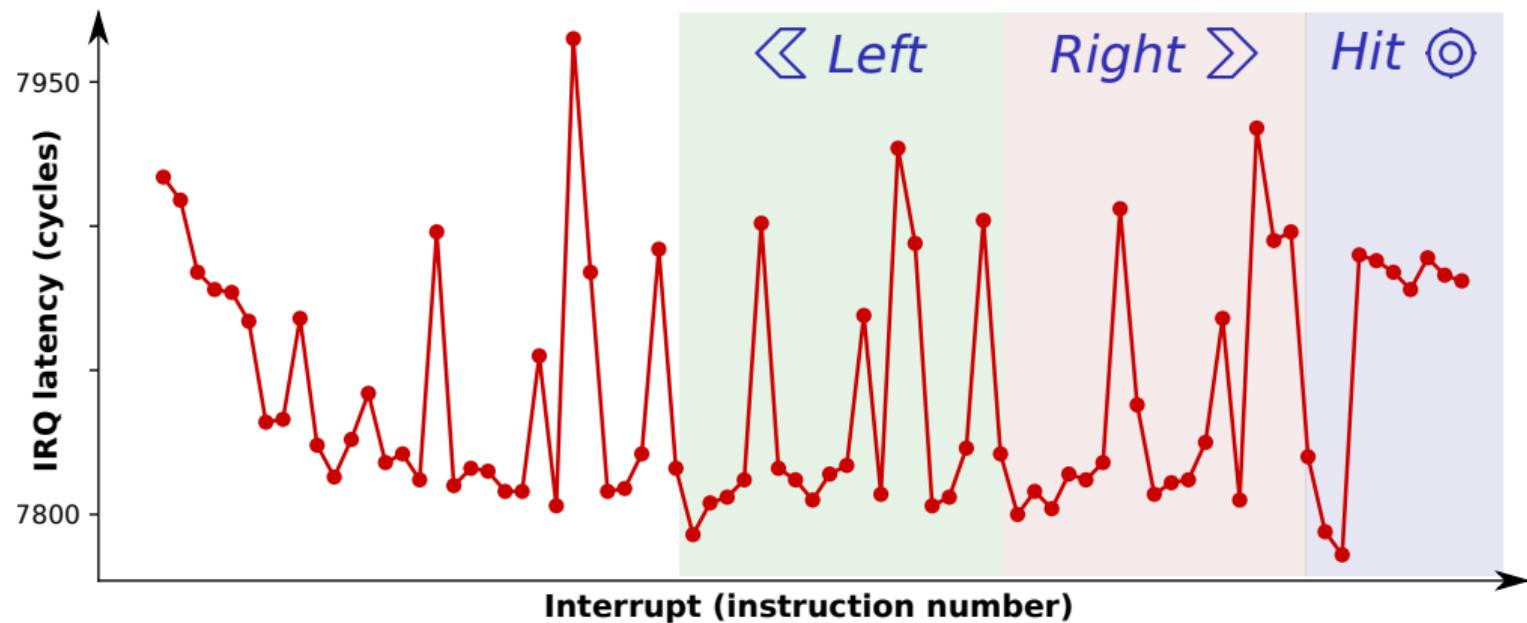
# De-anonymizing SGX enclave lookups with interrupt latency

**Goal:** Infer lookup → reconstruct `bsearch` control flow

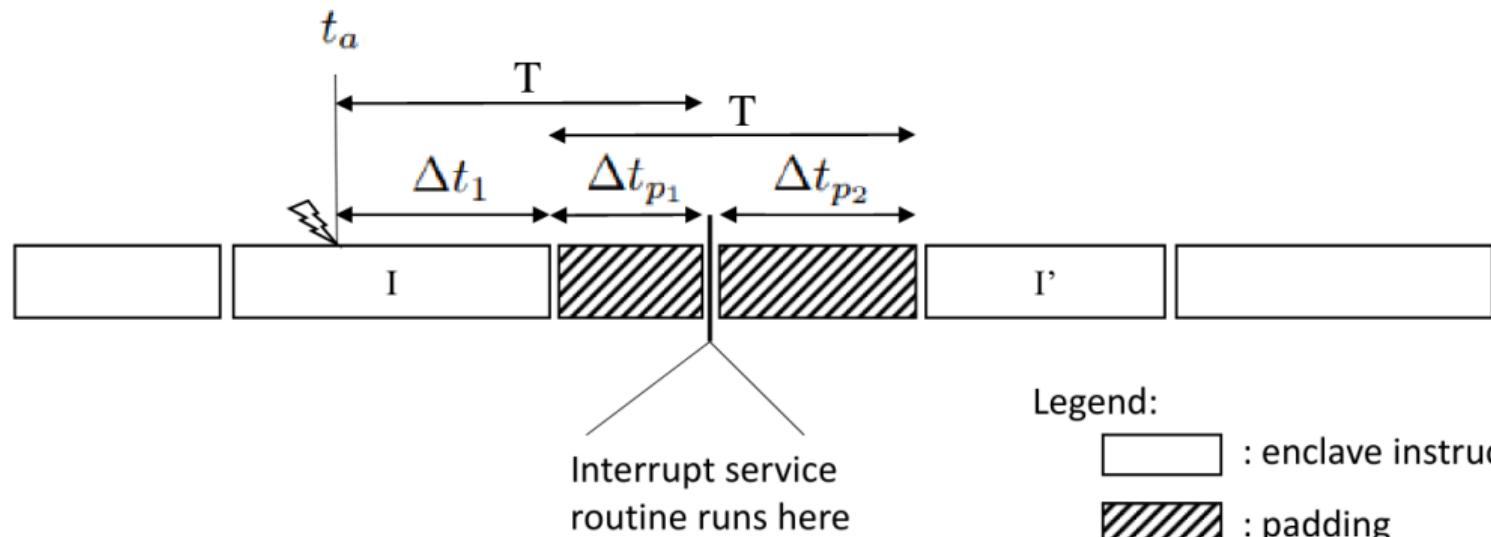


# De-anonymizing SGX enclave lookups with interrupt latency

**Goal:** Infer lookup → reconstruct `bsearch` control flow

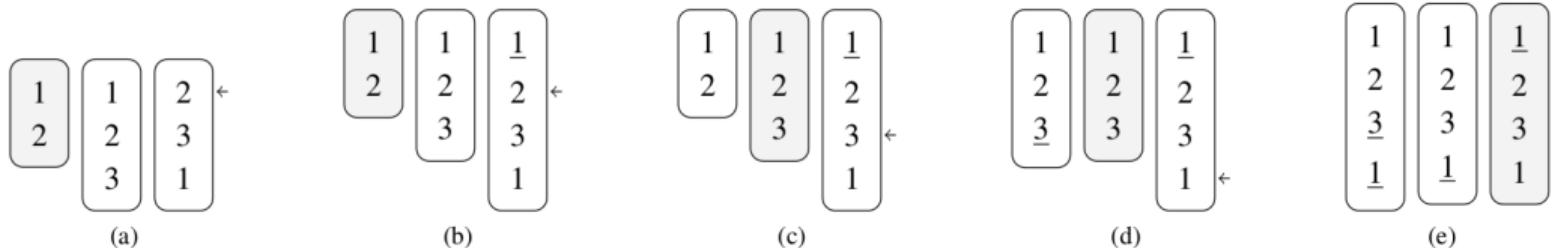


# Nemesis hardware defense: Padding interrupt latency



- Busi et al. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors", CSF 2020.

# Nemesis software defenses: Balancing vulnerable branches



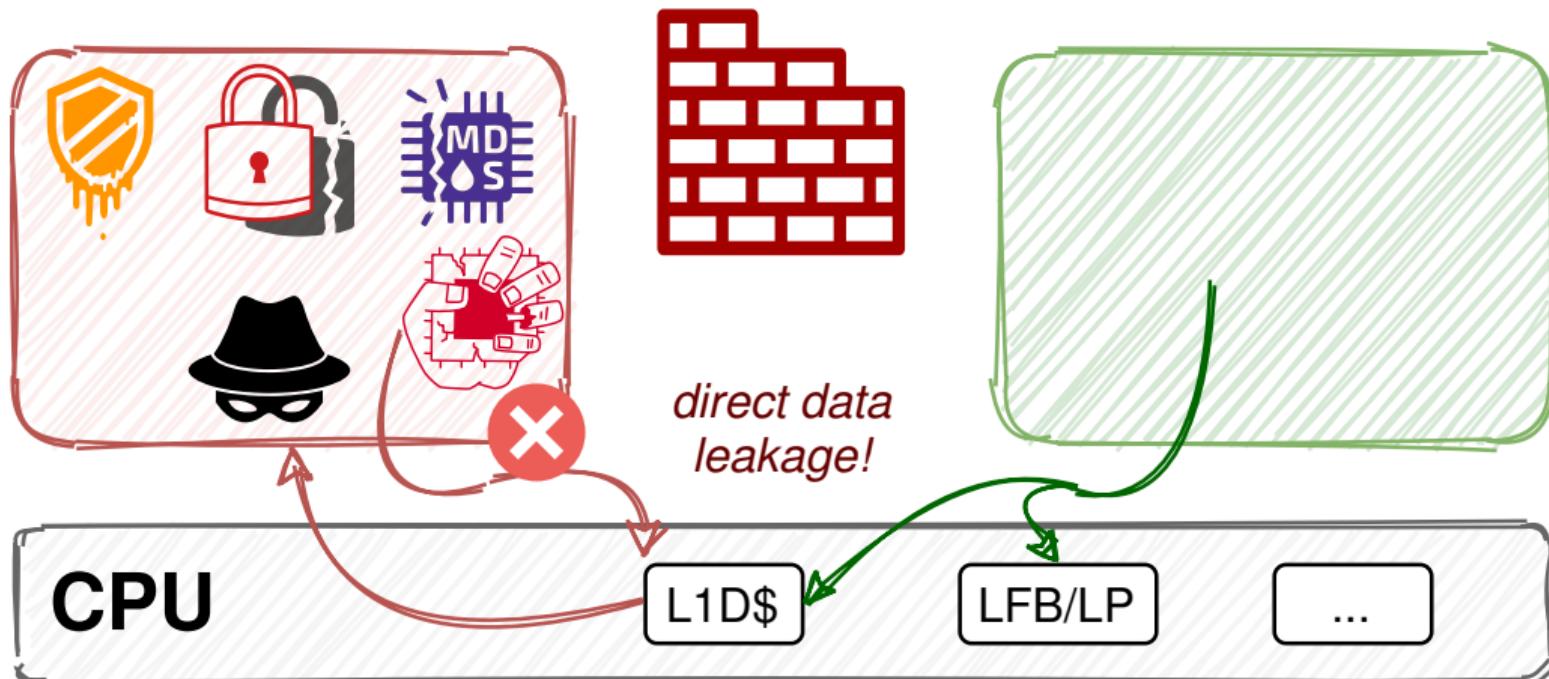
- Busi et al. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors", CSF 2020.
- Winderix et al. "Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks", EuroS&P 2021.
- Pouyanrad et al. "SCFMSP: Static detection of side channels in MSP430 programs", ARES 2020.
- Salehi et al. "NemesisGuard: Mitigating interrupt latency side channel attacks with static binary rewriting", Computer Networks 2022.



## Challenge 3: Fencing transient loads

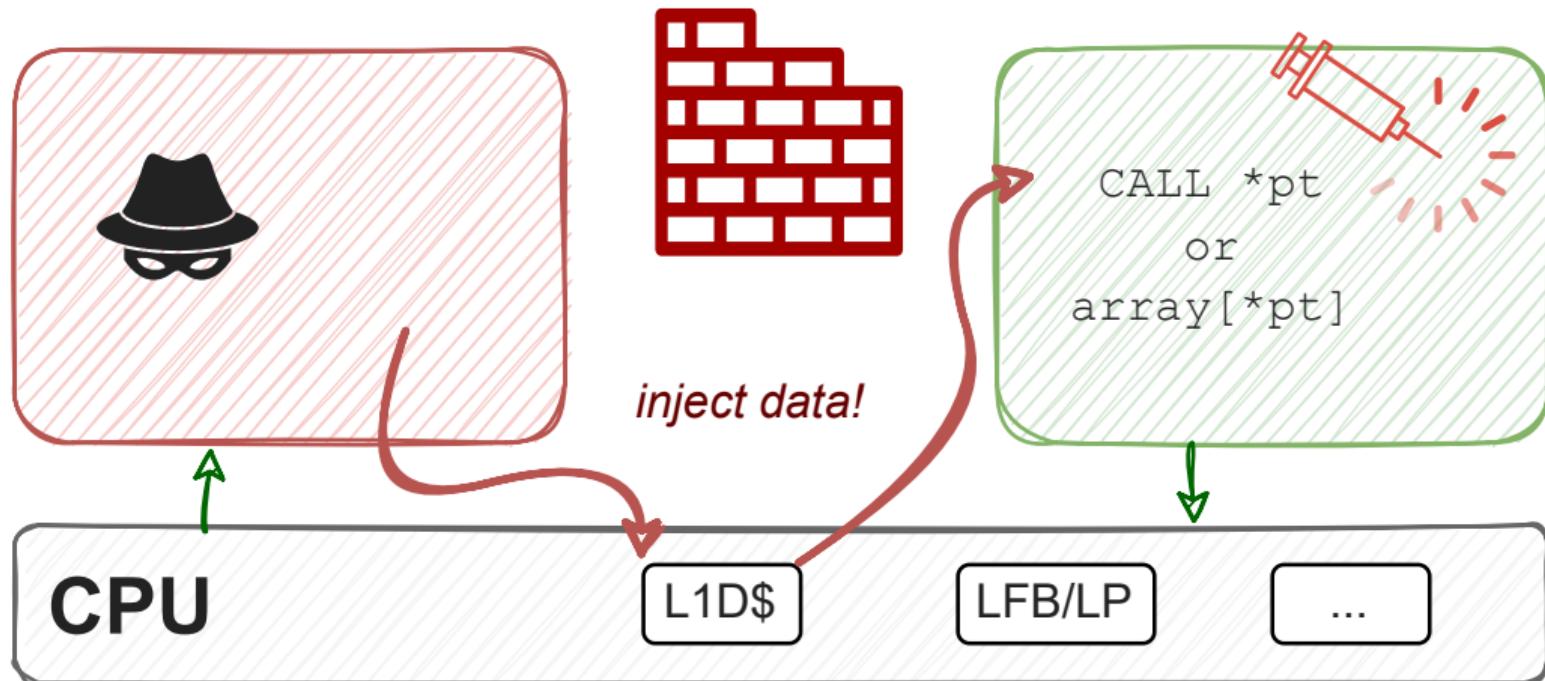
---

## 2018-2019: Leaking microarchitectural data buffers (Meltdown & friends)



Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution", USENIX 2018.

# 2020: Load Value Injection (LVI): The basic idea



Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection", S&P 2020.

# FOOD POISONING



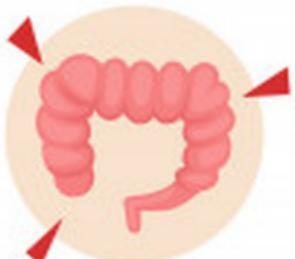
Overdue products



Medicine



Dizziness



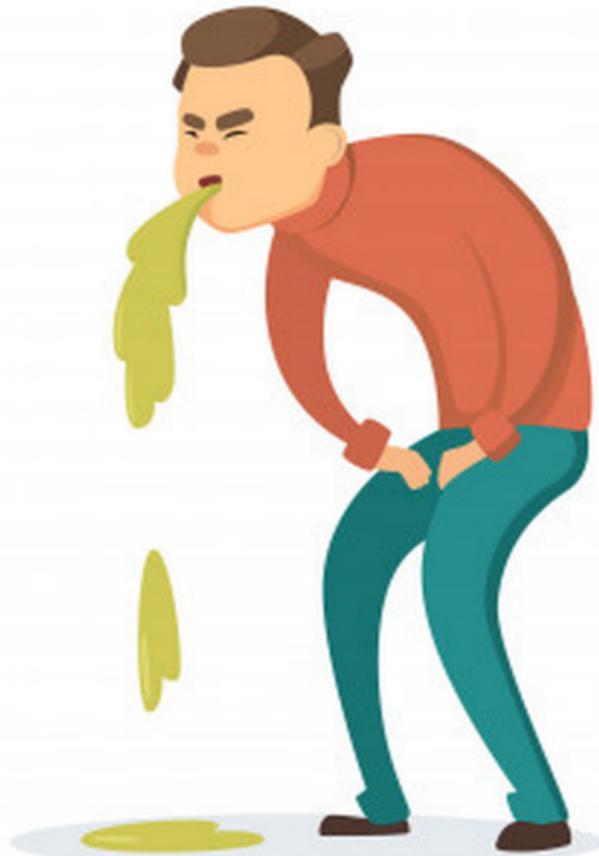
Intestinal colic



Diarrhea



Headache



## Mitigating LVI: Fencing vulnerable load instructions



# Mitigating LVI: Fencing vulnerable load instructions



## LFENCE—Load Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF AE E8	LFENCE	Z0	Valid	Valid	Serializes load operations.

A smaller red rectangular sign with the words "ALL WAY" written in white capital letters. It is mounted on a post and is positioned in front of a yellow house and some trees.

# Mitigating LVI: Compiler and assembler support



-mlfence-after-load

GNU Assembler Adds New Options For Mitigating Load Value Injection Attack

Written by Michael Larabel in [GNU](#) on 11 March 2020 at 02:55 PM EDT. [14 Comments](#)



-mlvi-hardening

LLVM Lands Performance-Hitting Mitigation For Intel LVI Vulnerability

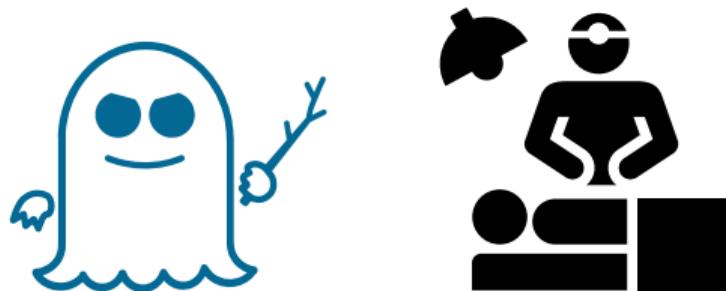
Written by Michael Larabel in [Software](#) on 3 April 2020. [Page 1 of 3.](#) [20 Comments](#)



-Qspectre-load

More Spectre Mitigations in [MSVC](#)

March 13th, 2020

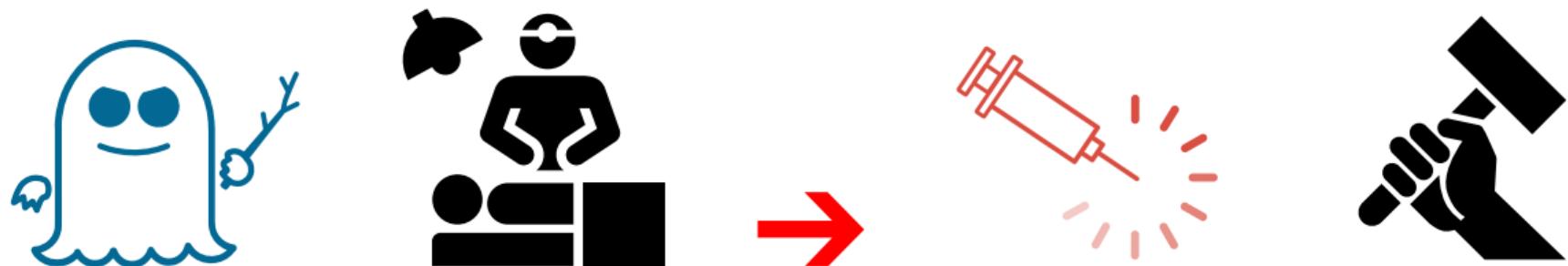


**23 fences**

October 2019—“surgical precision”

# Intel architectural enclaves: lfence counts

libsgx\_qe.signed.so



**23 fences**

October 2019—“surgical precision”

**49,315 fences**

March 2020—“big hammer”



## GNU Assembler Adds New Options For Mitigating Load Value Injection Attack

Written by Michael Larabel in [GNU](#) on 11 March 2020 at 02:55 PM EDT. [14 Comments](#)

## The Brutal Performance Impact From Mitigating The LVI Vulnerability

Written by Michael Larabel in [Software](#) on 12 March 2020. [Page 1 of 6](#). [76 Comments](#)

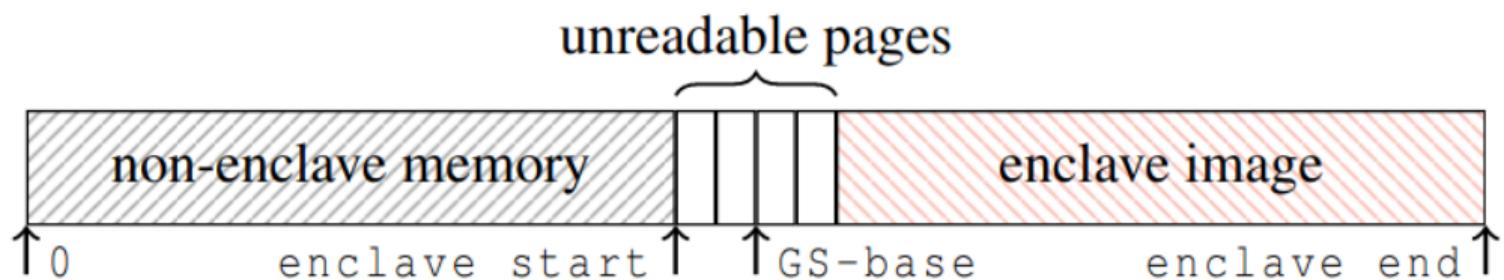
## LLVM Lands Performance-Hitting Mitigation For Intel LVI Vulnerability

Written by Michael Larabel in [Software](#) on 3 April 2020. [Page 1 of 3](#). [20 Comments](#)

## Looking At The LVI Mitigation Impact On Intel Cascade Lake Refresh

Written by Michael Larabel in [Software](#) on 5 April 2020. [Page 1 of 5](#). [10 Comments](#)

# LVI-NULL compiler mitigation



Giner et al. "Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX", USENIX Security 2022.

# Conclusions and takeaway

- ⇒ Trusted execution environments (Intel SGX) ≠ perfect!
- ⇒ Need for (compiler) mitigations:
  1. ABI/API sanitization
  2. Side-channel hardening: constant-time (or balanced?) code
  3. Transient-execution semantics

