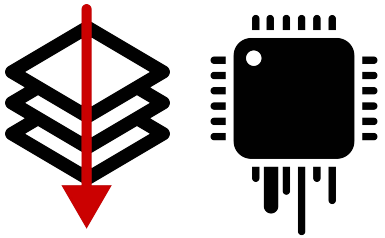


Fast by Design, Leaky by Consequence:

How Microarchitectural Attacks Dissolve CPU Isolation Boundaries

Jo Van Bulck

DistriNet, KU Leuven, Belgium jo.vanbulck@cs.kuleuven.be vanbulck.net

COSIC Course on Cryptography and Cybersecurity (June 25, 2026)

DistriNet

KU LEUVEN

Thursday 25 June TRACK B

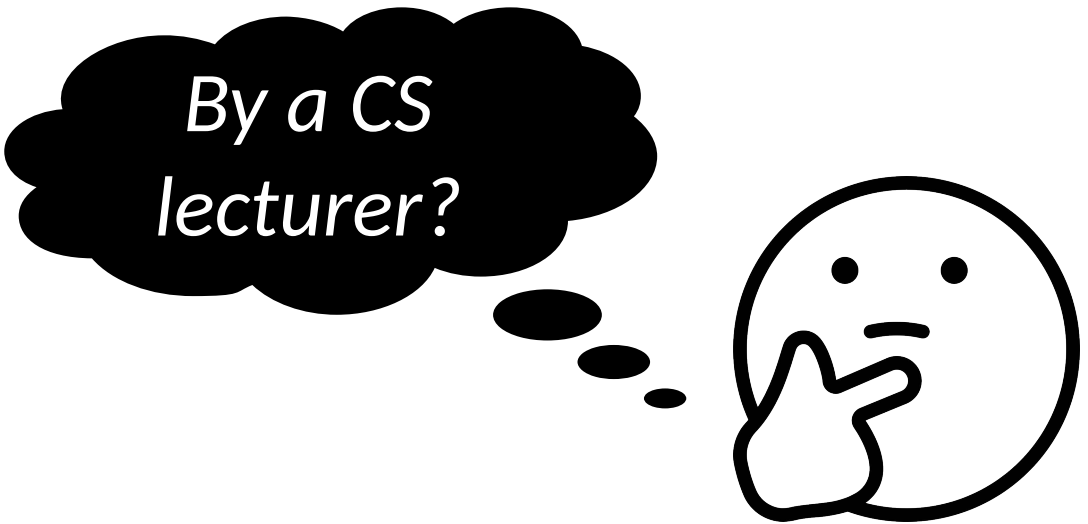
Hardware Security track

(00.39 – aula MTM)

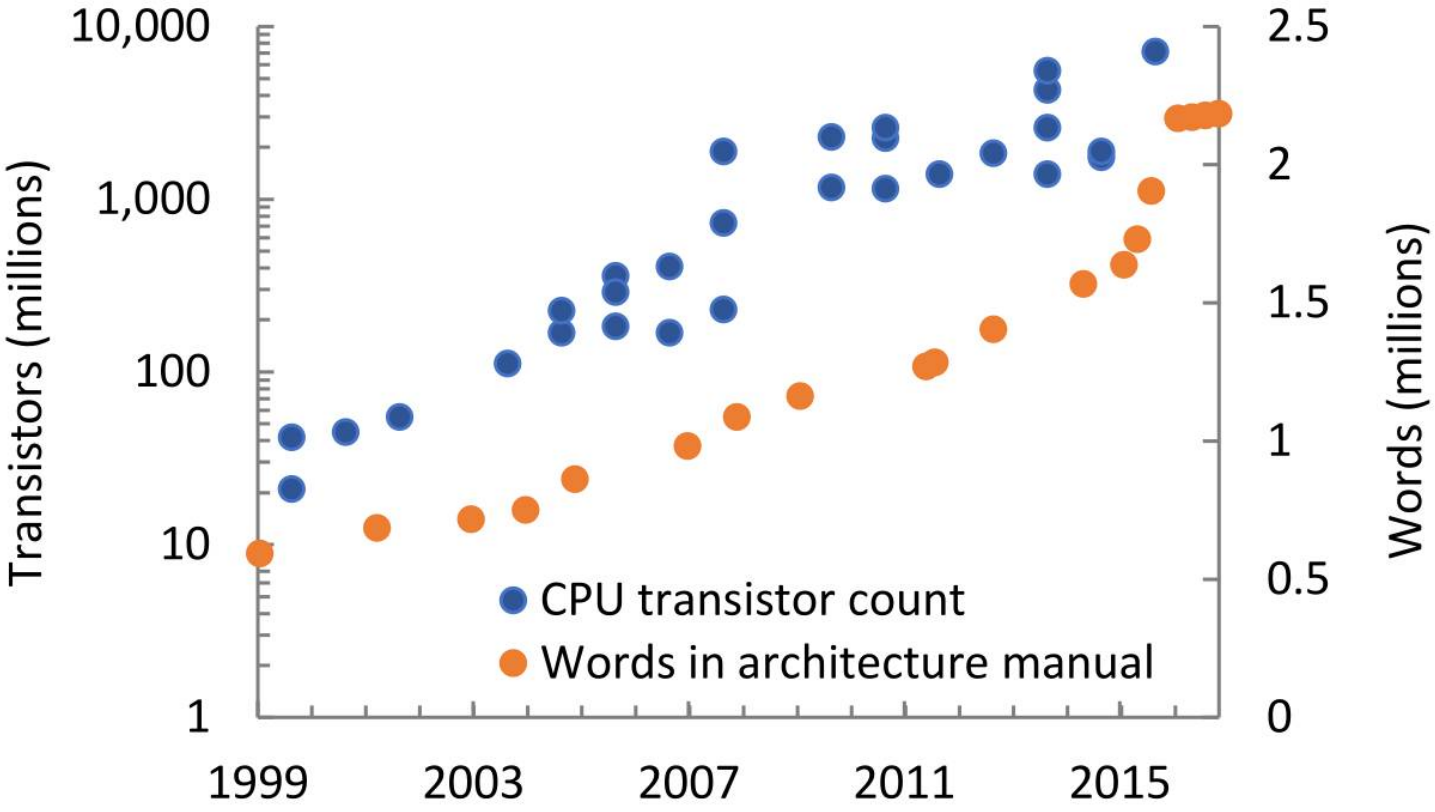
Thursday 25 June TRACK B

Hardware Security track

(00.39 – aula MTM)

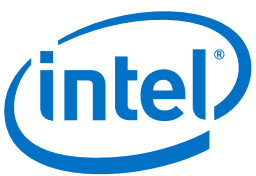


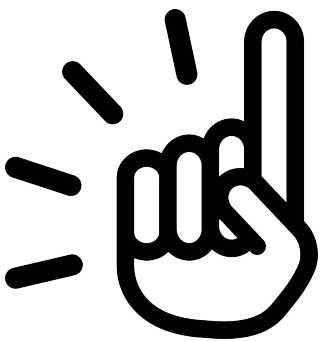
Modern ISA Complexity: Hardware is the New Software!



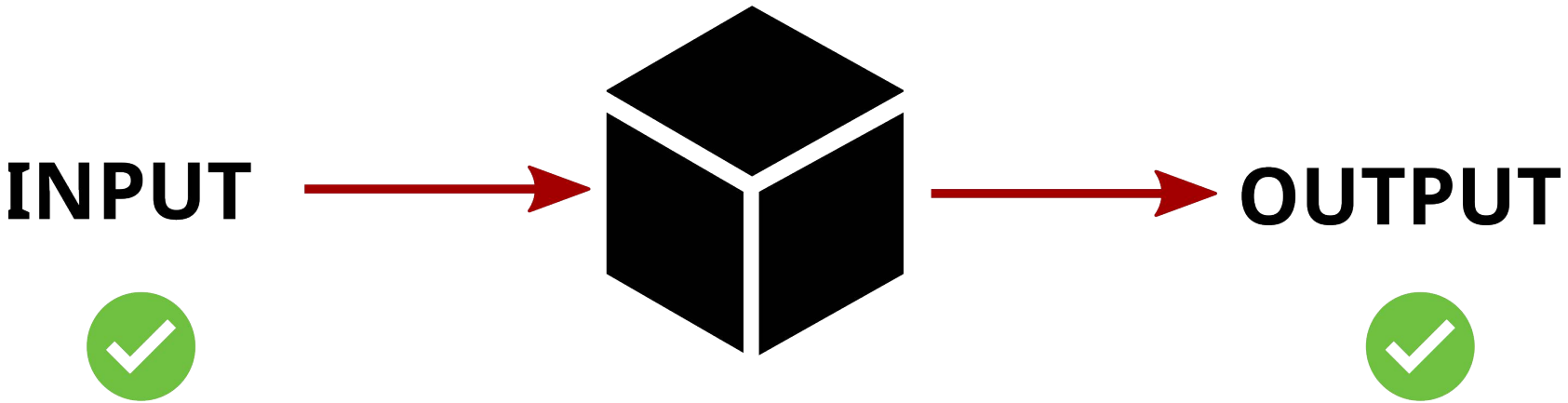
Baumann, "Hardware is the New Software", HotOS'17.

How to write "secure software" on modern processors?



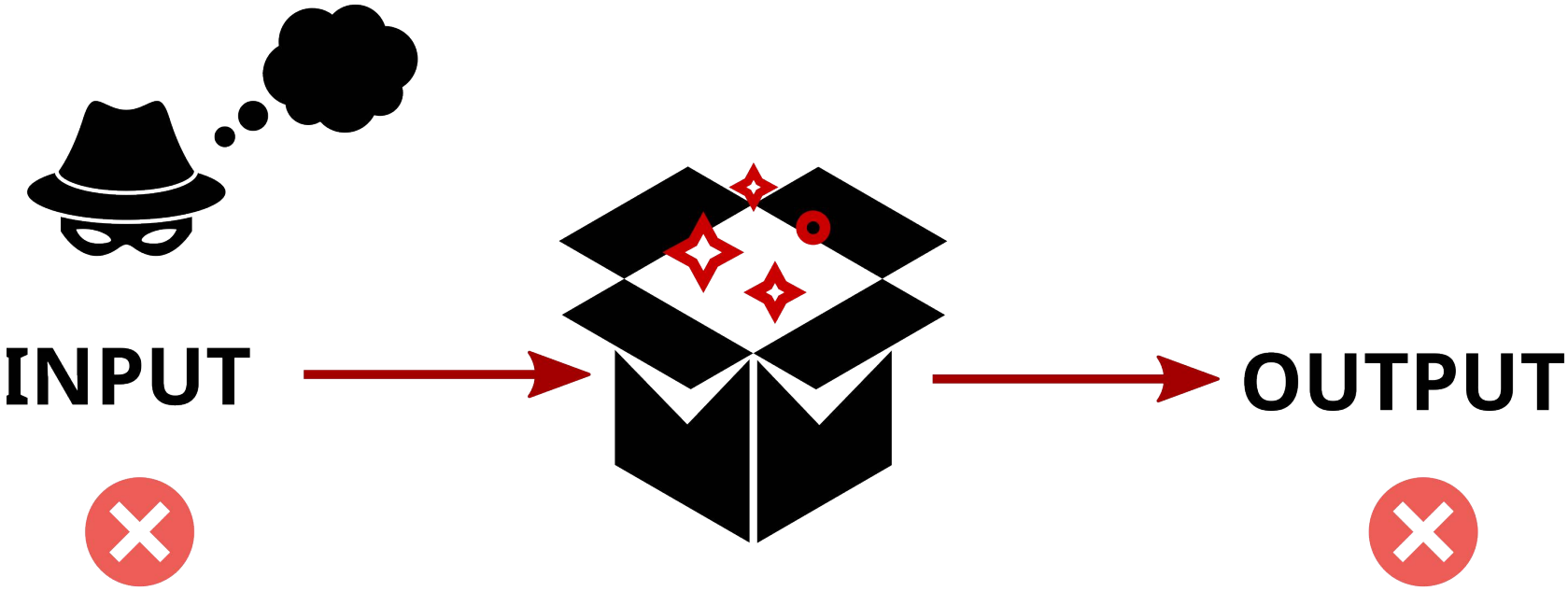


A Primer on Software Security



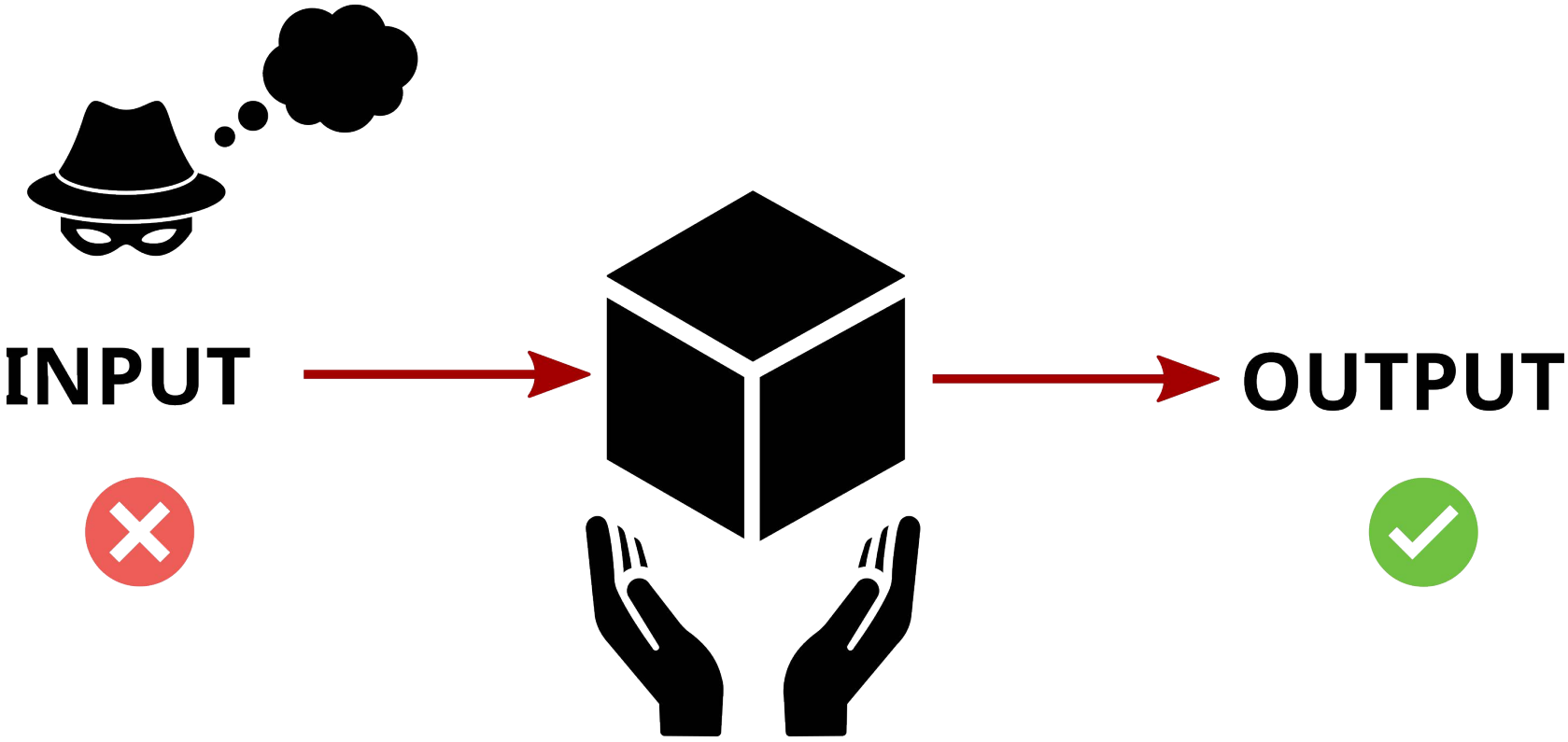
Secure program: Convert all input to *expected output*

A Primer on Software Security: Memory Safety



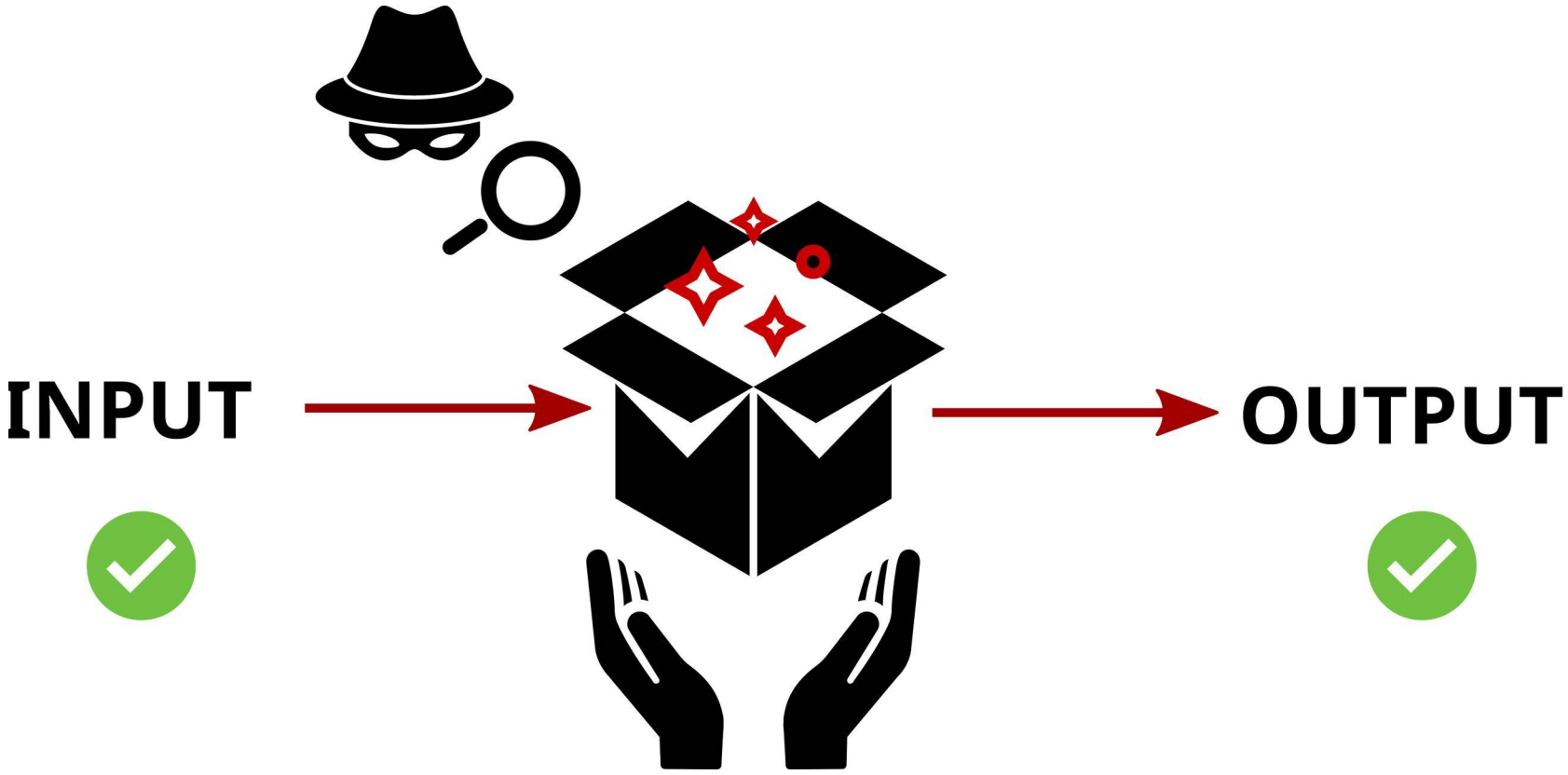
Buffer overflow vulnerabilities: trigger *unexpected behavior*

A Primer on Software Security: Memory Safety



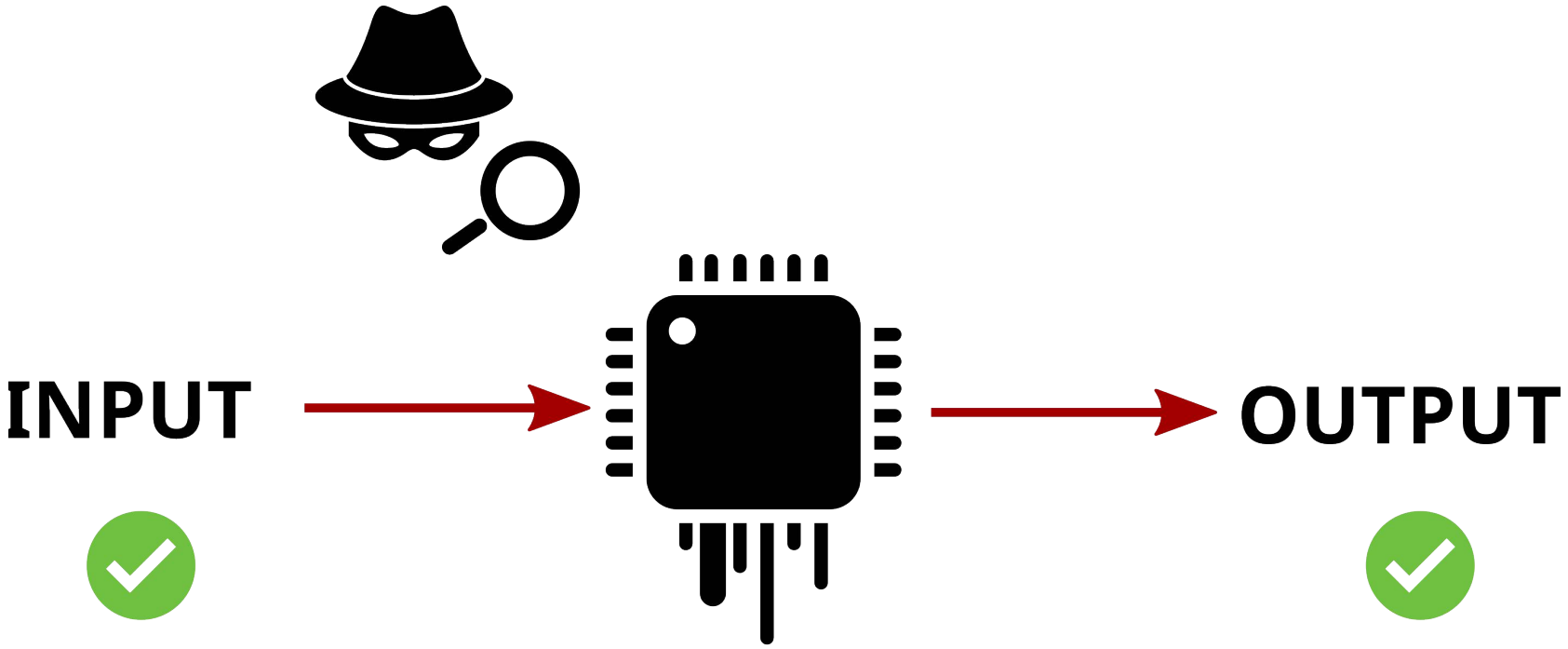
Safe languages & formal verification: Preserve *expected behavior*

A Primer on Software Security: Side-Channel Analysis



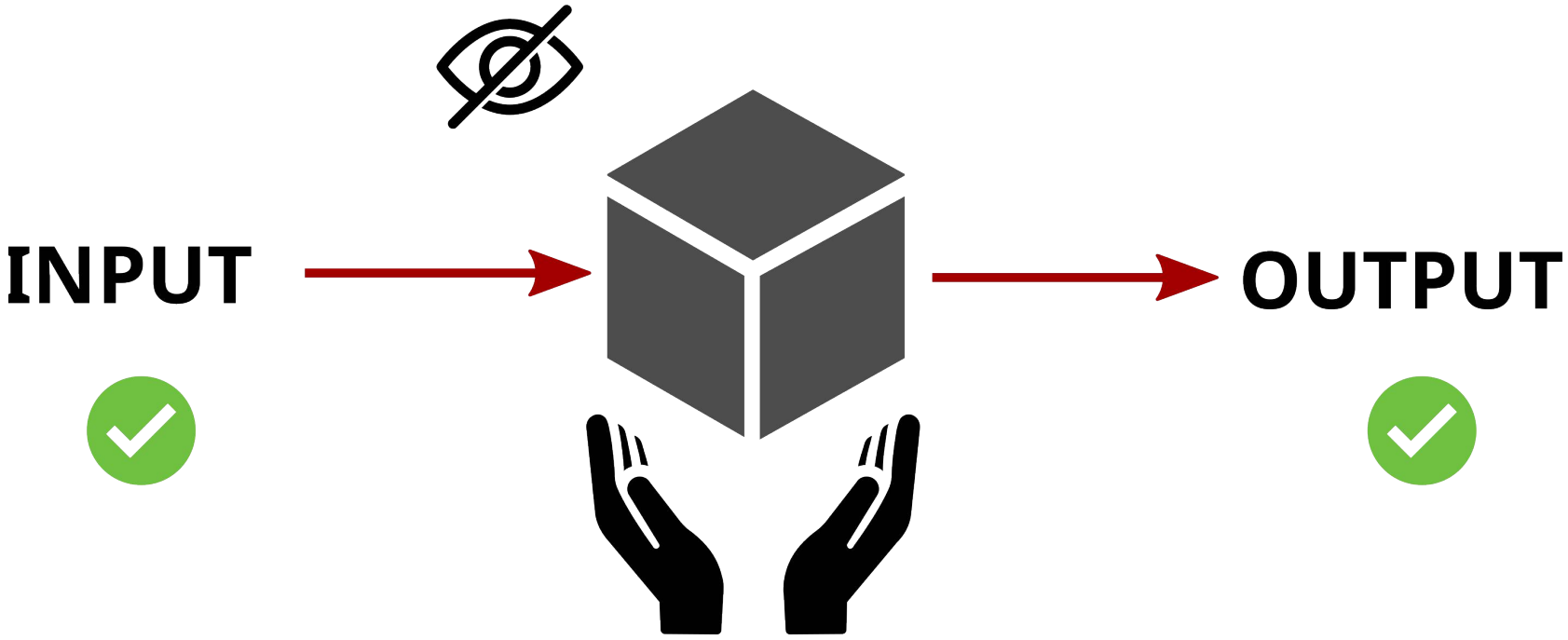
Side-channel attacks: Observe *side-effects* of the computation

A Primer on Software Security: Side-Channel Analysis



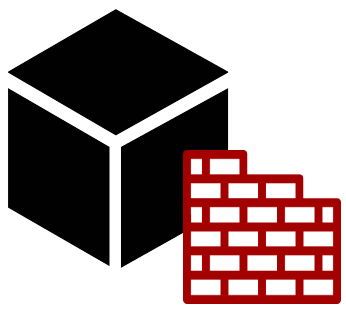
Microarchitectural leaks: *CPU timing variations* may leak SW secrets(!)

A Primer on Software Security: Constant-Time Programming



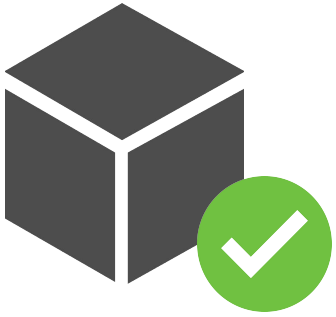
Constant-time code: Eliminate *secret-dependent* side-effects

Road Map



System Model

Timing Attacks

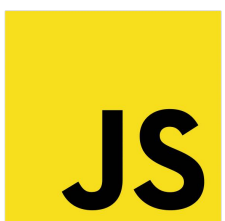


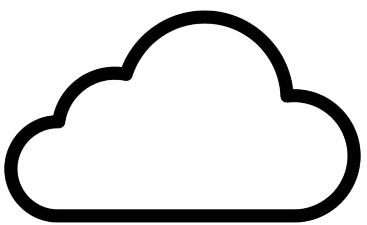
Constant Time

System Model: Shared Platform with Untrusted Code

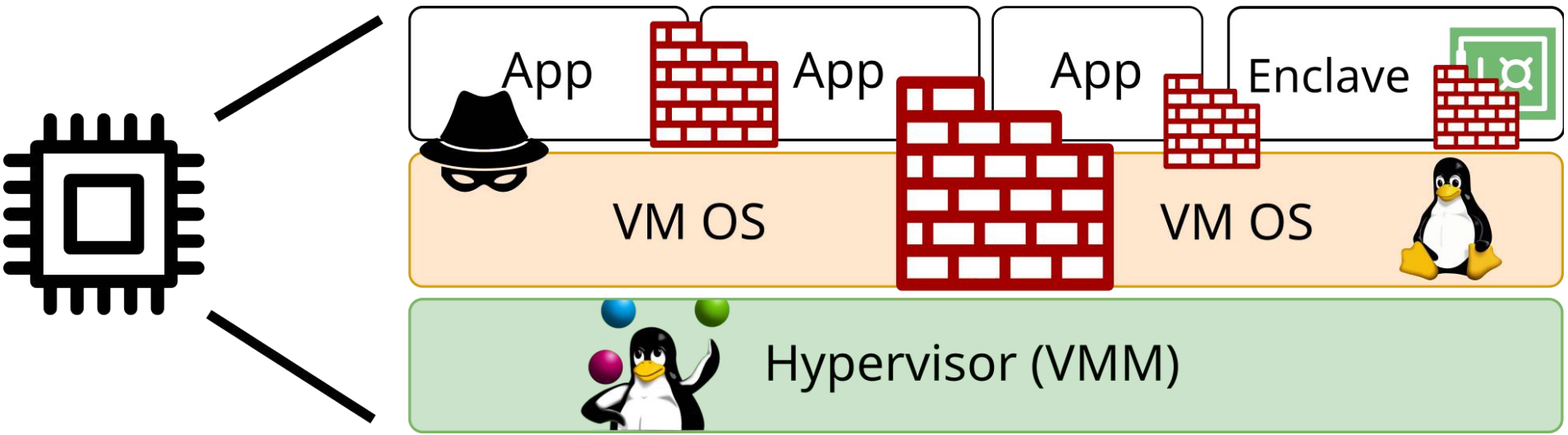
- A shared platform executing code from different stakeholders
- Attacker can execute code on the same shared platform as the victim

  → *e.g., JavaScript, cloud, app stores, etc.*
- Attacker knows the implementation details of the platform and the victim code



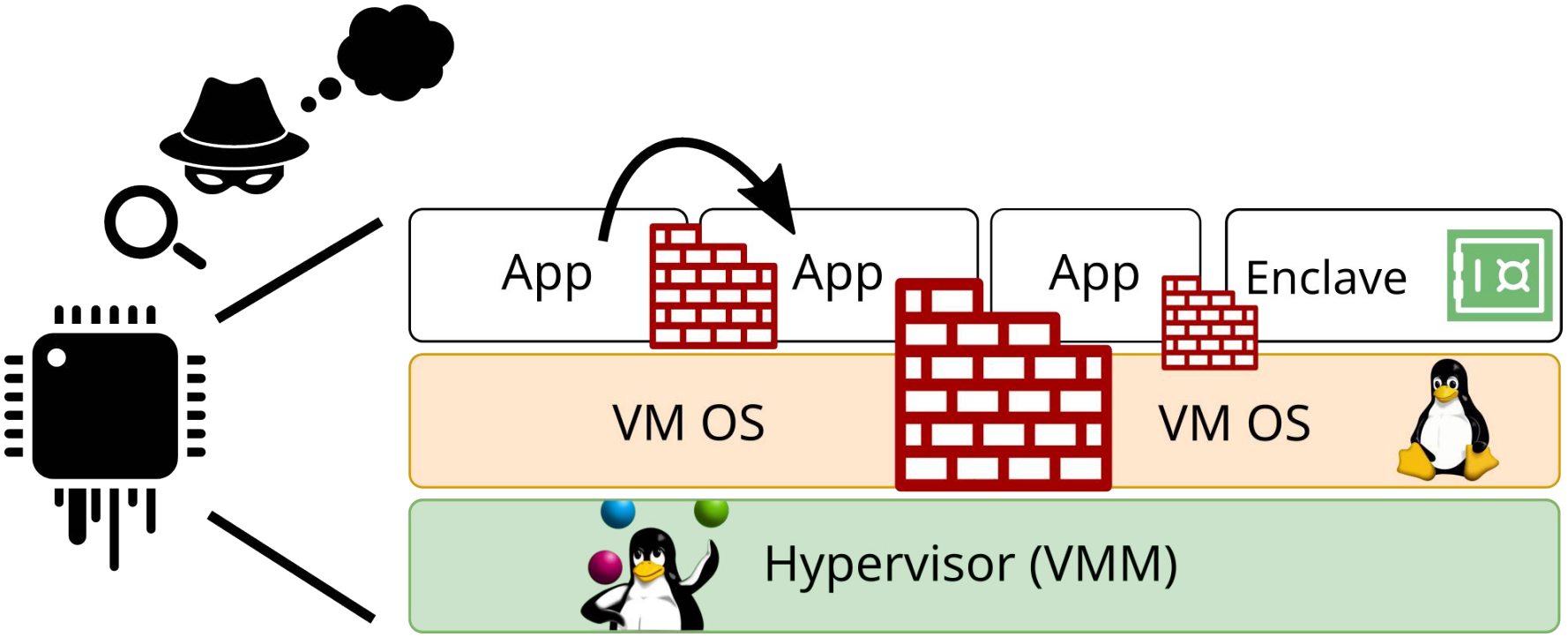


Summary: Architectural CPU Support for Software Security



- Different software **protection domains:** Processes, virtual machines, (enclaves)
- CPU builds “walls” for **memory isolation** between apps and privilege levels

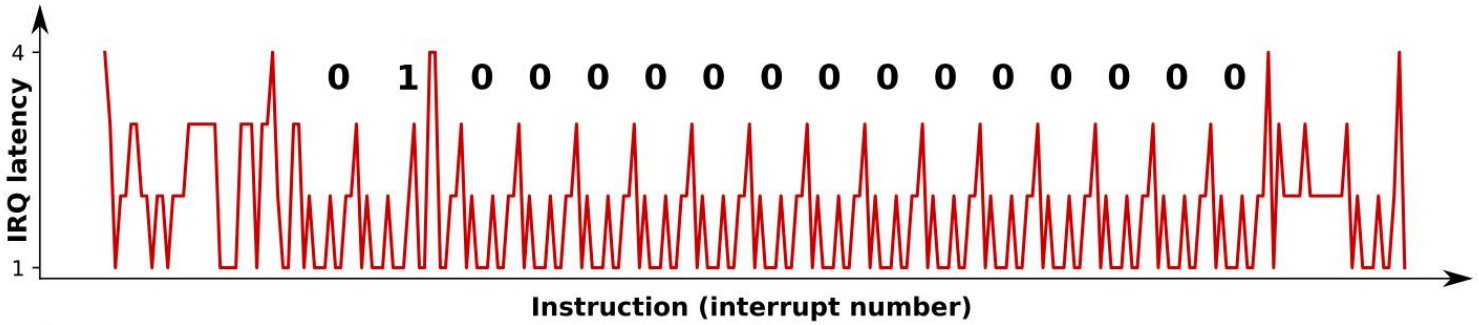
Summary: Architectural CPU Support for Software Security

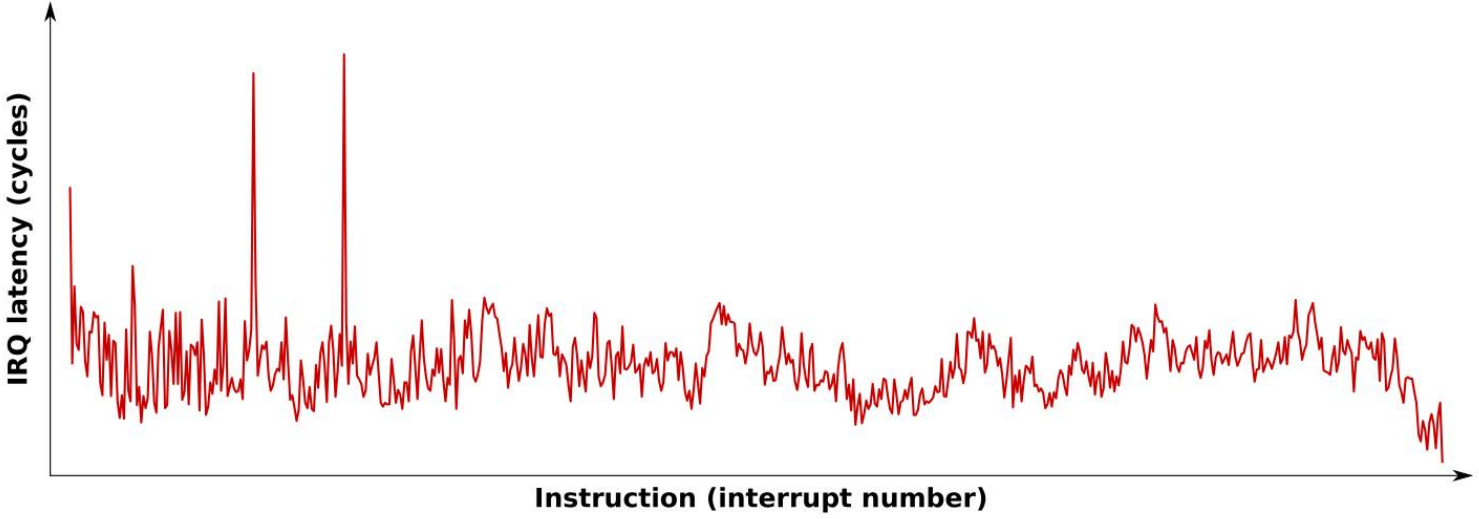


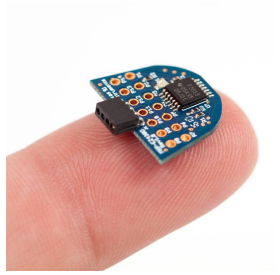
- Different software **protection domains:** Processes, virtual machines, (enclaves)
- CPU builds “walls” for **memory isolation** between apps and privilege levels
- ↔ *But architectural protection walls permeate microarchitectural side channels!*


VAULT DOOR
WEIGHT: 22 1/2 Tons
THICKNESS: 22 Inches
STEEL: 11 Layers of Special Cutting and Drill Resistant
LOCKS: 4 Hamilton Watch Movements for Time Locks

Microarchitectural Timing Leaks in Practice







Texas Instruments

intel

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

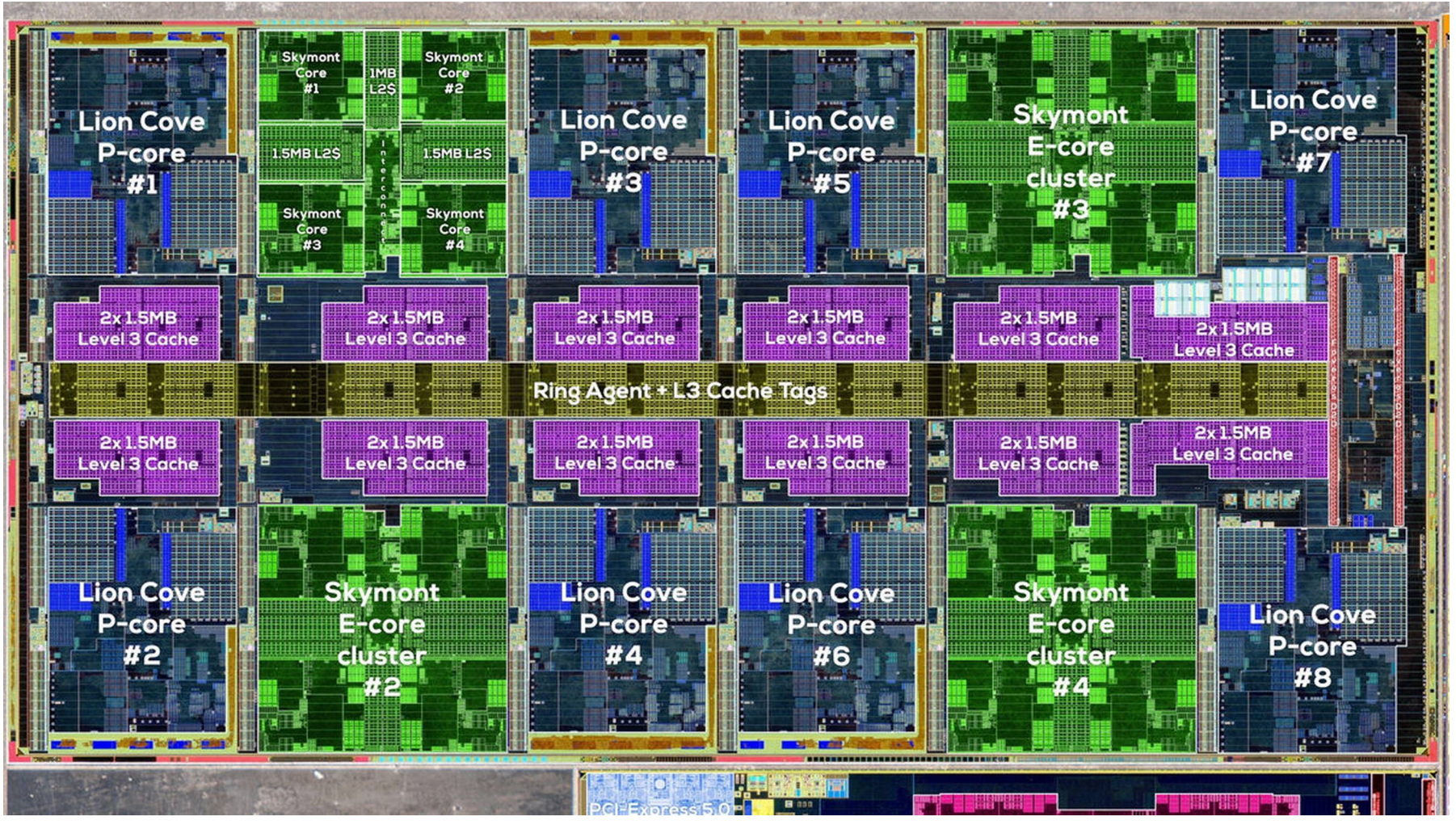
Architecture versus Microarchitecture

- The **Instruction Set Architecture (ISA)** defines behavior of the machine code:
  - Examples: x86, RISC-V, ARM, …
  - The ISA defines:
    - Architectural state: memory, registers, …
    - Instruction semantics

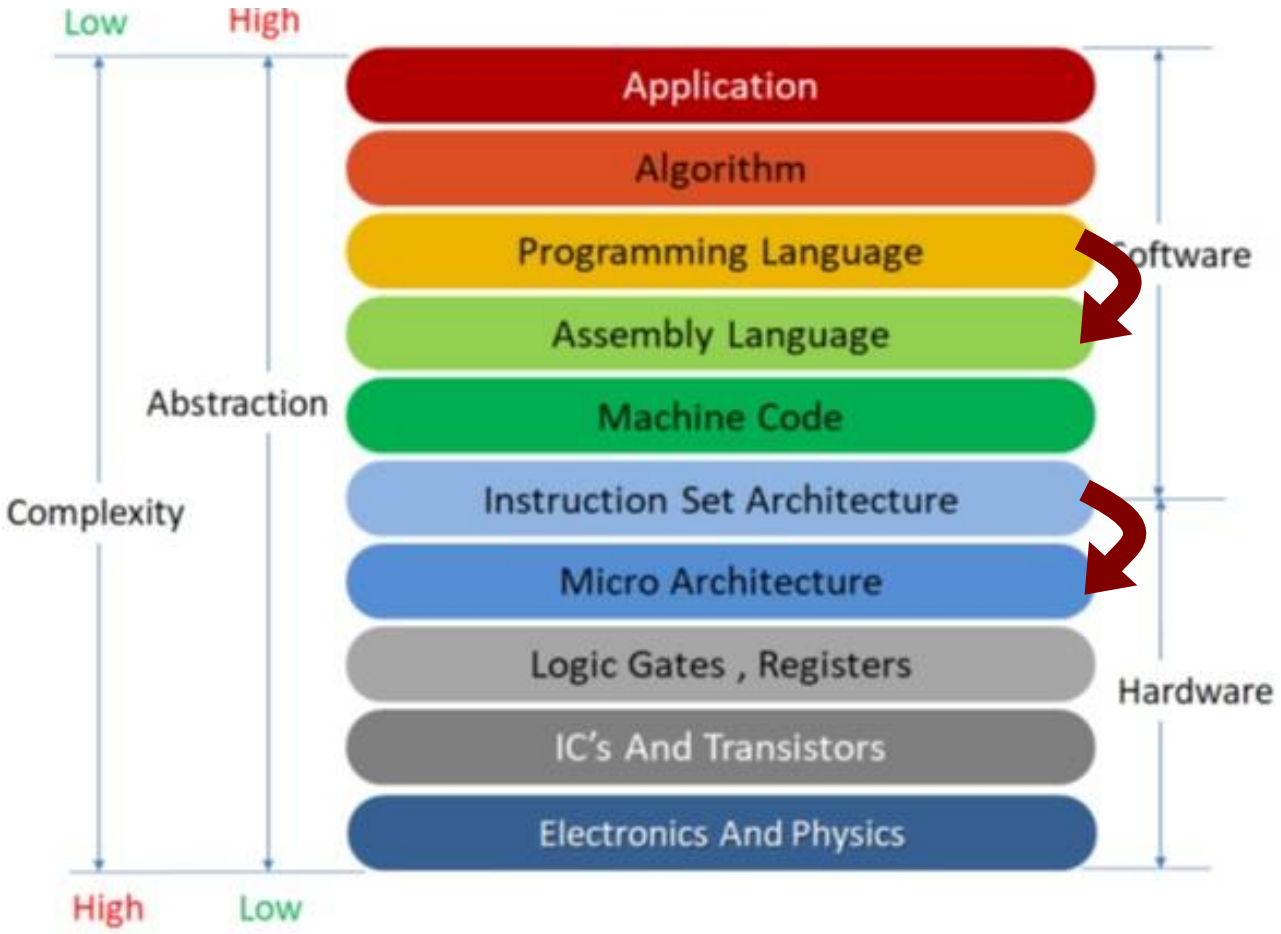
- The **microarchitecture** is the way the ISA is implemented in a particular processor:
  - Examples: single-cycle versus pipelined, in-order versus out-of-order, …
  - This can introduce additional state and behavior:
    - State: e.g., for performance improvements (caches, branch predictor state, various CPU buffers, …)
    - Behavior: speculative execution, out-of-order execution, …

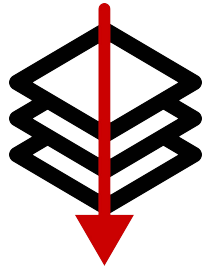
From Architecture...

From Architecture… to Microarchitecture



Aside: Security across the System Stack

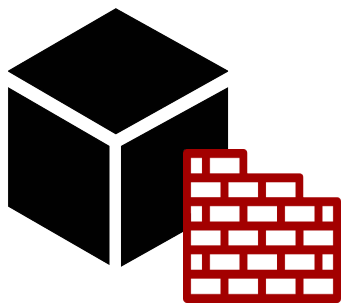




Memory safety attacks

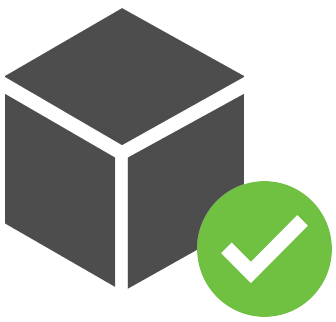
Microarchitectural attacks

Road Map



System Model

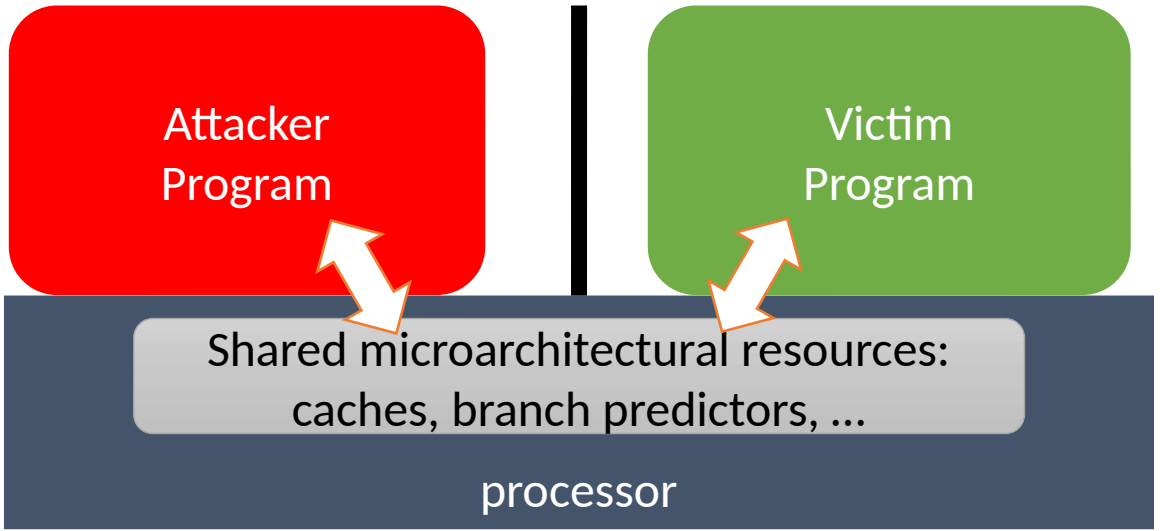
Timing Attacks



Constant Time

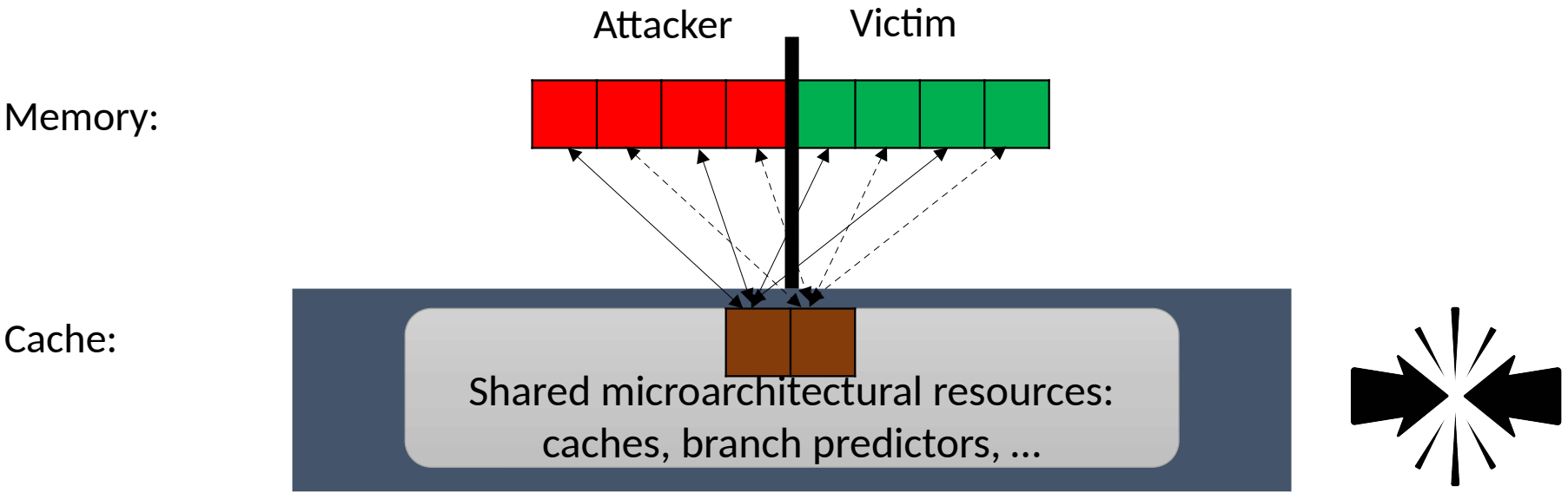
Idea: Microarchitectural Contention

- Isolation mechanisms guarantee **architectural isolation**
- **Microarchitectural attacks** aim to break isolation by exploiting the fact that the microarchitecture shares resources across isolation domains

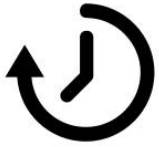


Idea: Microarchitectural Contention

- Isolation mechanisms guarantee **architectural isolation**
- **Microarchitectural attacks** aim to break isolation by exploiting the fact that the microarchitecture shares resources across isolation domains
- E.g., memory of different stakeholders can compete for the same **cache entry**



Example: CPU Cache Timing Side Channel



Cache principle: CPU speed ≫ DRAM → *cache code/data*

```
while true do
    maccess(&a);
endwh
```

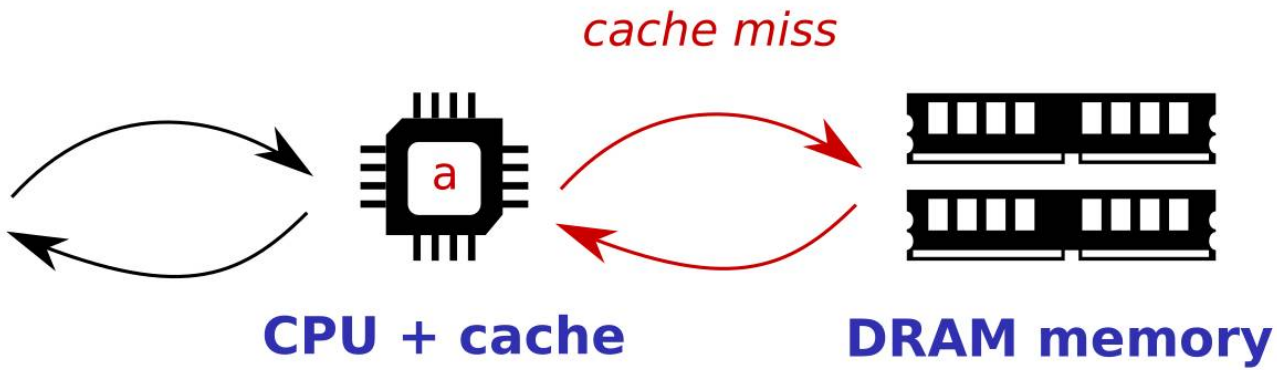
CPU + cache

DRAM memory

Example: CPU Cache Timing Side Channel

Cache miss: Request data from (slow) DRAM upon first use

```
while true do
    maccess(&a);
endwh
```

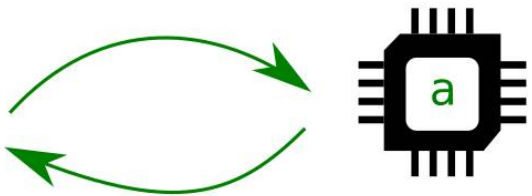


Example: CPU Cache Timing Side Channel

Cache hit: No DRAM access required for subsequent uses

```
while true do
    maccess(&a);
endwh
```

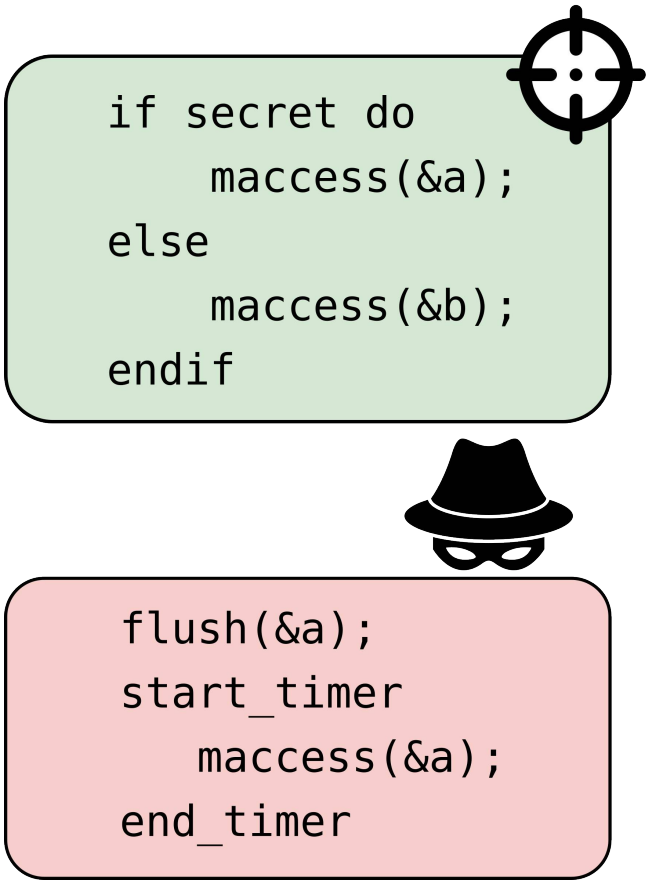
cache hit



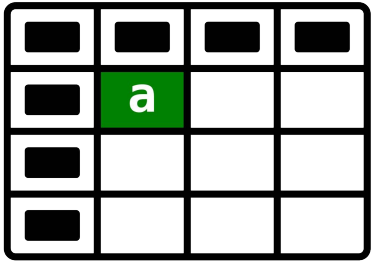
CPU + cache

DRAM memory

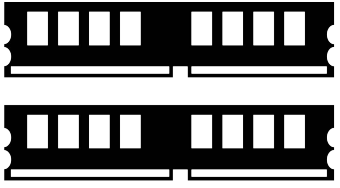
Cache Timing Attacks in Practice: Flush+Reload



'a' is accessible to attacker

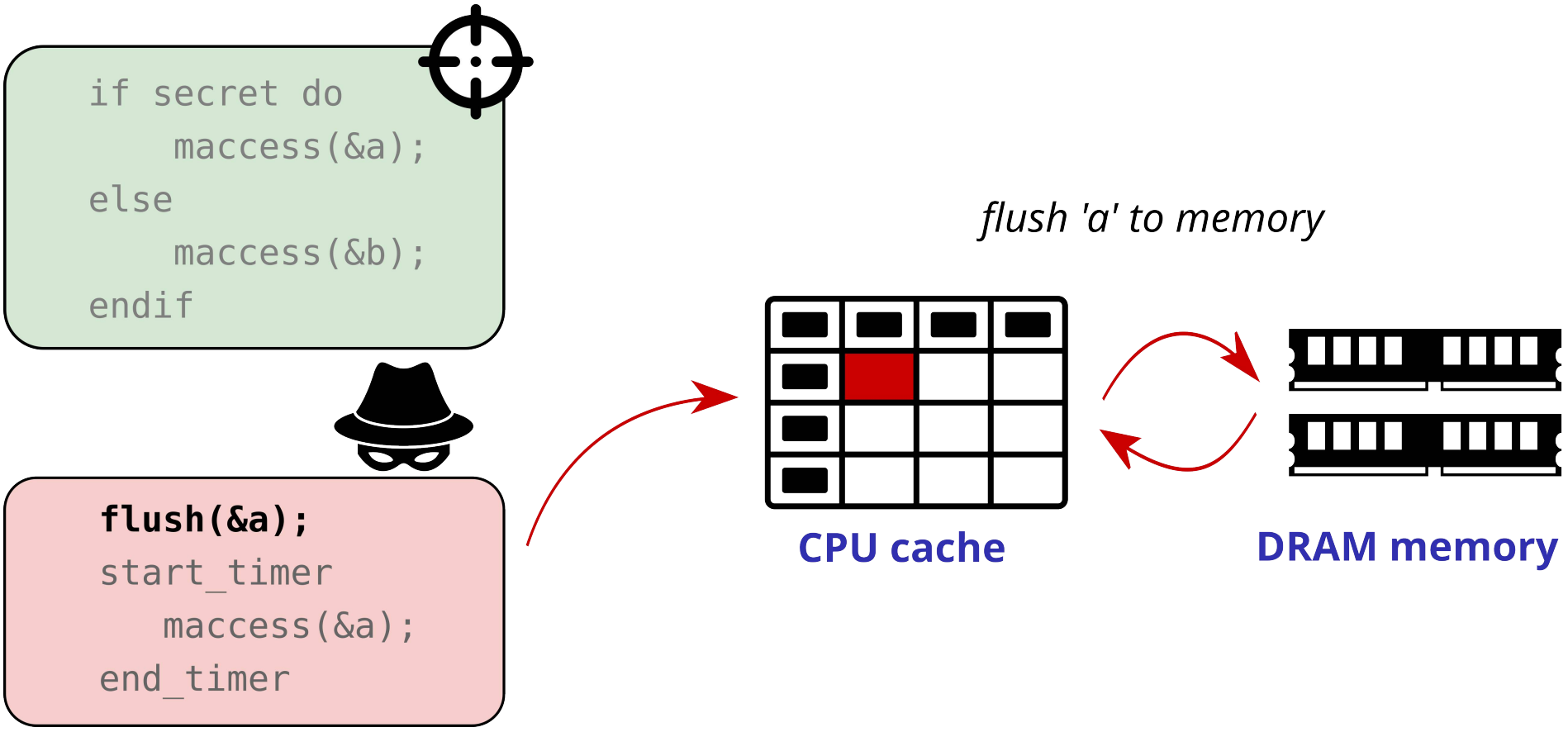


CPU cache

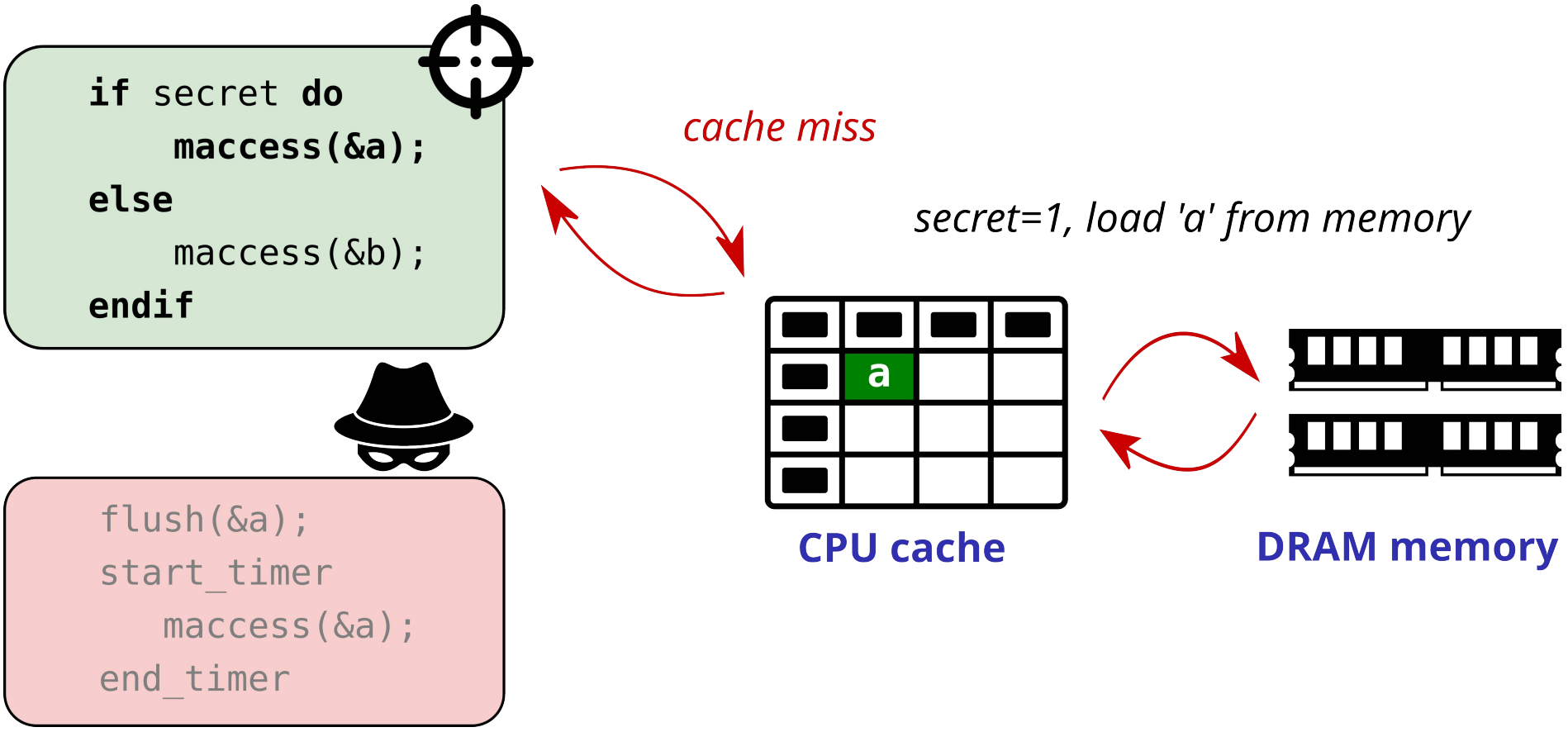


DRAM memory

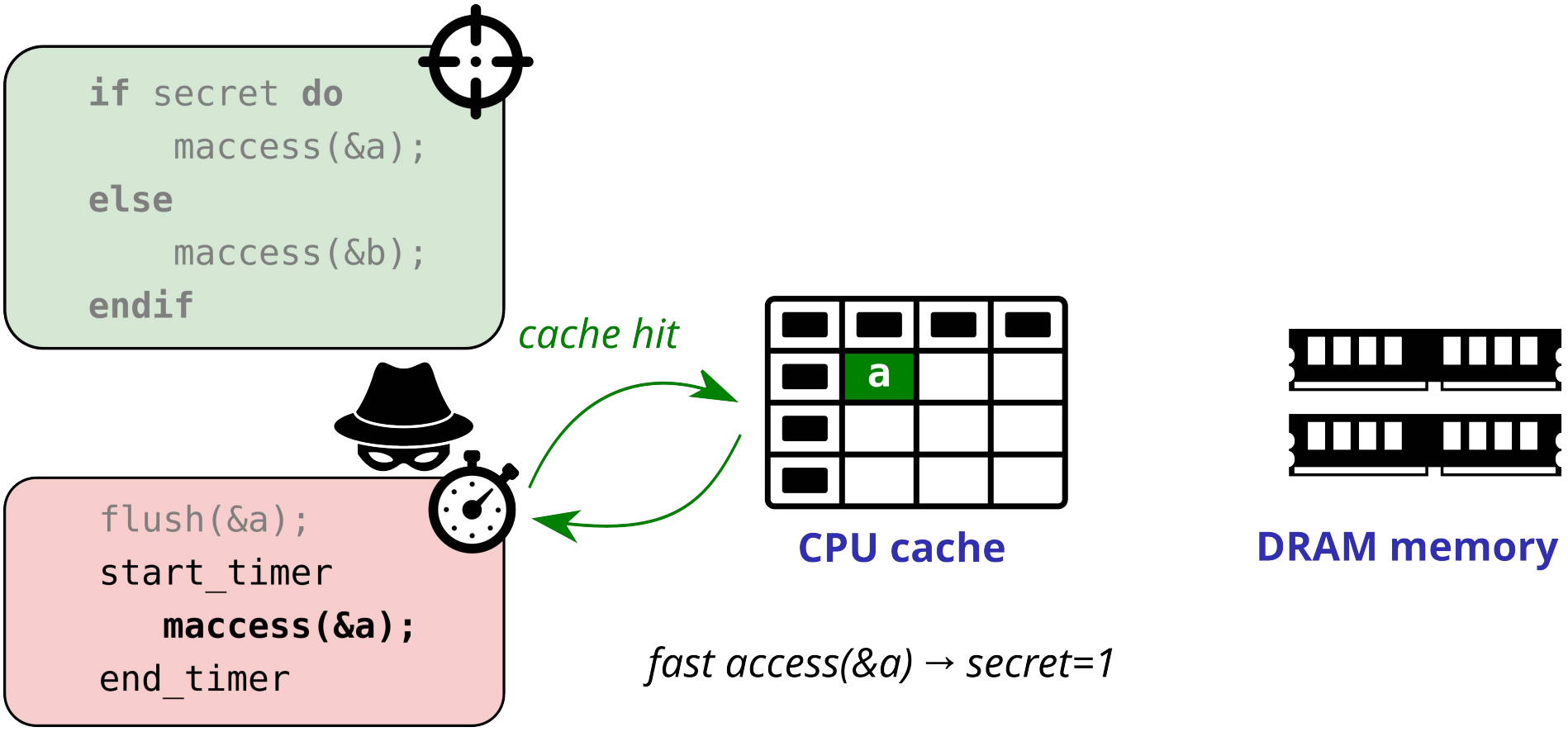
Cache Timing Attacks in Practice: Flush+Reload



Cache Timing Attacks in Practice: Flush+Reload



Cache Timing Attacks in Practice: Flush+Reload



Demo: Spying on Keystrokes with Flush+Reload

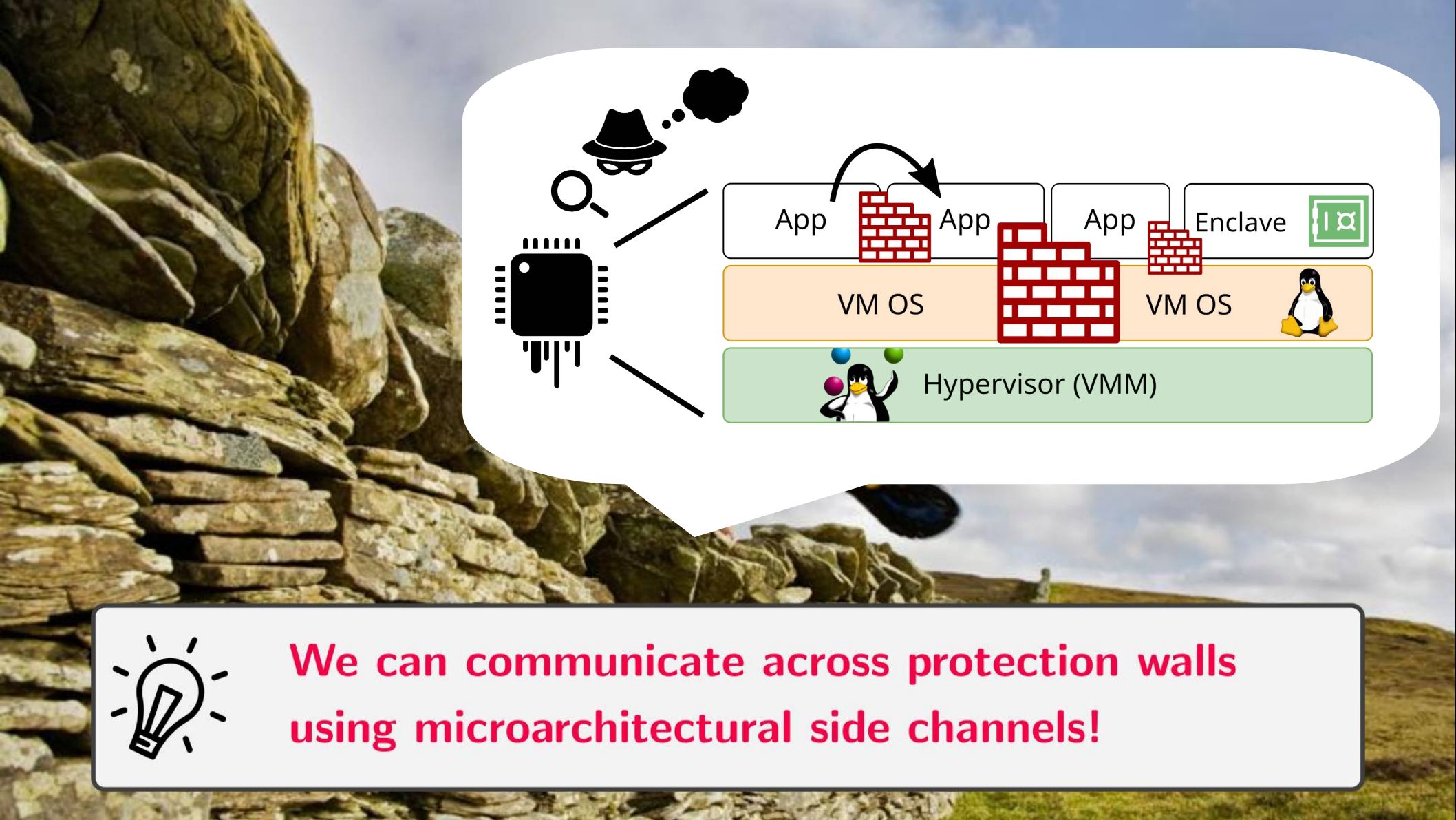
```
25336620182821: Cache Hit (218 cycles) after a pause of 3 cycles
25336620297955: Cache Hit (216 cycles) after a pause of 43 cycles
25336620341217: Cache Hit (216 cycles) after a pause of 15 cycles
25336620363985: Cache Hit (212 cycles) after a pause of 4 cycles
25336620483903: Cache Hit (218 cycles) after a pause of 42 cycles
25336620499835: Cache Hit (216 cycles) after a pause of 3 cycles
25336620552419: Cache Hit (216 cycles) after a pause of 19 cycles
25336621476911: Cache Hit (218 cycles) after a pause of 300 cycles
25336974127733: Cache Hit (220 cycles) after a pause of 104704 cycles
25337739302241: Cache Hit (214 cycles) after a pause of 263629 cycles
25337739686069: Cache Hit (218 cycles) after a pause of 116 cycles
25337739773947: Cache Hit (218 cycles) after a pause of 27 cycles
25337739997613: Cache Hit (228 cycles) after a pause of 84 cycles
25338346337023: Cache Hit (228 cycles) after a pause of 211810 cycles
25338346617849: Cache Hit (224 cycles) after a pause of 81 cycles
25338346627851: Cache Hit (228 cycles) after a pause of 2 cycles
25338346634917: Cache Hit (228 cycles) after a pause of 1 cycles
25338346653587: Cache Hit (222 cycles) after a pause of 5 cycles
25338346811743: Cache Hit (220 cycles) after a pause of 58 cycles
25338346899541: Cache Hit (222 cycles) after a pause of 35 cycles
25338346911083: Cache Hit (222 cycles) after a pause of 3 cycles
25339081895869: Cache Hit (204 cycles) after a pause of 268339 cycles
25339081934737: Cache Hit (228 cycles) after a pause of 3 cycles
25339082052305: Cache Hit (226 cycles) after a pause of 34 cycles
25339082092569: Cache Hit (228 cycles) after a pause of 8 cycles
25339082116253: Cache Hit (224 cycles) after a pause of 3 cycles
25339082273651: Cache Hit (202 cycles) after a pause of 53 cycles
25339815487639: Cache Hit (226 cycles) after a pause of 232157 cycles
```

https://github.com/isec-tugraz/cache_template_attacks

```
3 super secret keystroke timings
4
5 F
```

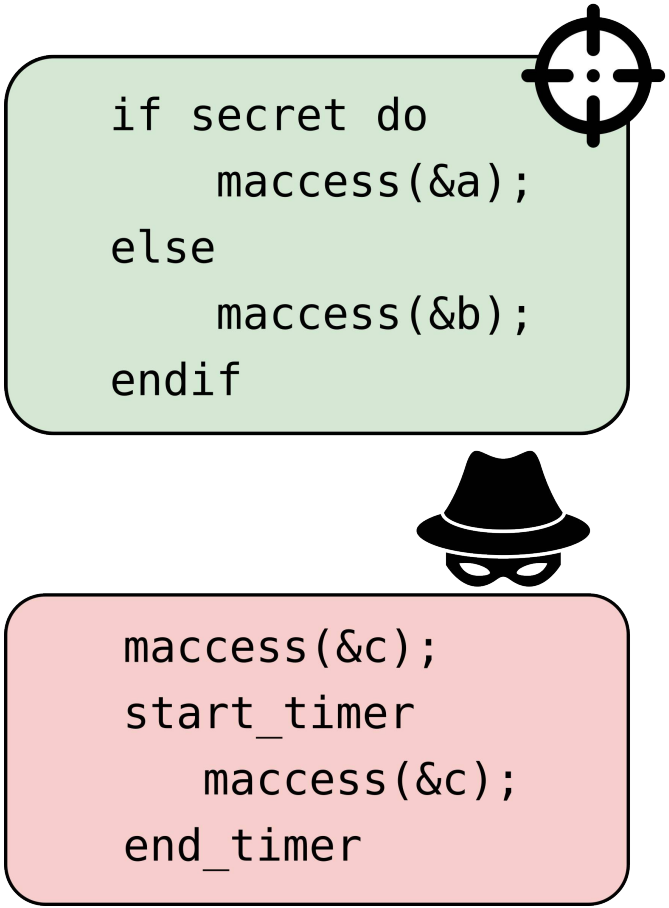
Plain Text Tab Width: 8 Ln 5, Col 2 INS


We can communicate across protection walls
using microarchitectural side channels!

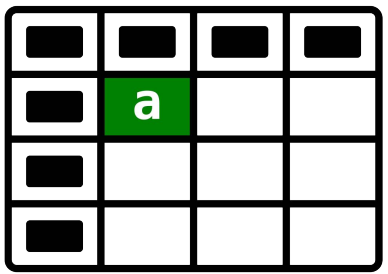


We can communicate across protection walls using microarchitectural side channels!

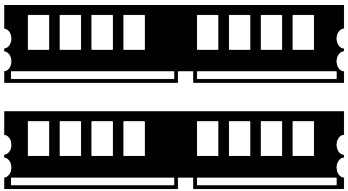
Prime+Probe: Generalized Cache Timing Attacks



*'a' is **not** accessible to attacker*

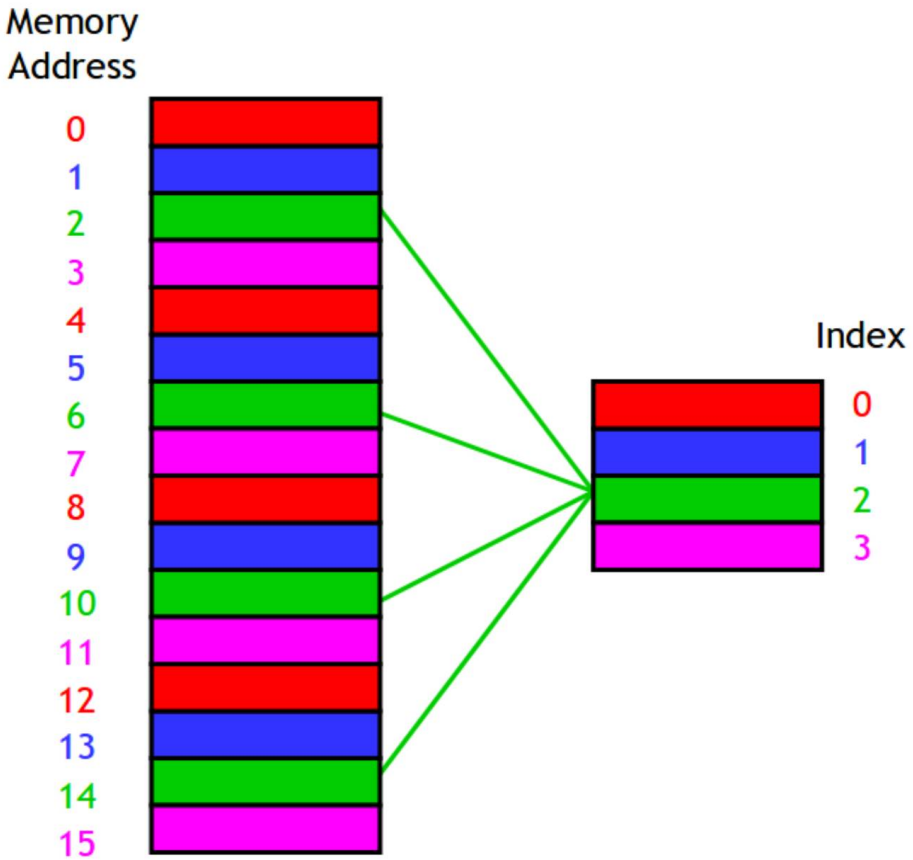


CPU cache

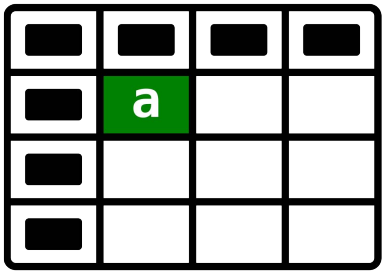


DRAM memory

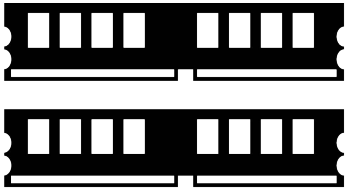
Prime+Probe: Generalized Cache Timing Attacks



*'a' is **not** accessible to attacker*

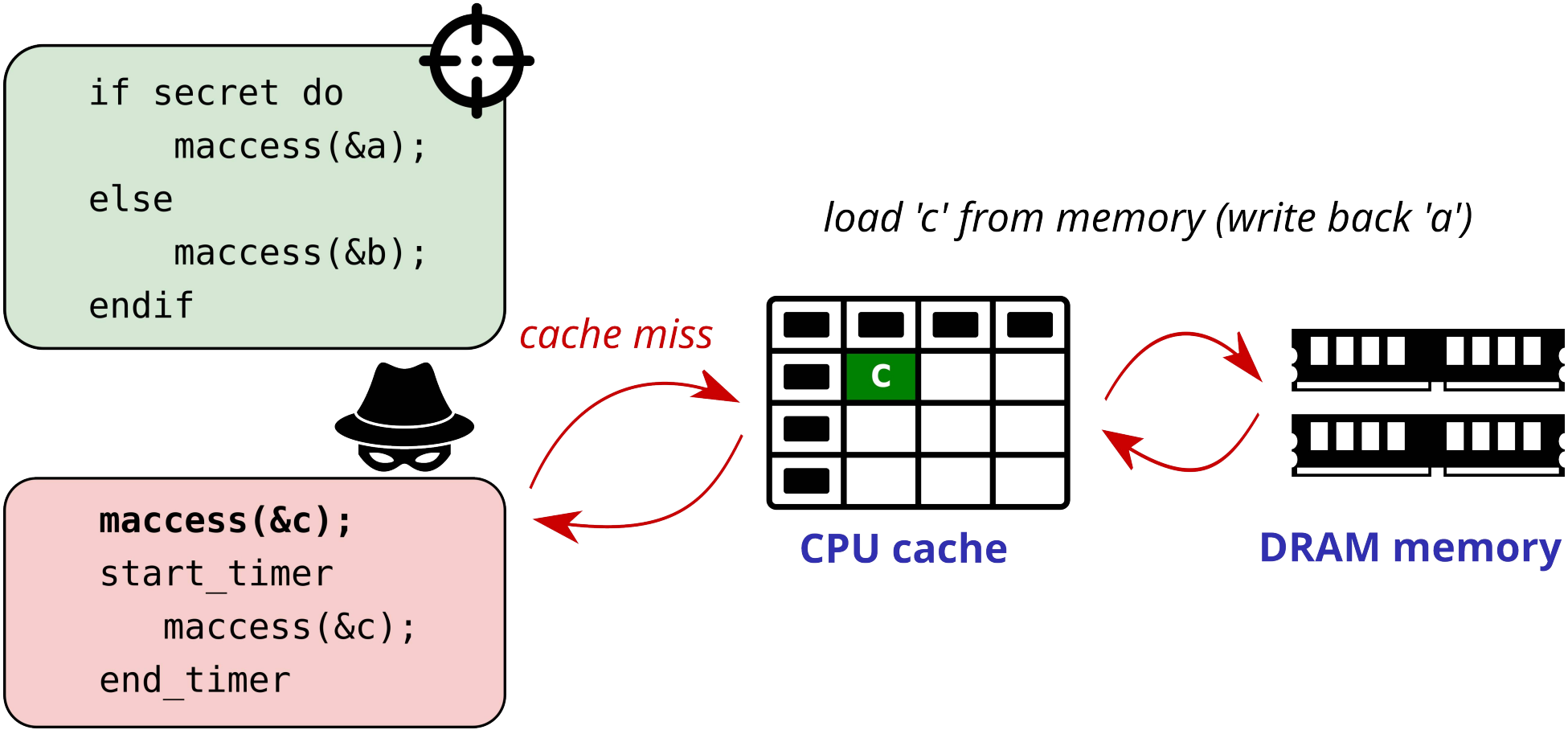


CPU cache

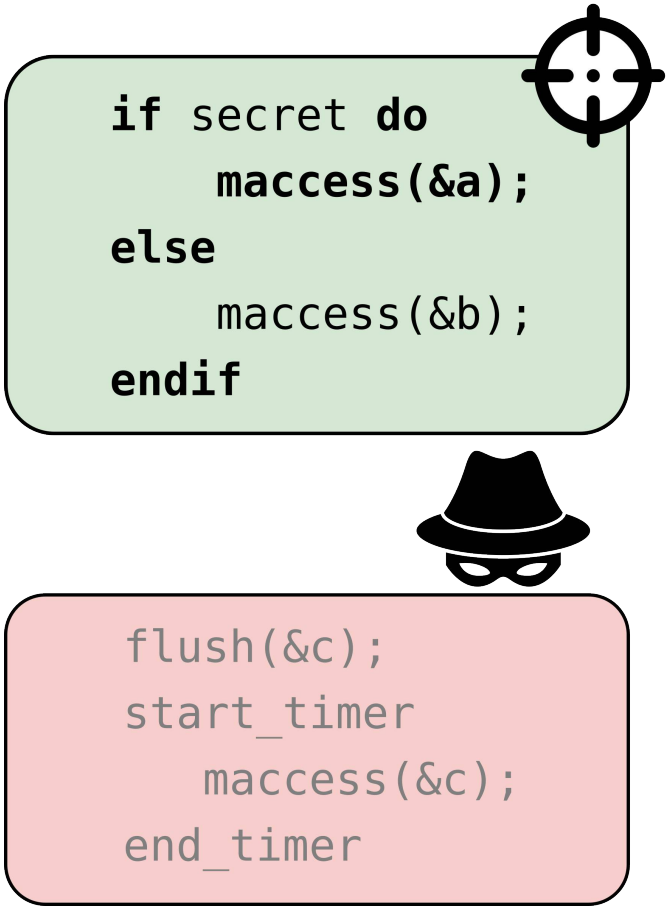


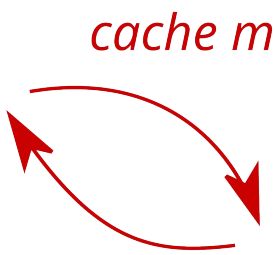
DRAM memory

Prime+Probe: Generalized Cache Timing Attacks

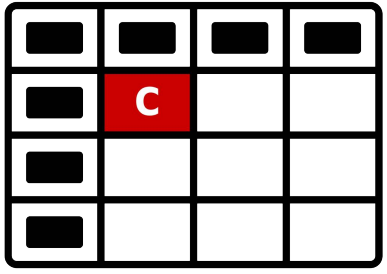


Prime+Probe: Generalized Cache Timing Attacks

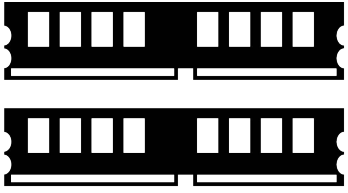




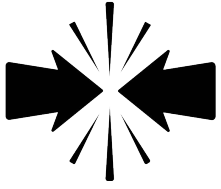
'a' and 'c' share the same cache line (!)



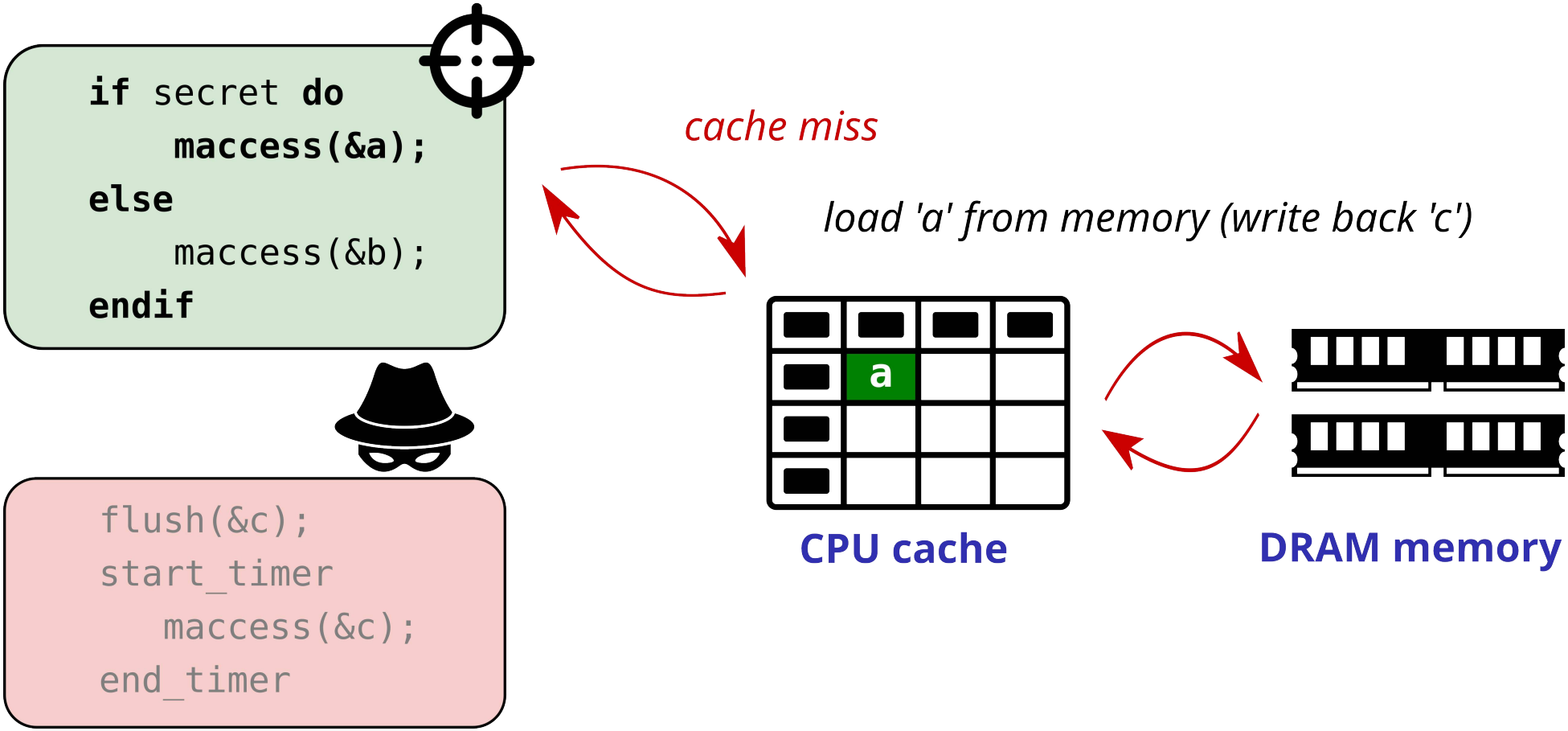
CPU cache



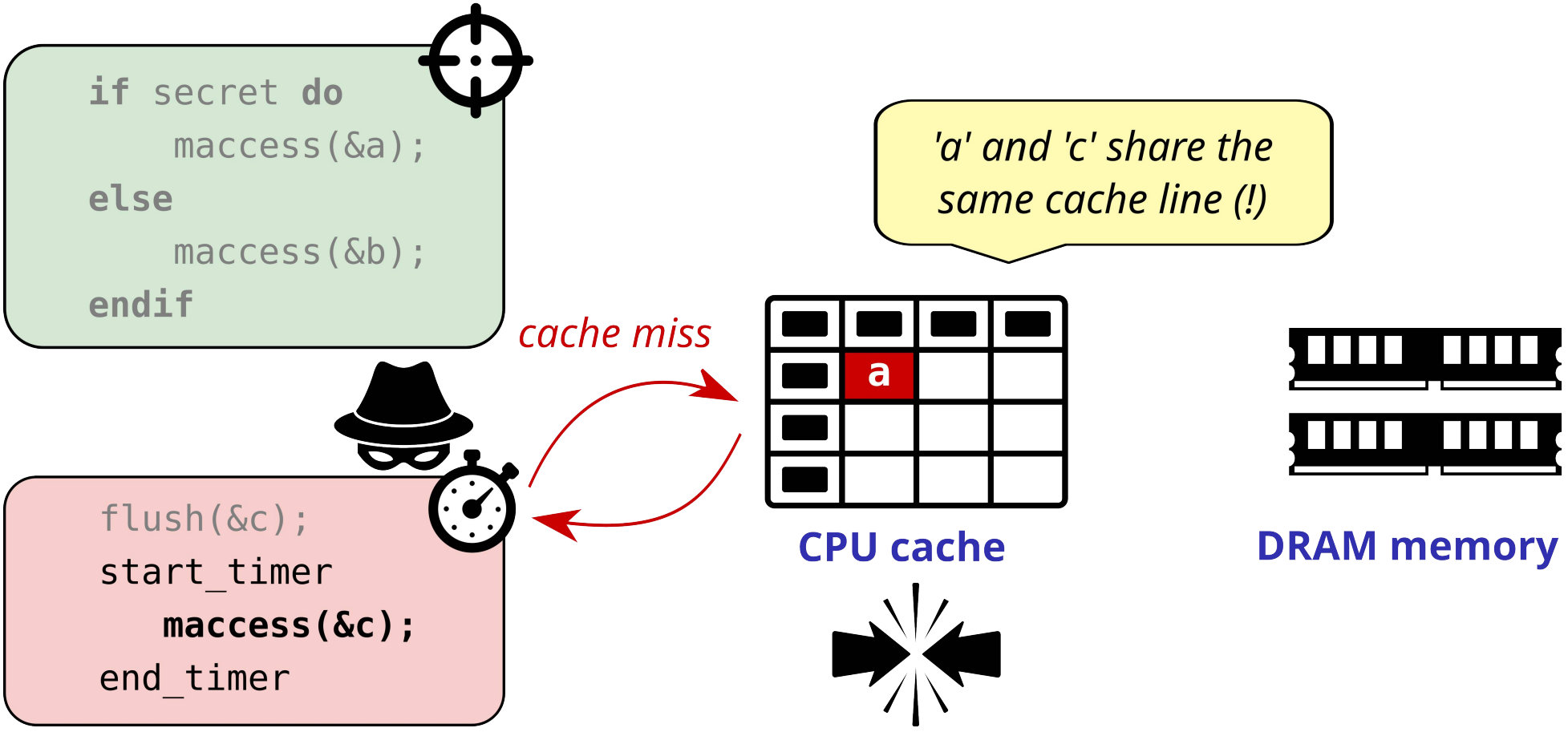
DRAM memory



Prime+Probe: Generalized Cache Timing Attacks

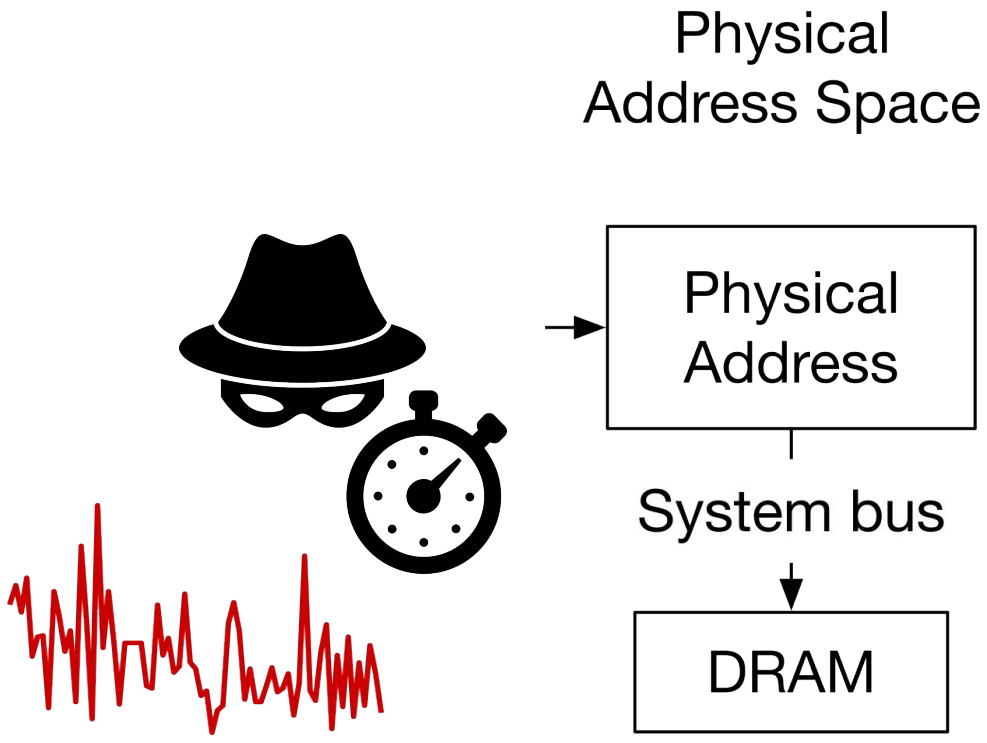


Prime+Probe: Generalized Cache Timing Attacks

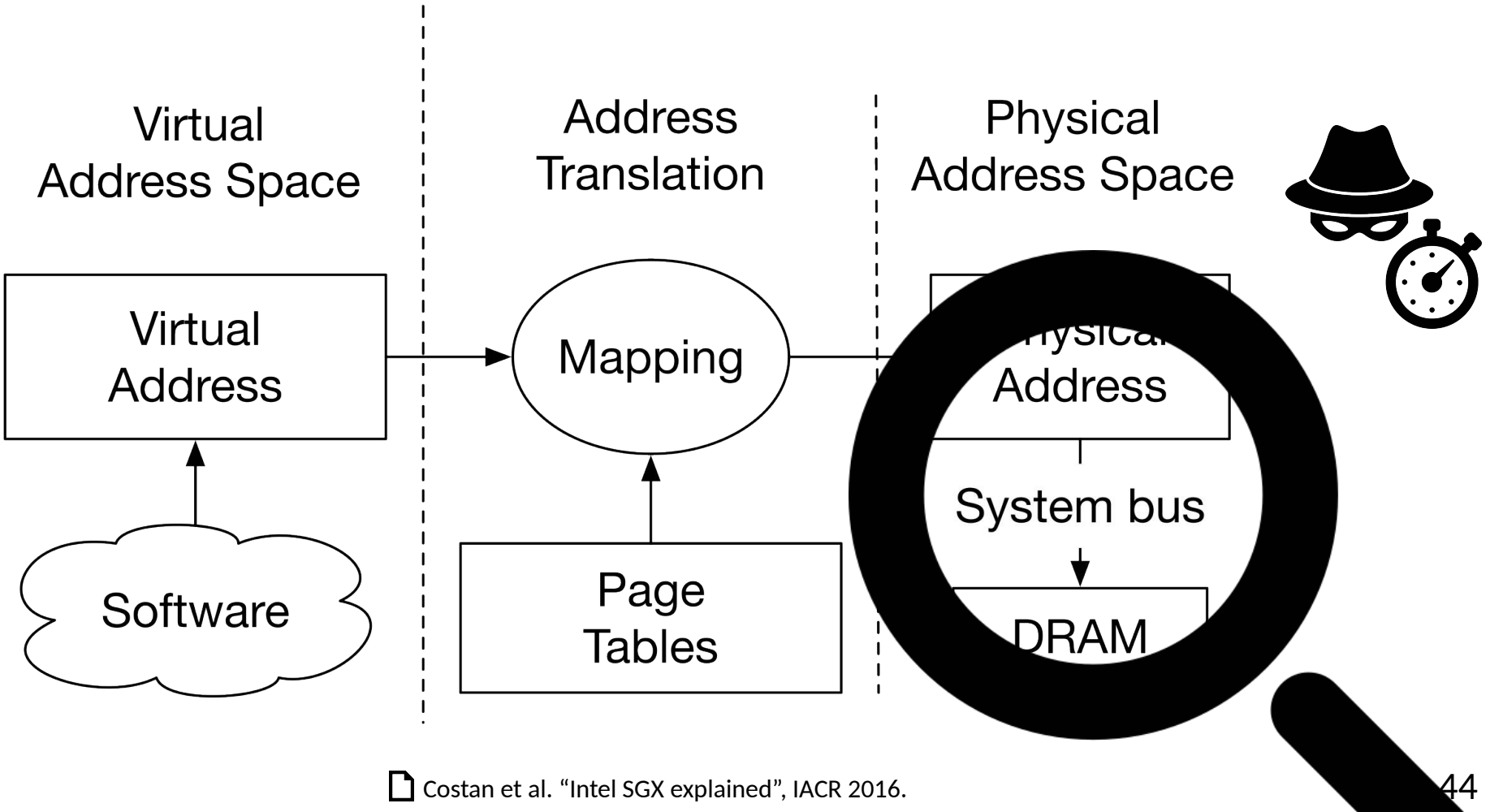


Challenge #1: Deterministic Spatial Resolution?

Background: The Physical Address Space

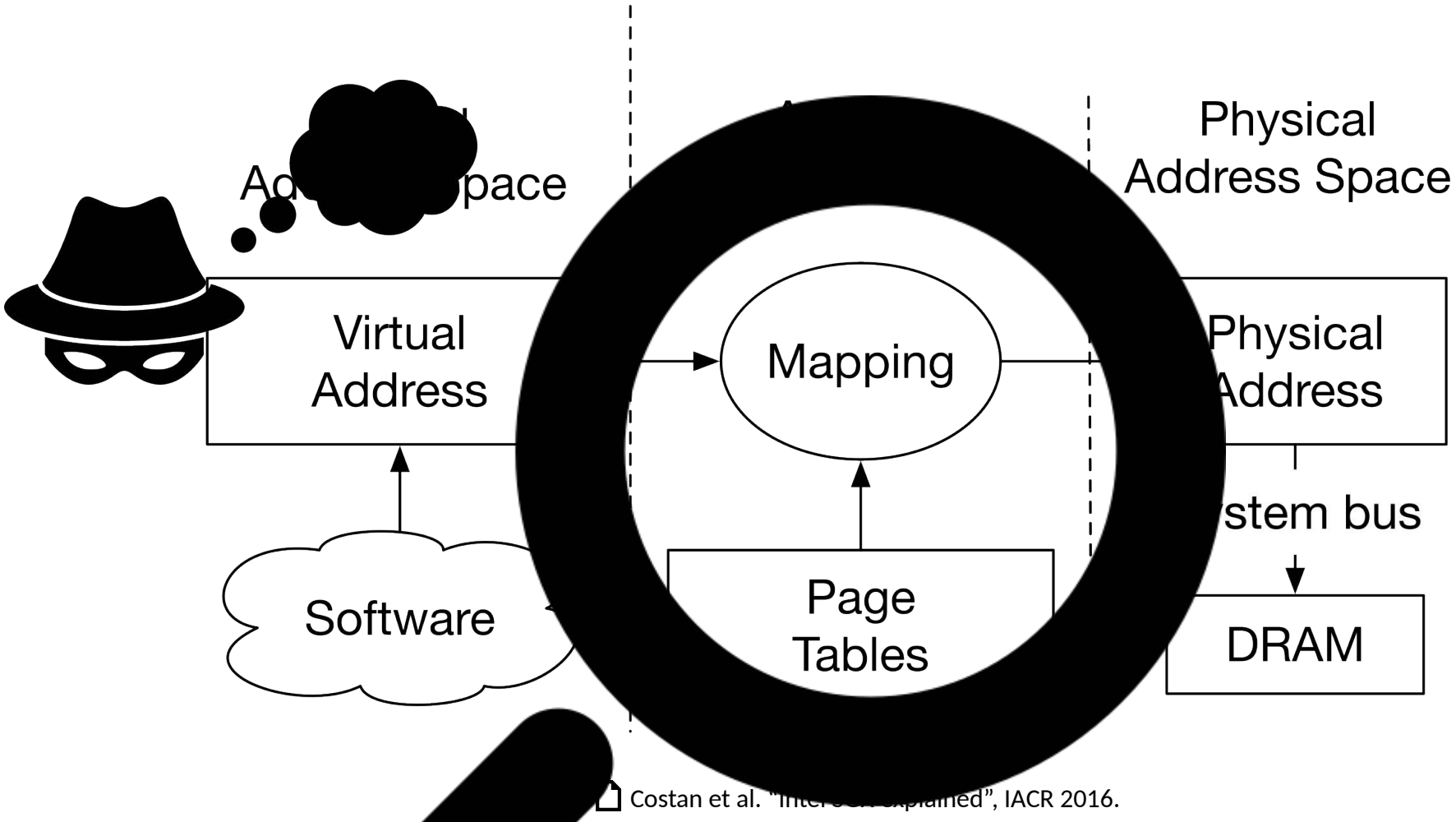


Background: The Virtual Memory Abstraction



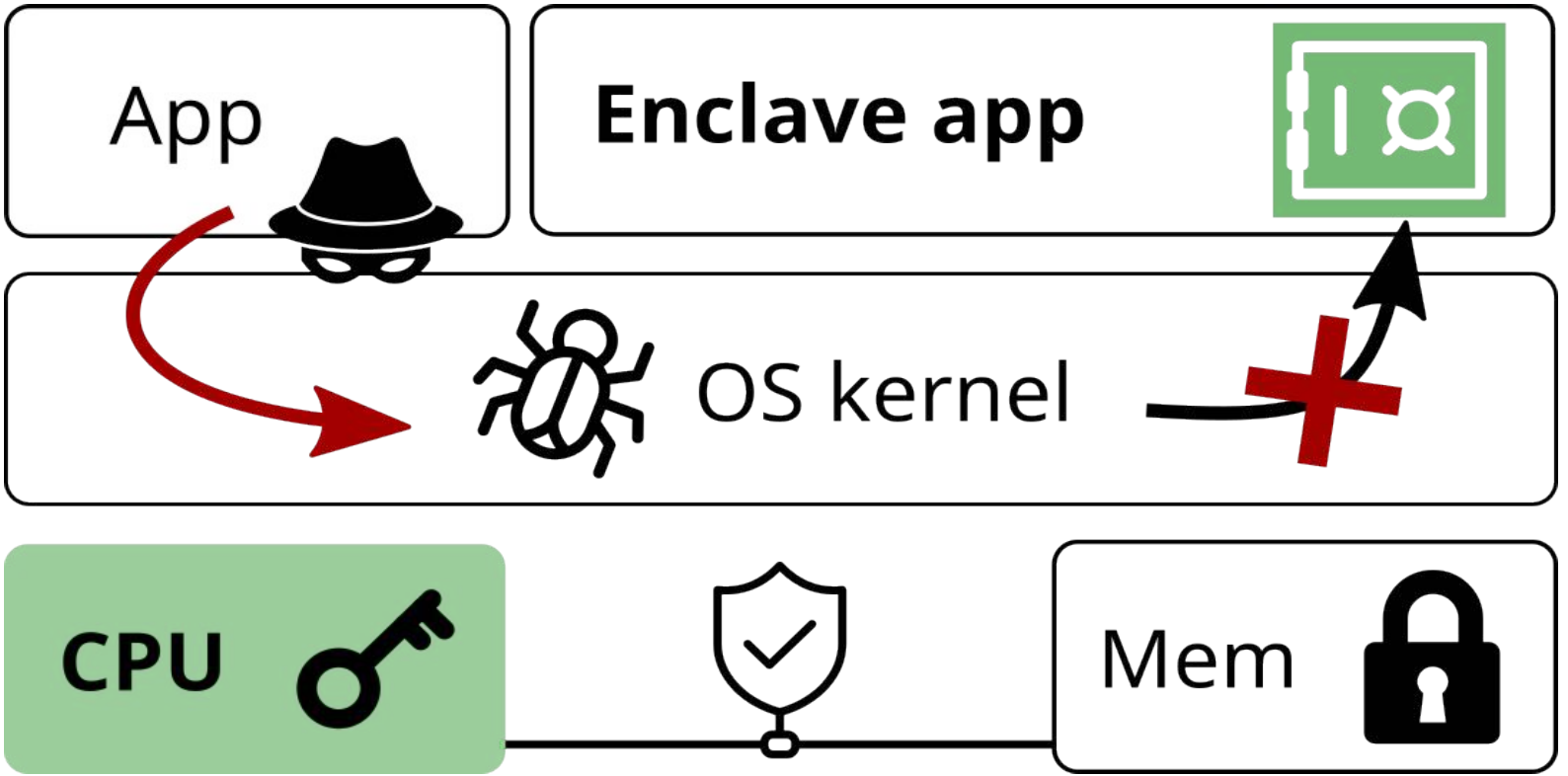
Costan et al. “Intel SGX explained”, IACR 2016.

Background: The Virtual Memory Abstraction



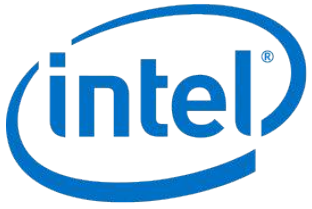
Costan et al. "Intel SGX Explained", IACR 2016.

Confidential Computing: Reducing Attack Surface



Trusted execution: Hardware-level isolation and attestation

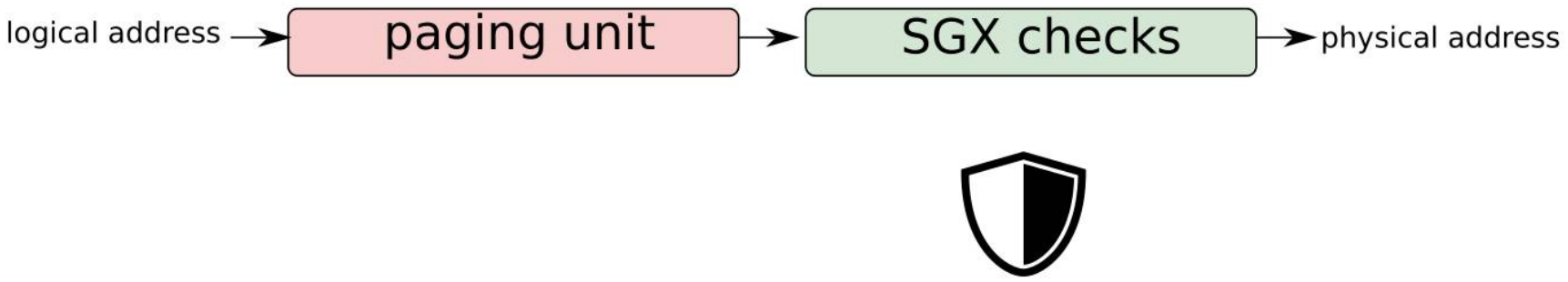
The Rise of Trusted Execution Environments (TEEs)



- 2004: ARM TrustZone
- 2015: **Intel Software Guard Extensions (SGX)**
- 2016: AMD Secure Encrypted Virtualization (SEV)
- 2018: IBM Protected Execution Facility (PEF)
- 2020: AMD SEV with Secure Nested Paging (SEV-SNP)
- 2022: Intel Trust Domain Extensions (TDX)
- 2023: ARM Confidential Compute Architecture (CCA)
- 2024: NVIDIA Confidential Computing

TEEs are here to stay...

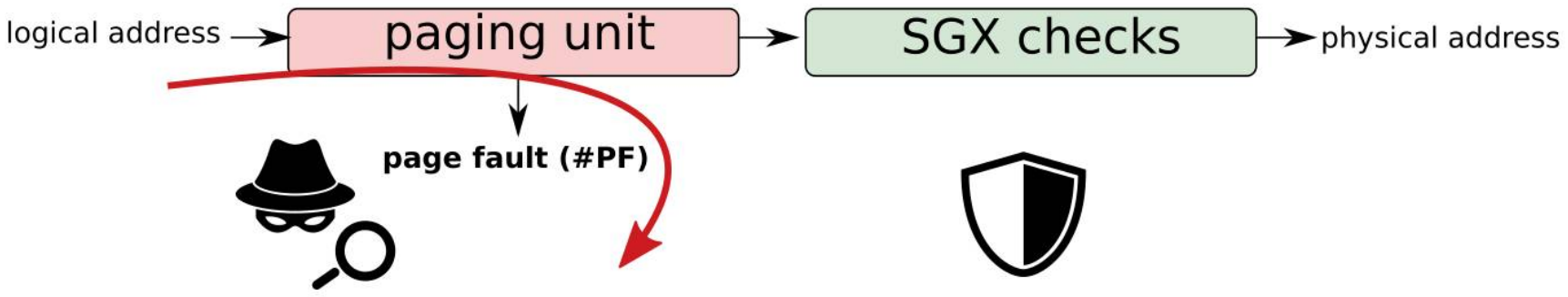
Idea: Page Faults as a Side Channel



SGX machinery protects against direct address remapping attacks

Xu et al. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”, IEEE S&P 2015.

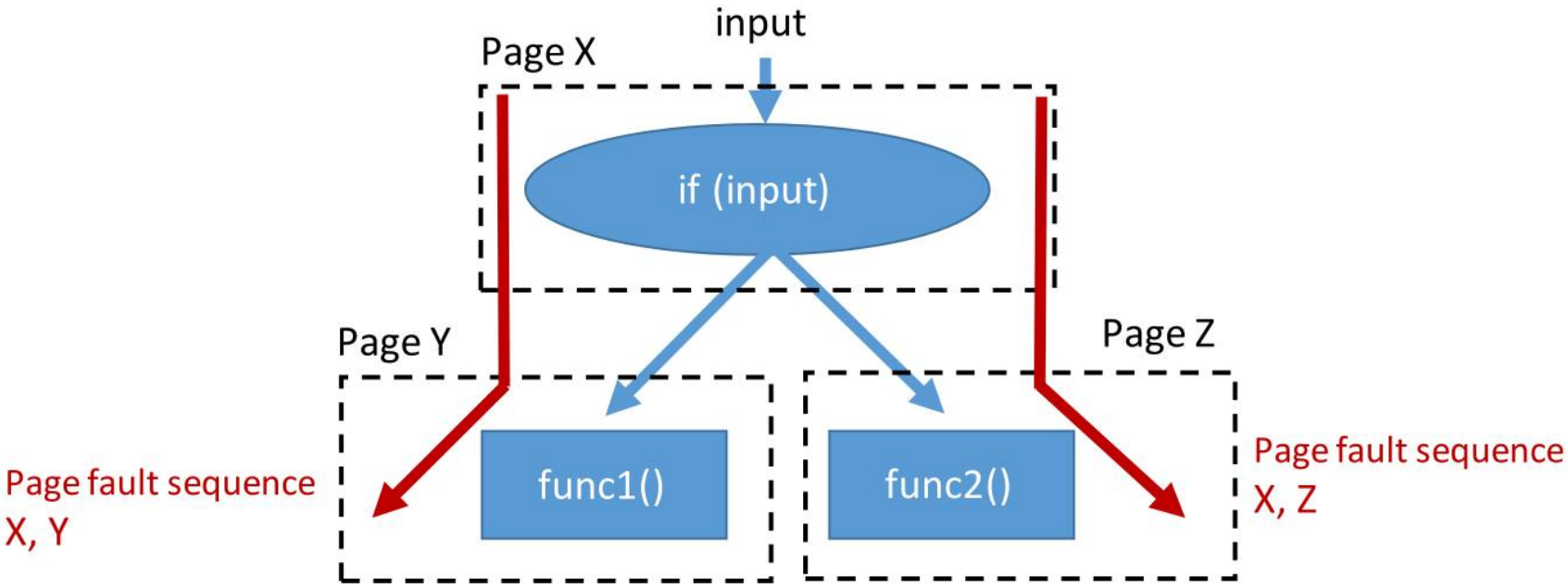
Idea: Page Faults as a Side Channel



. . . but untrusted address translation may **fault(!)**

Xu et al. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”, IEEE S&P 2015.

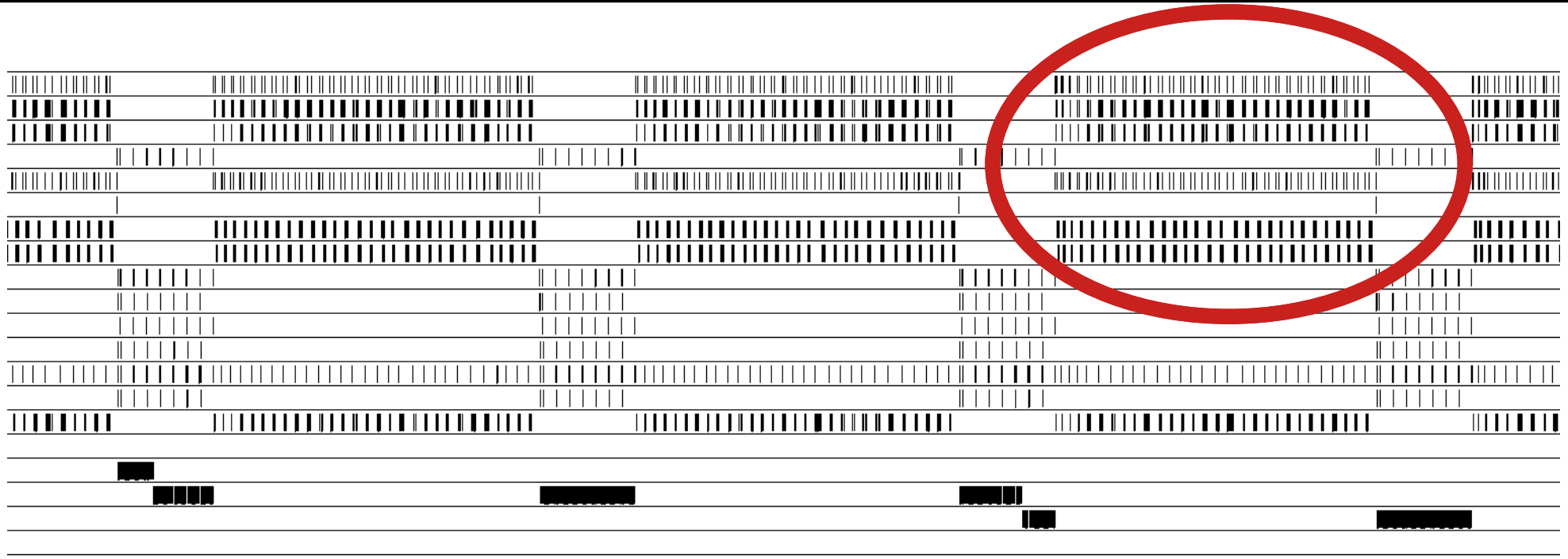
Intel SGX: Page Faults as a Side Channel



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

⇒ Page fault traces leak **private control data/flow**

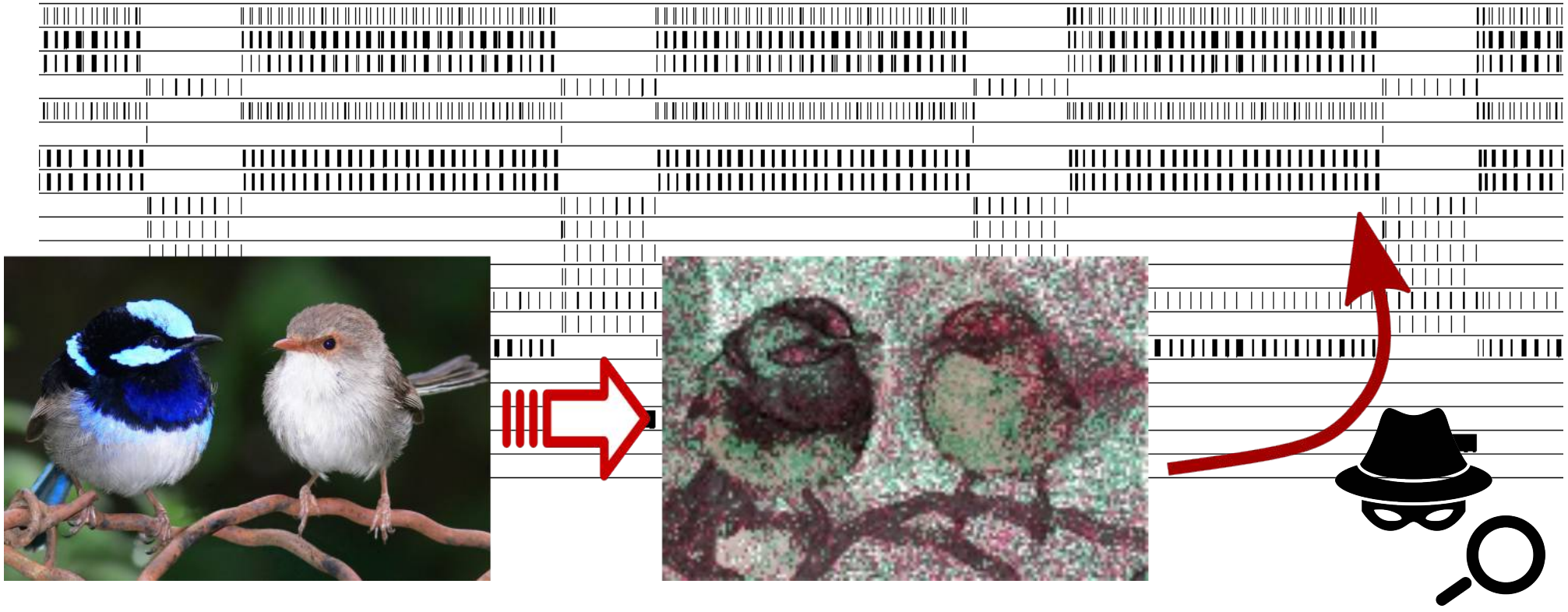
Spatial Resolution: Page-Granular Memory Access Traces



Detailed trace of (coarse-grained) code and data accesses over time…

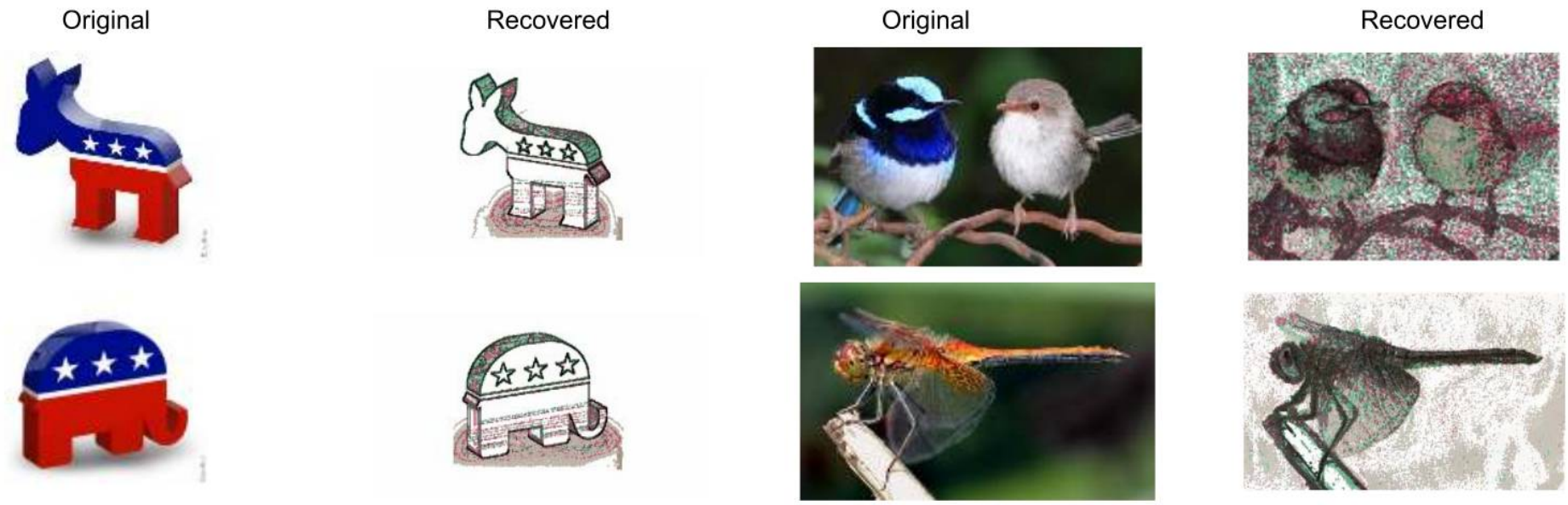
Vanoverloop et. al “TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms”, USENIX’25.

Spatial Resolution: Page-Granular Memory Access Traces



Vanoverloop et. al “TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms”, USENIX’25.

Spatial Resolution: Page-Granular Memory Access Traces

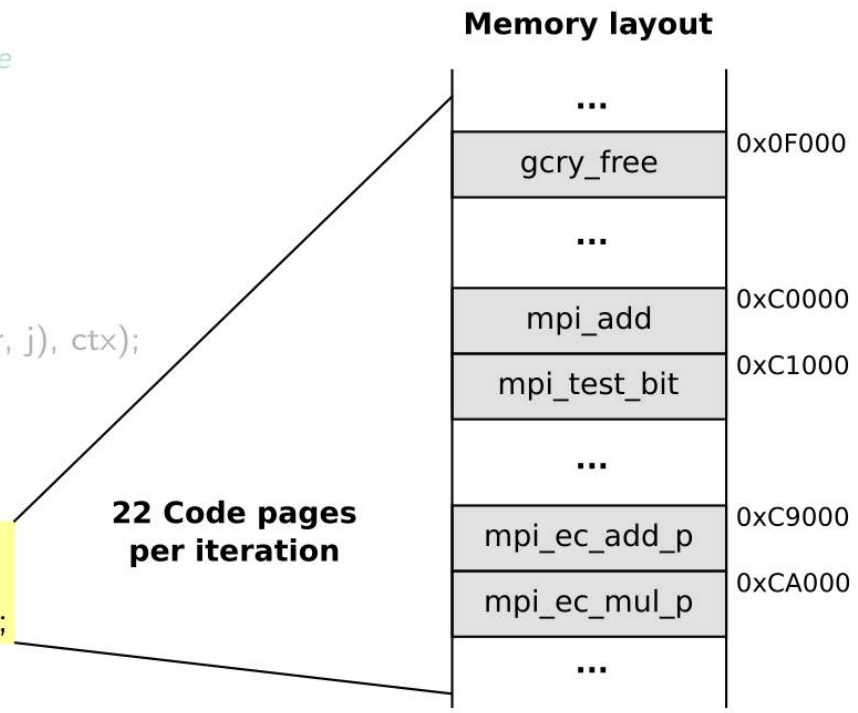


Xu et al.: "Controlled-channel attacks: Deterministic side channels for untrusted operating systems", Oakland 2015.

. . . but **many faults** and a coarse-grained **4 KiB granularity**

Attacking Libgcrypt EdDSA (Simplified)

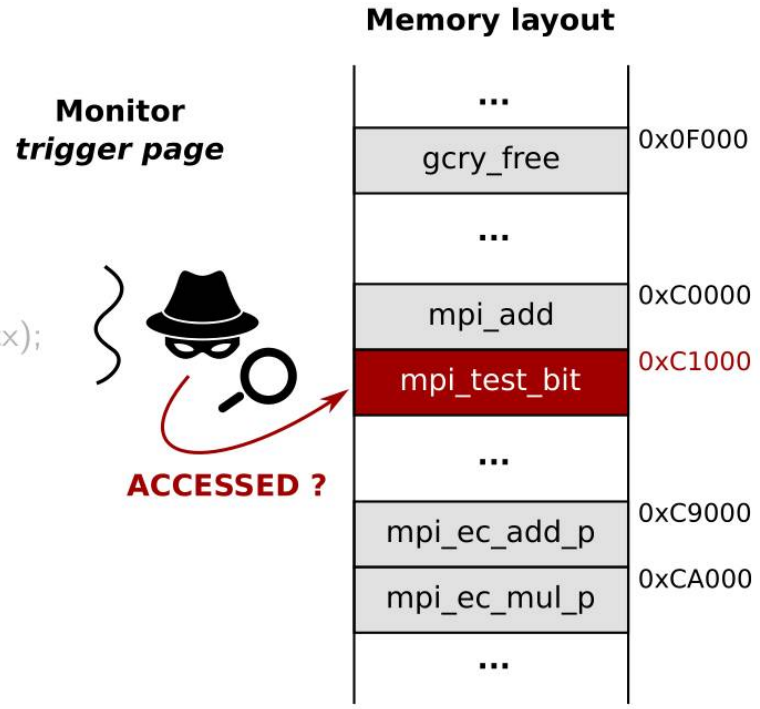
```
1  if (mpi_is_secure (scalar)) {
2      /* If SCALAR is in secure memory we assume that it is the
3         secret key we use constant time operation. */
4      point_init (&tmppnt);
5
6      for (j=nbits-1; j >= 0; j--) {
7          _gcry_mpi_ec_dup_point (result, result, ctx);
8          _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
9          point_swap_cond (result, &tmppnt, mpi_test_bit (scalar, j), ctx);
10     }
11     point_free (&tmppnt);
12 } else {
13     for (j=nbits-1; j >= 0; j--) {
14         _gcry_mpi_ec_dup_point (result, result, ctx);
15         if (mpi_test_bit (scalar, j))
16             _gcry_mpi_ec_add_points (result, result, point, ctx);
17     }
18 }
```



Van Bulck et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution", USENIX 2017.

Attacking Libgcrypt EdDSA (Simplified)

```
1  if (mpi_is_secure (scalar)) {
2      /* If SCALAR is in secure memory we assume that it is the
3         secret key we use constant time operation. */
4      point_init (&tmppnt);
5
6      for (j=nbits-1; j >= 0; j--) {
7          _gcry_mpi_ec_dup_point (result, result, ctx);
8          _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
9          point_swap_cond (result, &tmppnt, mpi_test_bit (scalar, j), ctx);
10     }
11     point_free (&tmppnt);
12 } else {
13     for (j=nbits-1; j >= 0; j--) {
14         _gcry_mpi_ec_dup_point (result, result, ctx);
15         if (mpi_test_bit (scalar, j))
16             _gcry_mpi_ec_add_points (result, result, point, ctx);
17     }
18 }
```



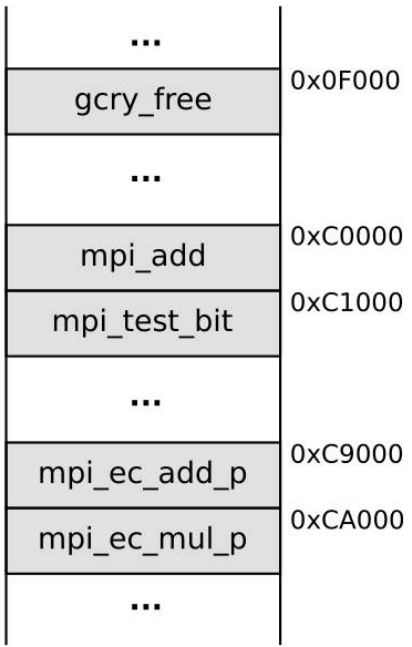
Van Bulck et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution", USENIX 2017.

Attacking Libgcrypt EdDSA (Simplified)

```
1  if (mpi_is_secure (scalar)) {
2      /* If SCALAR is in secure memory we assume that it is the
3         secret key we use constant time operation. */
4      point_init (&tmppnt);
5
6      for (j=nbits-1; j >= 0; j--) {
7          _gcry_mpi_ec_dup_point (result, result, ctx);
8          _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
9          point_swap_cond (result, &tmppnt, mpi_test_bit (scalar, j), ctx);
10     }
11     point_free (&tmppnt);
12 } else {
13     for (j=nbits-1; j >= 0; j--) {
14         _gcry_mpi_ec_dup_point (result, result, ctx);
15         if (mpi_test_bit (scalar, j))
16             _gcry_mpi_ec_add_points (result, result, point, ctx);
17     }
18 }
```

INTERRUPT

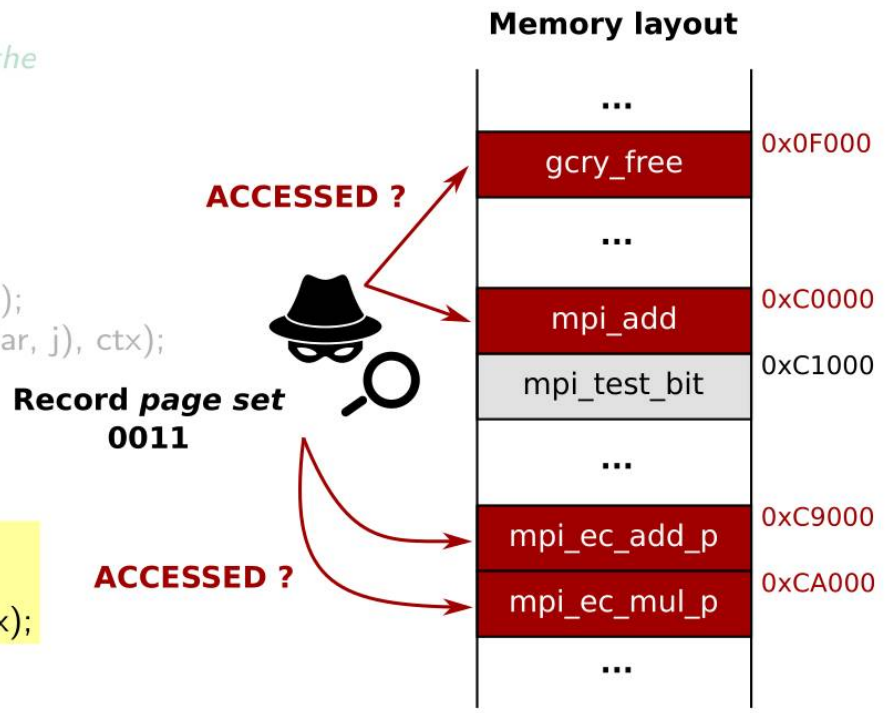
Memory layout



Van Bulck et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution", USENIX 2017.

Attacking Libgcrypt EdDSA (Simplified)

```
1  if (mpi_is_secure (scalar)) {
2      /* If SCALAR is in secure memory we assume that it is the
3         secret key we use constant time operation. */
4      point_init (&tmppnt);
5
6      for (j=nbits-1; j >= 0; j--) {
7          _gcry_mpi_ec_dup_point (result, result, ctx);
8          _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
9          point_swap_cond (result, &tmppnt, mpi_test_bit (scalar, j), ctx);
10     }
11     point_free (&tmppnt);
12 } else {
13     for (j=nbits-1; j >= 0; j--) {
14         _gcry_mpi_ec_dup_point (result, result, ctx);
15         if (mpi_test_bit (scalar, j))
16             _gcry_mpi_ec_add_points (result, result, point, ctx);
17     }
18 }
```



Van Bulck et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution", USENIX 2017.

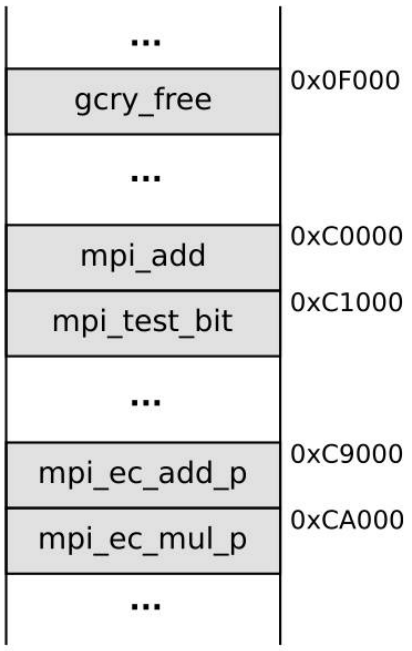
Attacking Libgcrypt EdDSA (Simplified)

```
1  if (mpi_is_secure (scalar)) {
2      /* If SCALAR is in secure memory we assume that it is the
3         secret key we use constant time operation. */
4      point_init (&tmppnt);
5
6      for (j=nbits-1; j >= 0; j--) {
7          _gcry_mpi_ec_dup_point (result, result, ctx);
8          _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
9          point_swap_cond (result, &tmppnt, mpi_test_bit (scalar, j), ctx);
10     }
11     point_free (&tmppnt);
12 } else {
13     for (j=nbits-1; j >= 0; j--) {
14         _gcry_mpi_ec_dup_point (result, result, ctx);
15         if (mpi_test_bit (scalar, j))
16             _gcry_mpi_ec_add_points (result, result, point, ctx);
17     }
18 }
```

RESUME

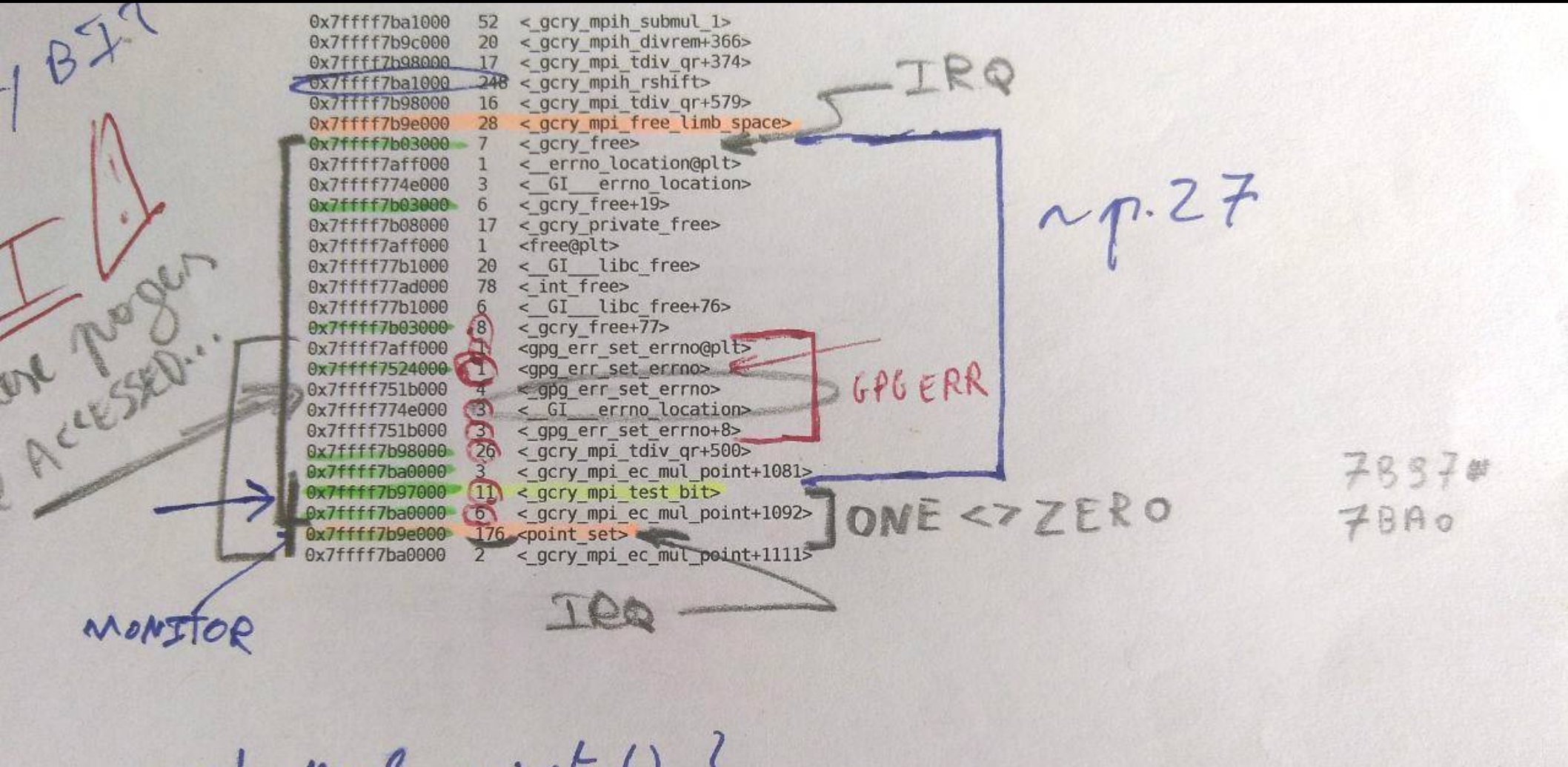
Full 512-bit key recovery, single run

Memory layout



Van Bulck et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution", USENIX 2017.

Side-Channel Analysis: From Metadata Patterns to Secrets



Challenge #2: Limited Temporal Resolution?

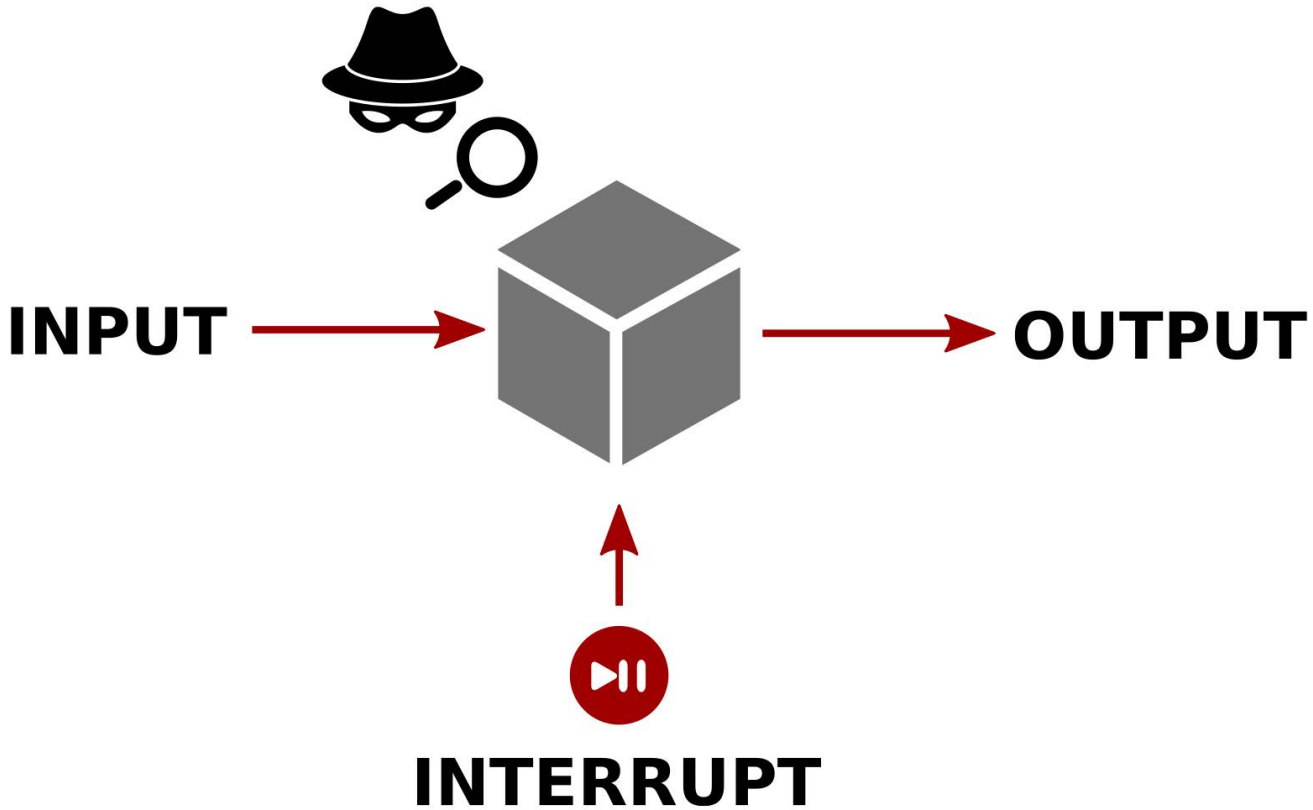
Challenge: Side-Channel Sampling Rate

Slow
shutter speed

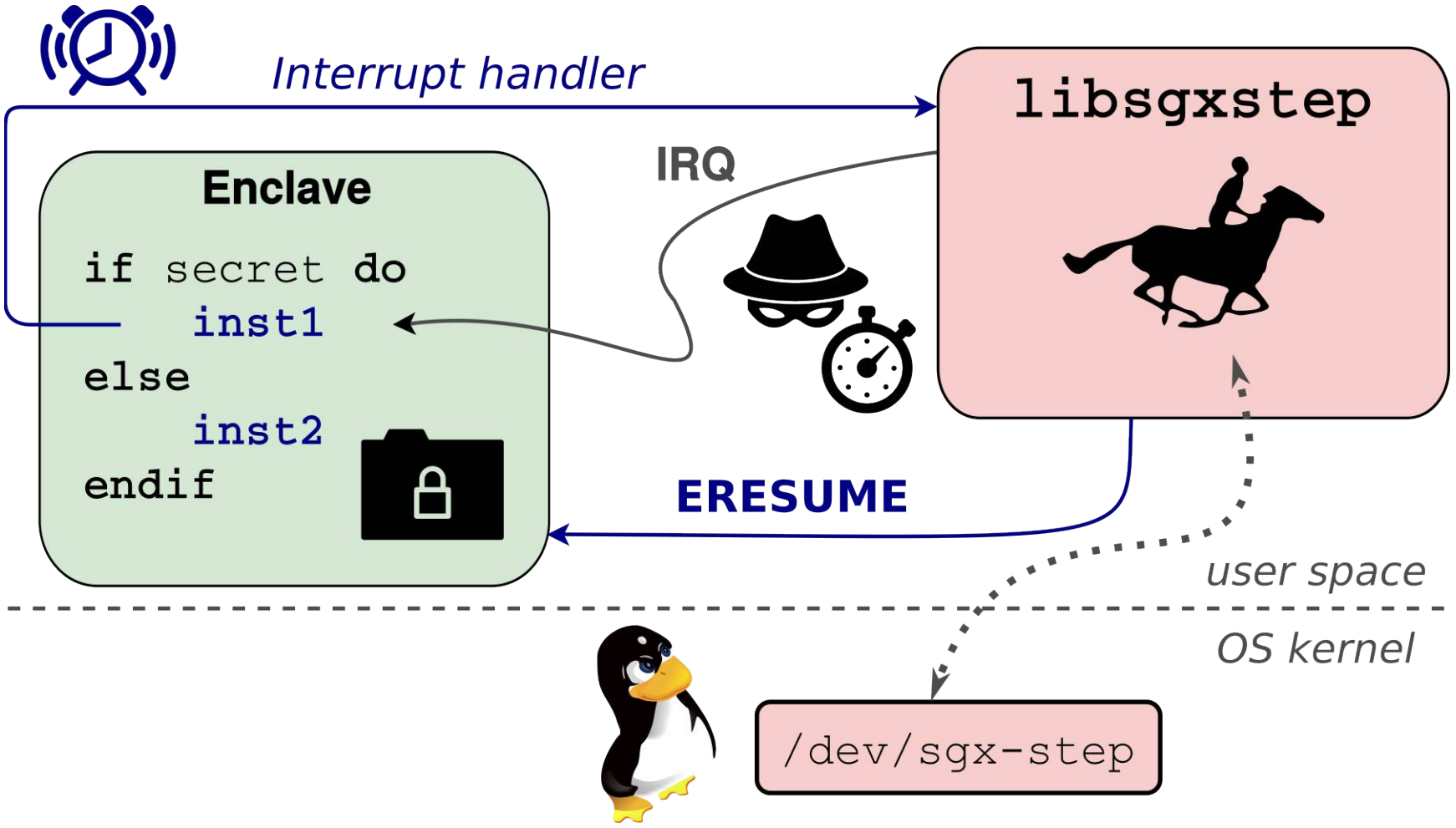
Medium
shutter speed

Fast
shutter speed

SGX-Step: Executing Enclaves one Instruction at a Time



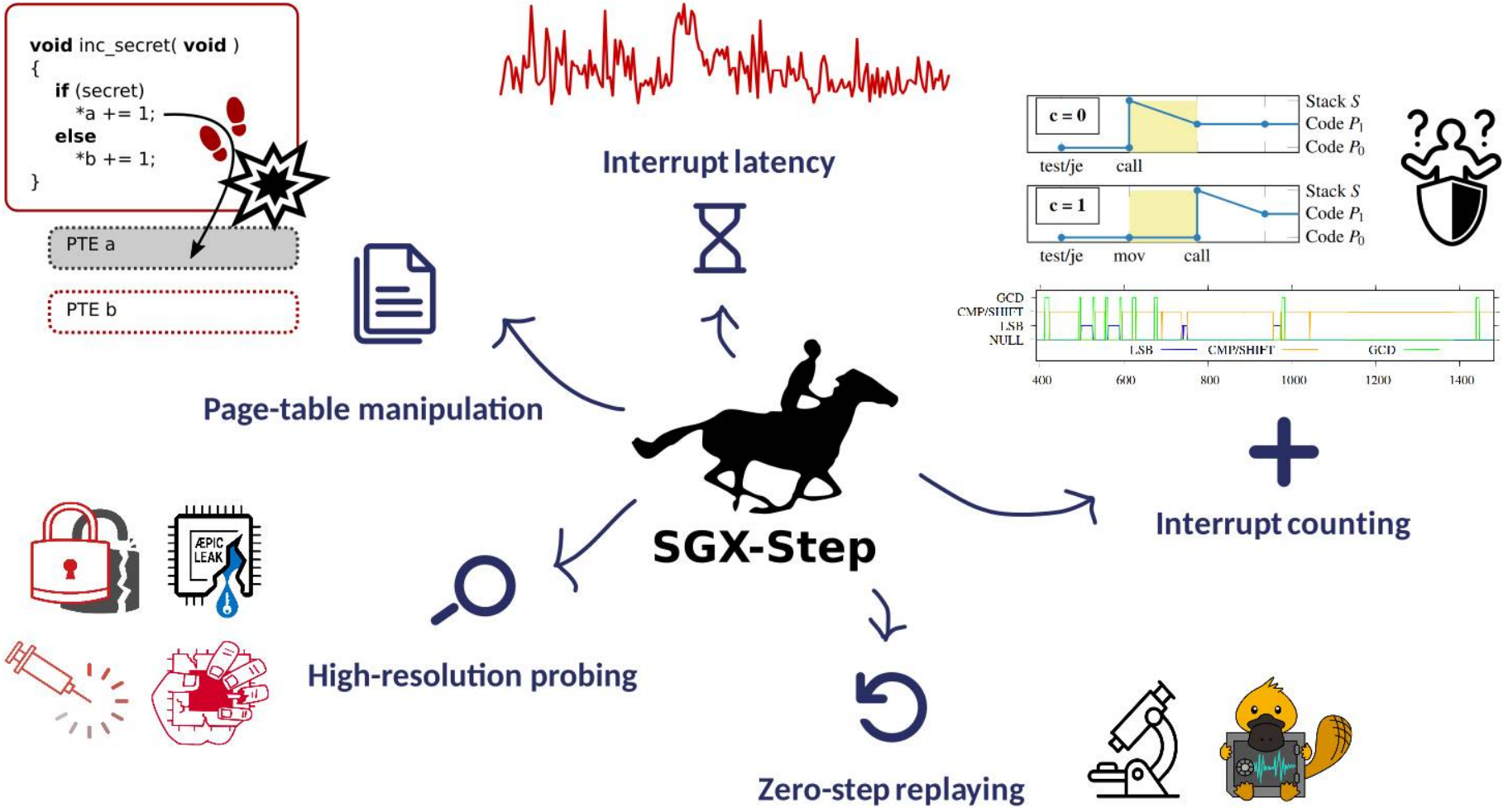
SGX-Step: Executing Enclaves one Instruction at a Time



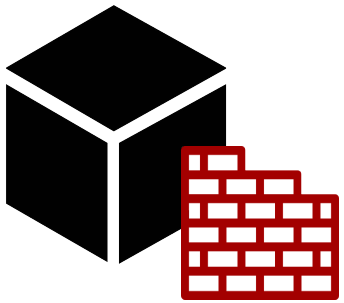
SGX-Step Demo: Single-Stepping Password Comparison

```
[idt.c] IDT[ 45] @0x7f83225492d0 = 0x556b4424b000 (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 0x7f8322548000
[apic.c] APIC_ID=2000000; LVTT=400ec; TDCR=0
[apic.c] APIC timer one-shot mode with division 2 (lvtt=2d/tdcr=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 2413/2336 IRQs (AEP/IDT)
jo@breuer:~/sgx-step-demo$
```

SGX-Step: A Versatile Open-Source Attack Framework

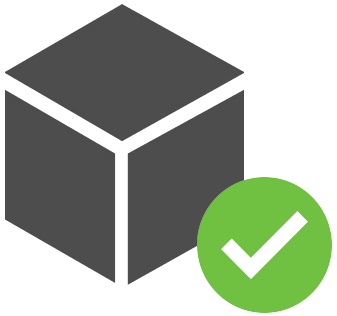


Road Map



System Model

Timing Attacks



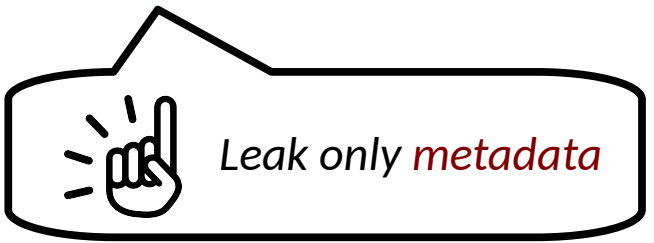
Constant Time

Vulnerable Patterns: Secret-Dependent Code/Data Accesses

```
1 void secret_vote(char candidate)
2 {
3     if (candidate == 'a')
4         vote_candidate_a();
5     else
6         vote_candidate_b();
7 }
```

```
1 int secret_lookup(int s)
2 {
3     if (s > 0 && s < ARRAY_LEN)
4         return array[s];
5     return -1;
6
7 }
```

What are the ways for adversaries to create an "oracle" for all victim code+data memory access sequences?



Software Mitigation: “Constant-Time” Programming

```
1 void check_pwd(char *input)
2 {
3     for (int i=0; i < PWD_LEN; i++)
4       if (input[i] != pwd[i])
5           return 0;
6
7     return 1;
8 }
```

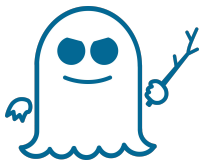
```
1 void check_pwd(char *input)
2 {
3     int rv = 0x0;
4     for (int i=0; i < PWD_LEN; i++)
5         rv |= input[i] ^ pwd[i];
6
7     return (result == 0);
8 }
```

Rewrite program such that execution time does not depend on secrets

→ manual, error-prone solution; side-channels are likely here to stay...

Software Mitigation: "Constant-Time" Programming

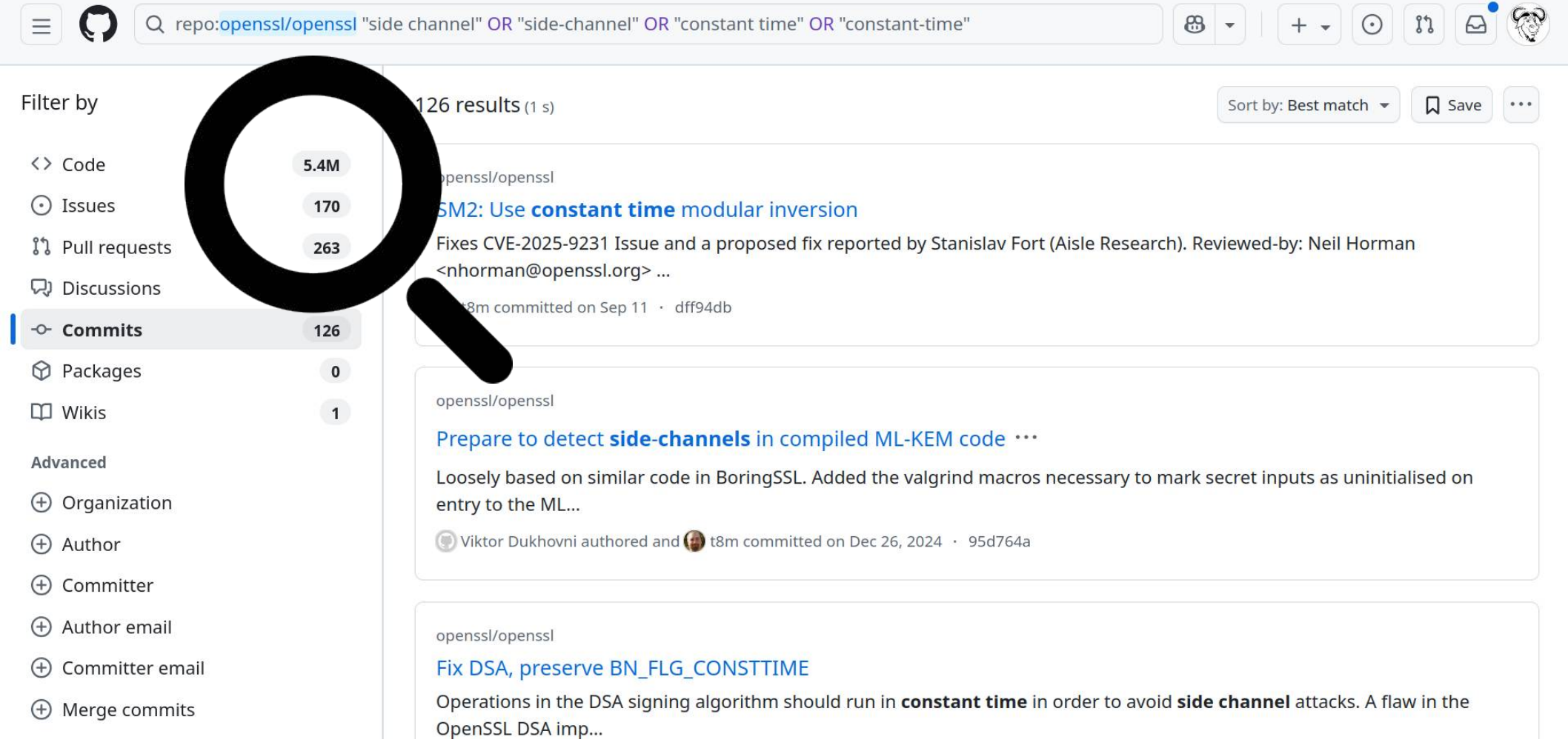
- **"Constant-time" leakage model:** Programmer makes sure that:
  - Control flow of the program does not depend on secrets
  - Memory addresses that are accessed do not depend on secrets
- State-of-the-art **crypto libraries** are (manually) implemented to be secure under this model [1,2]
- (But such programs still leak secrets on speculative processors)

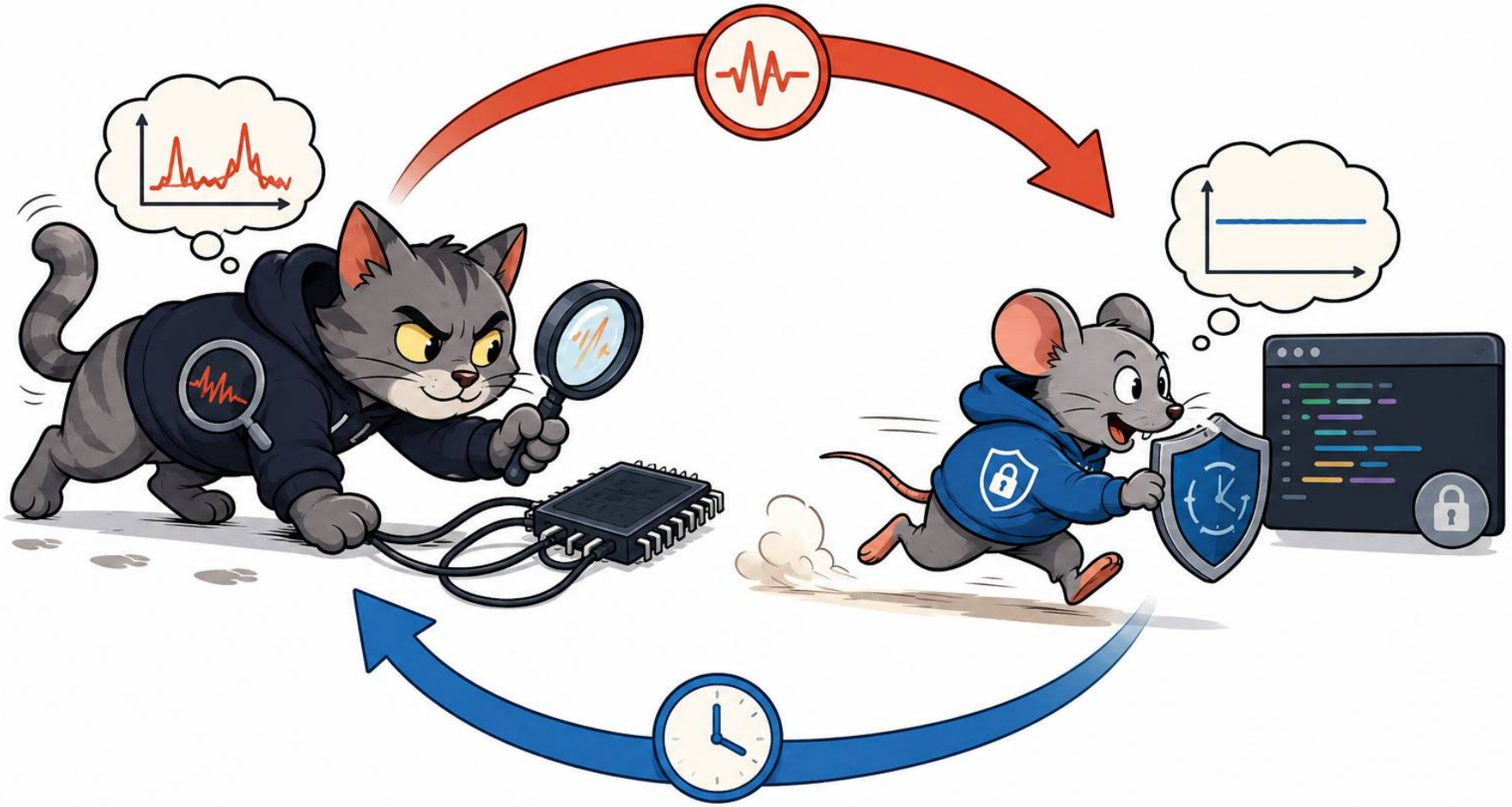


(1) Almeida et al., *Verifying Constant-Time Implementations*, USENIX Security 2016.

(2) Jancar et al., *"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks*, S&P 2022.

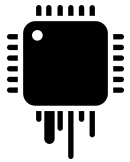
Example: Constant-Time Mitigations in OpenSSL



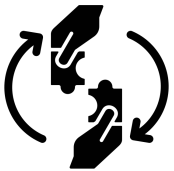


Conclusions and Take-Away

Microarchitectural attacks break architectural isolation “walls”



→ Need for **constant-time software** mitigations (crypto)



Scientific understanding driven by **attacker-defender race**