Secrets Beneath the Silicon:

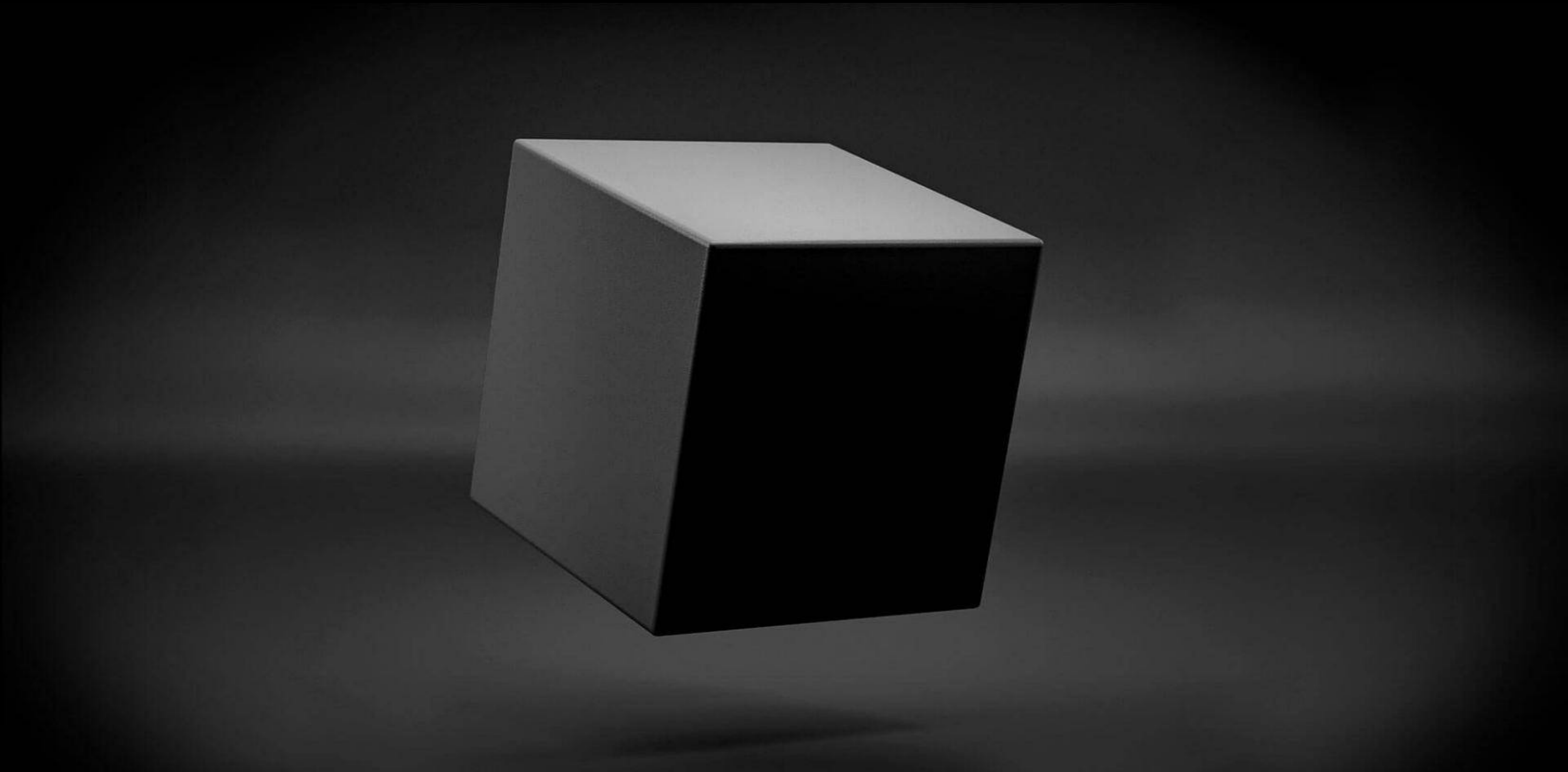
How Microarchitectural Attacks Break CPU Isolation

Jo Van Bulck

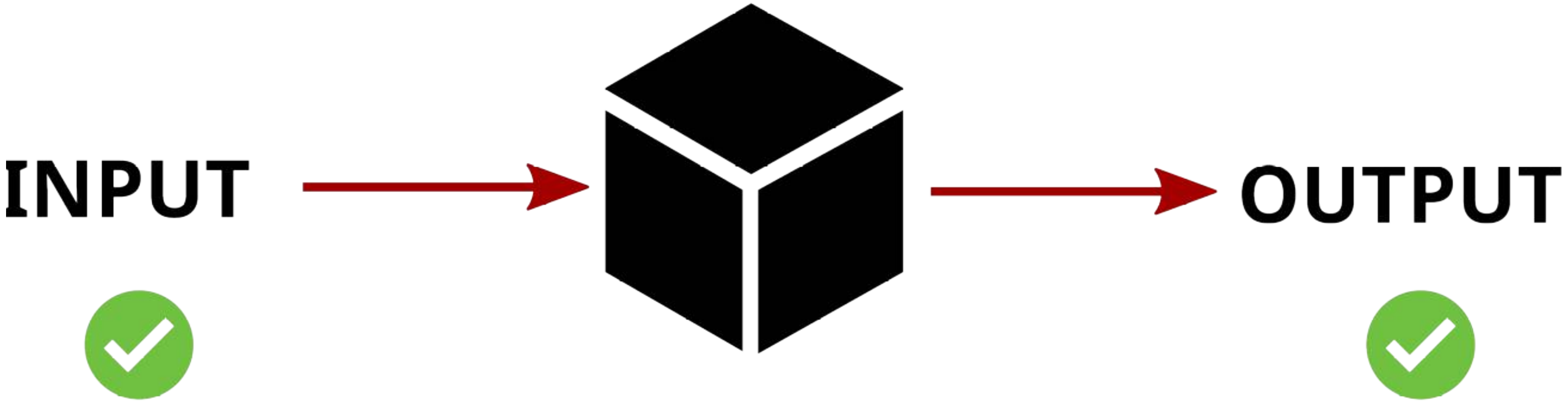
DistriNet, KU Leuven, Belgium   jo.vanbulck@cs.kuleuven.be   vanbulck.net

KU Leuven Semiconductor School, February 13

~50 Years of Systems Security in One Picture…

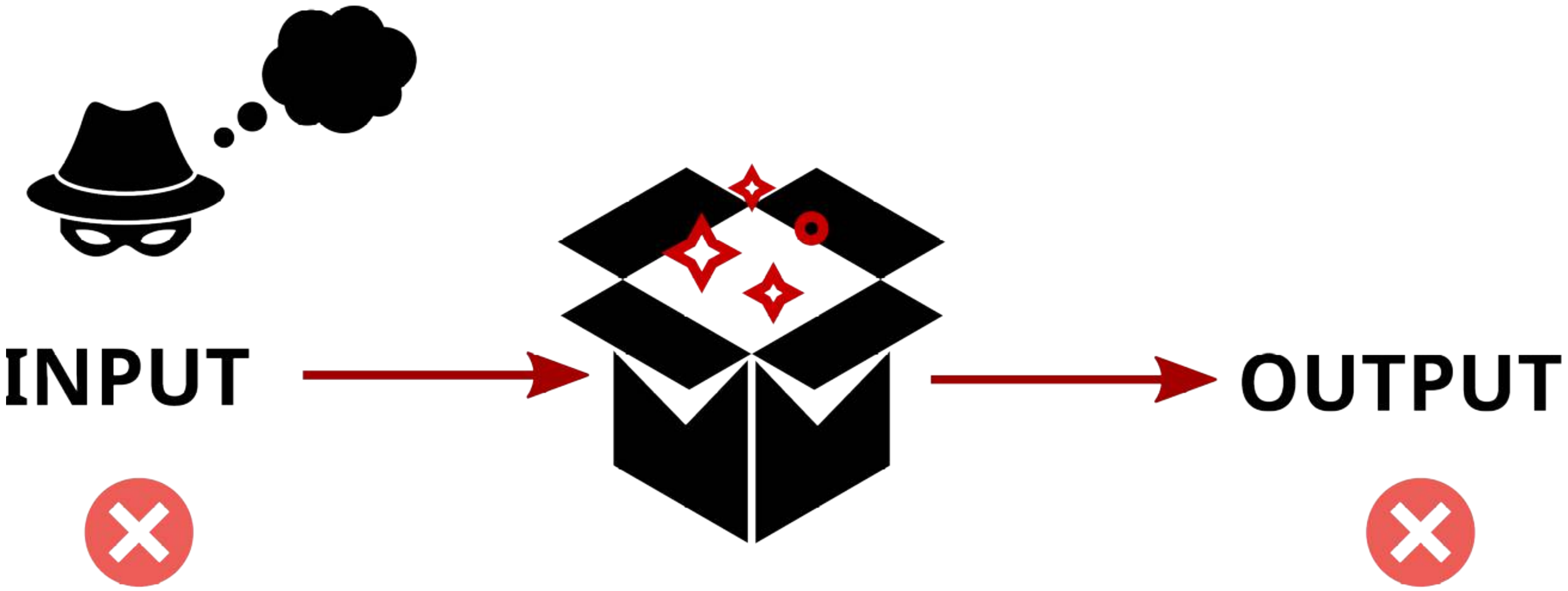


A primer on Software Security



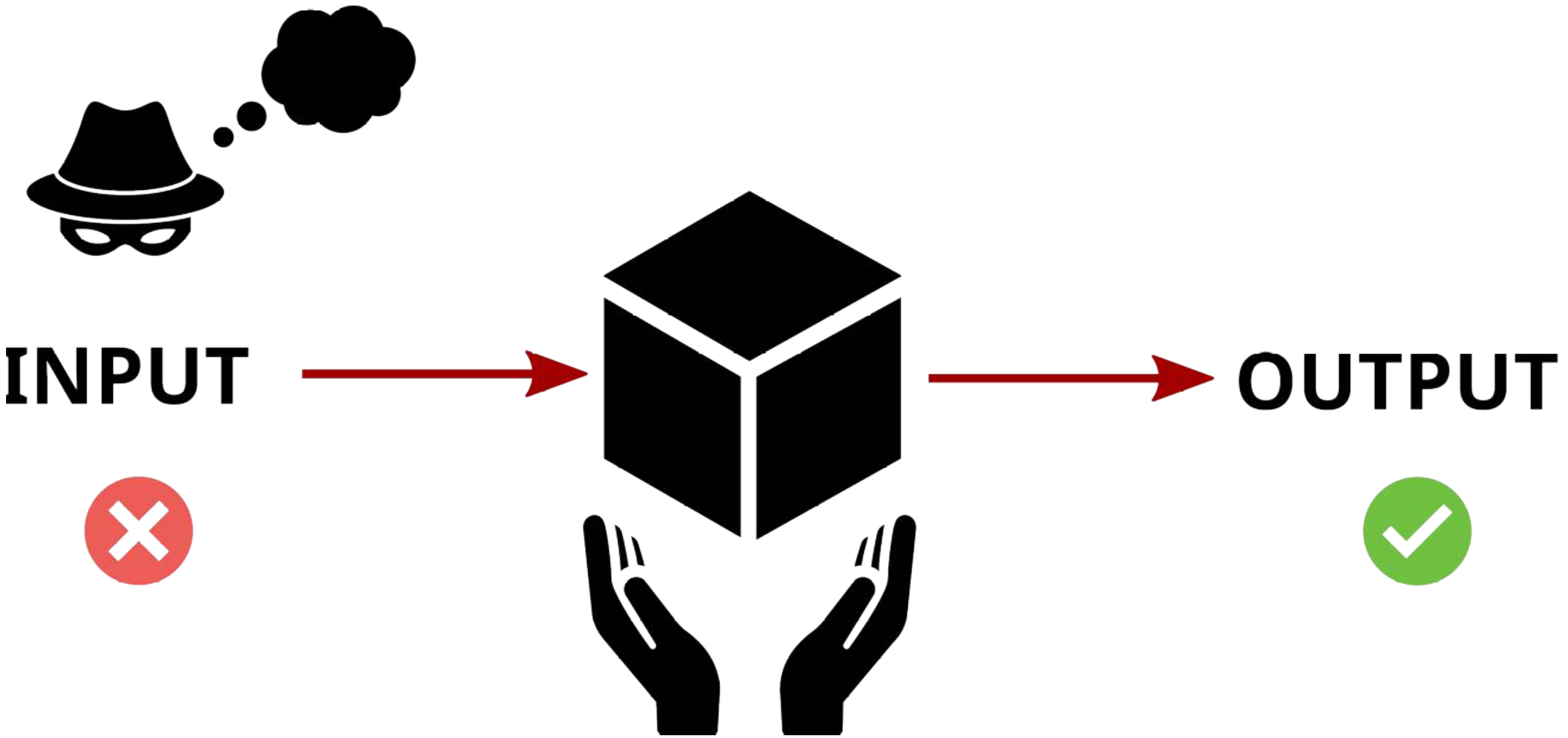
Secure program: Convert all input to *expected output*

A primer on Software Security (traditional attacks)



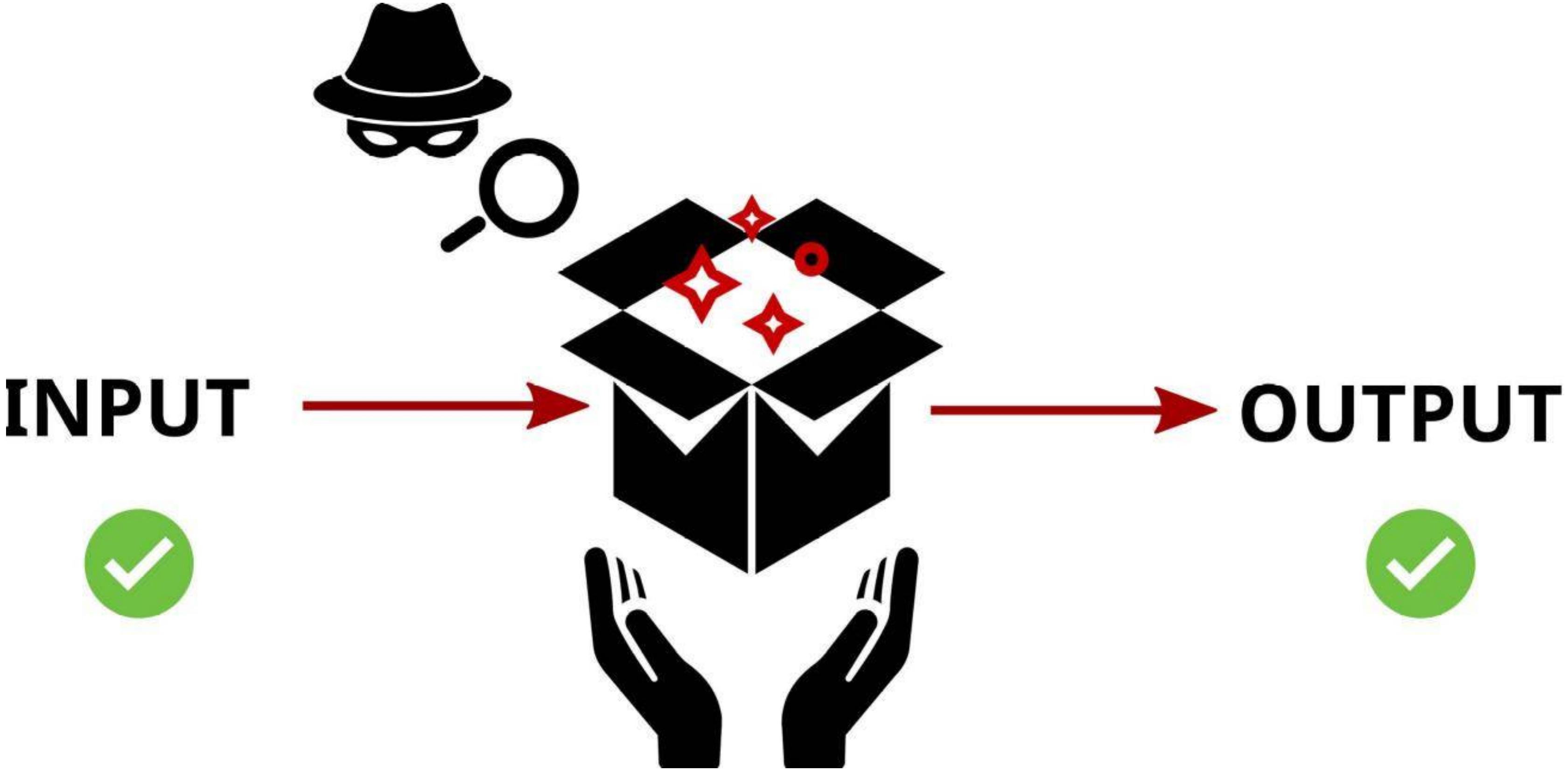
Buffer overflow vulnerabilities: trigger *unexpected behavior*

A primer on Software Security (traditional attacks)



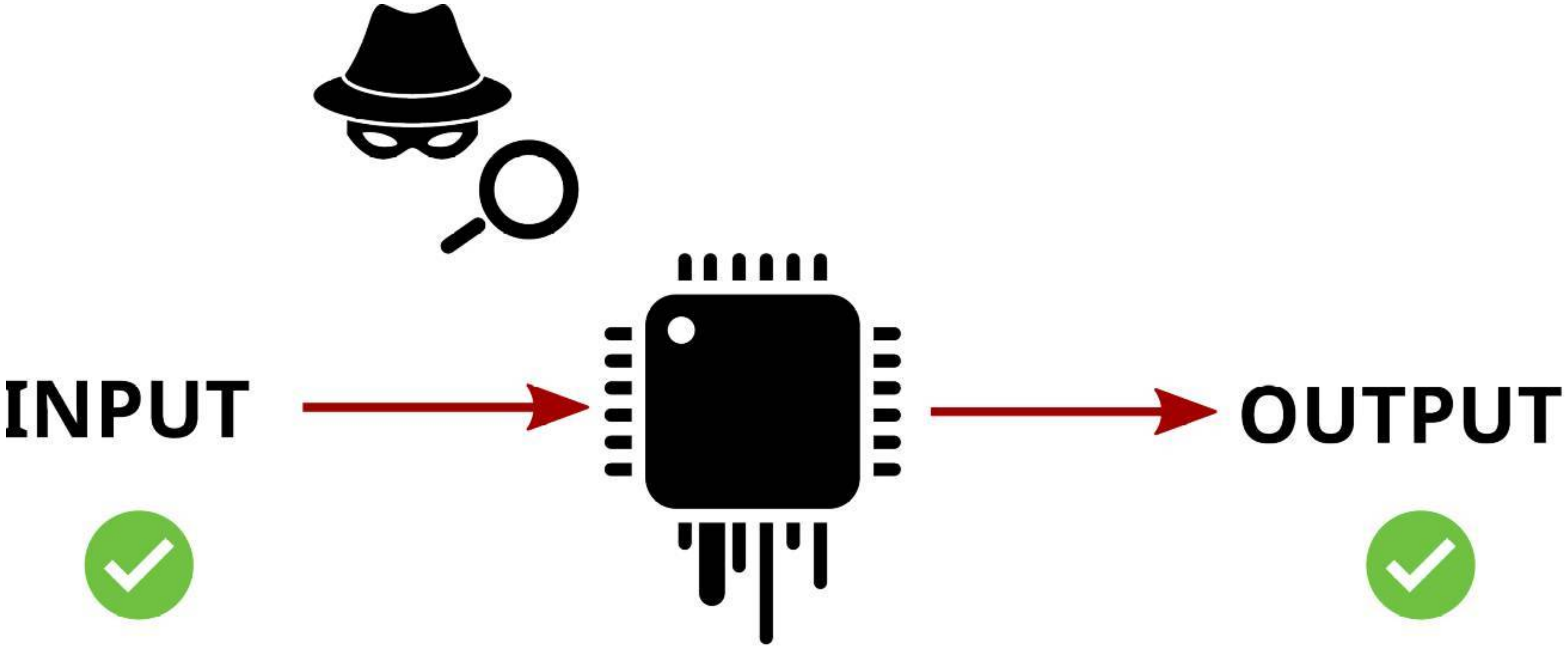
Safe languages & formal verification: Preserve *expected behavior*

A primer on Software Security (this lecture)



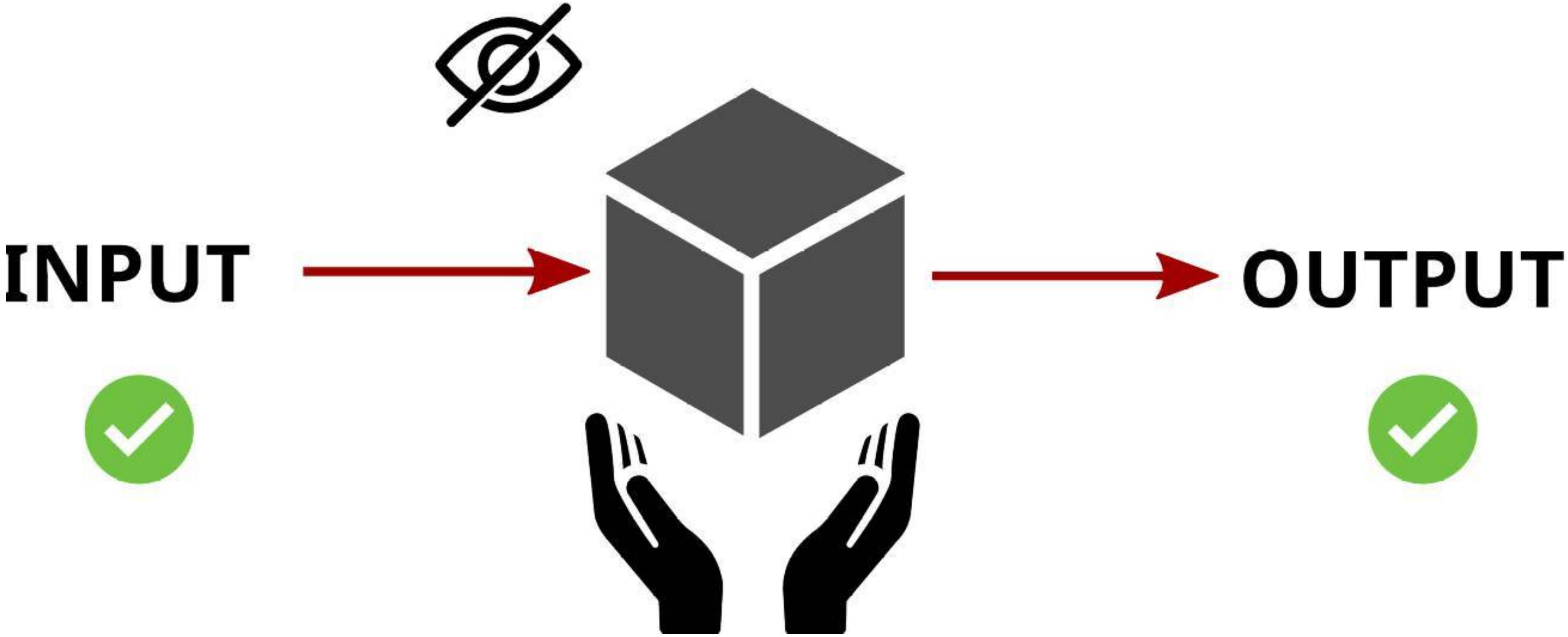
Side-channel attacks: Observe *side-effects* of the computation

A primer on Software Security (this lecture)



Microarchitectural leaks: *HW optimizations* do not respect *SW abstractions(!)*

A primer on Software Security (this lecture)



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

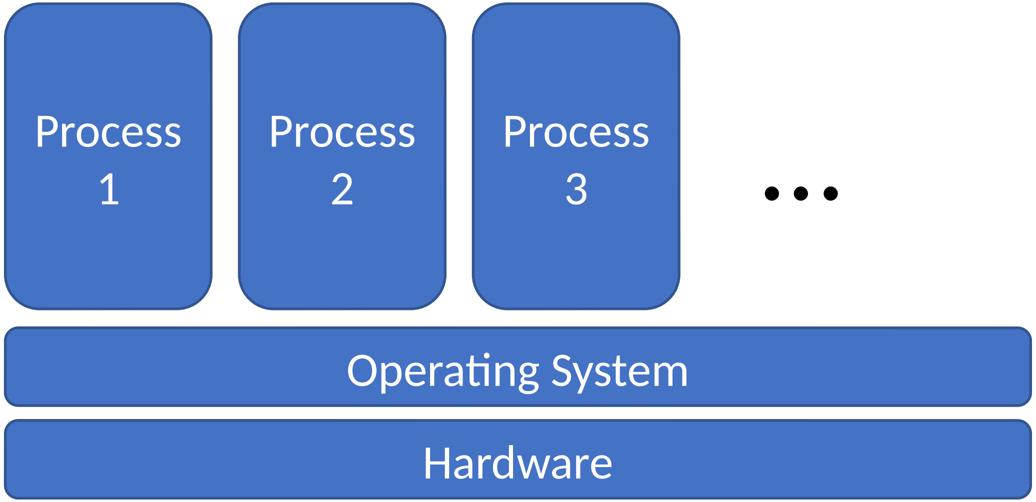
Introduction: The Setting of this Lecture

- **System model:**
  - A shared platform executing code from different stakeholders
- **Attacker model:**
  - Attacker can execute code on the same shared platform as the victim
  - Attacker knows the implementation details of the platform and the victim code
- **Objectives** of the lecture are to understand:
  - How software could be attacked in this setting
  - What the vulnerabilities are that enable these attacks
  - What defenses can help remove these vulnerabilities or mitigate these attacks

Overview

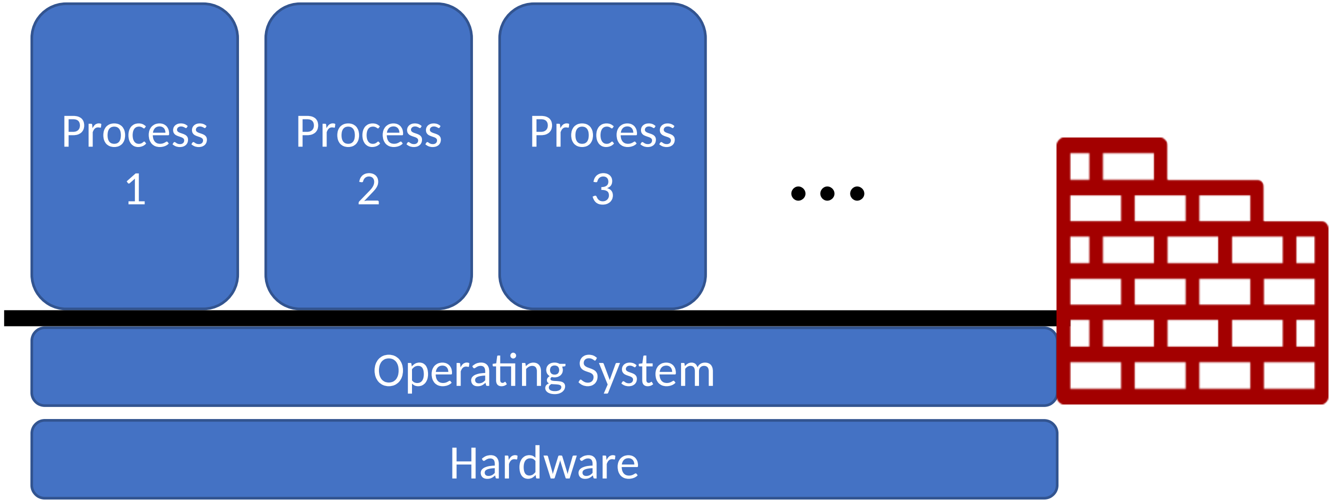
1. **System model**
   - Architectural isolation mechanisms for shared platforms
   - Architecture vs. Microarchitecture
2. Microarchitectural side-channel attacks
3. Transient-execution attacks
4. Conclusions

Classic Hierarchical OS Protection



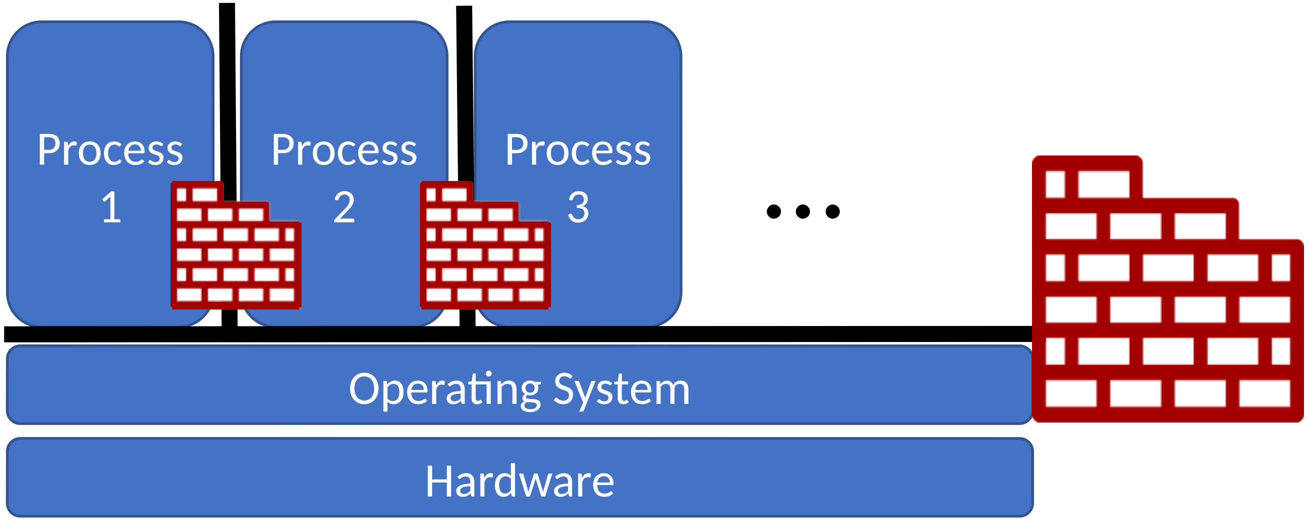
Protecting the Kernel: CPU Privilege Levels

OS is protected from applications by privilege level



Protecting Processes: Virtual Memory

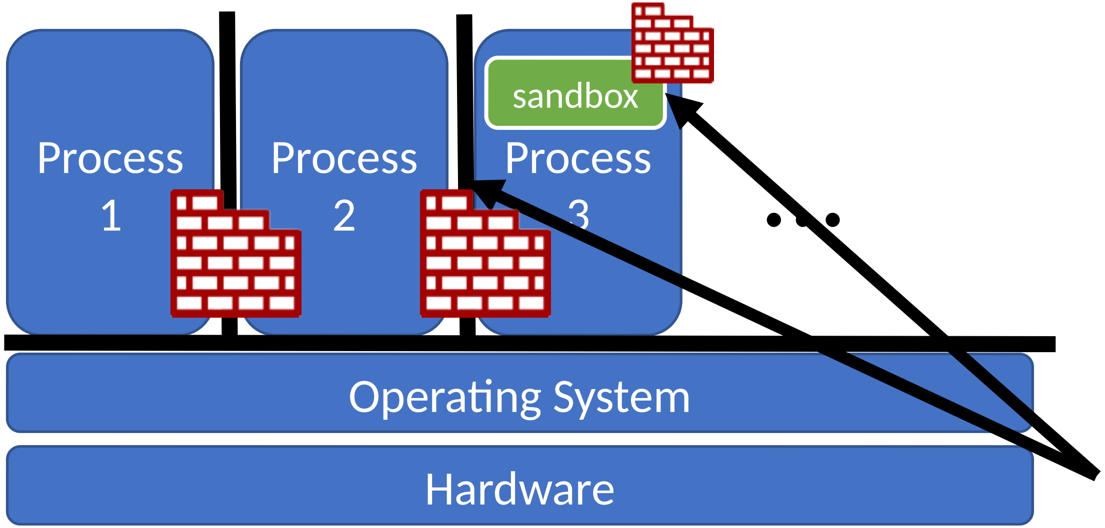
Processes are protected from each other through virtual memory isolation (page tables)



OS is protected from applications by privilege level

Fine-grained Protection: Software-Defined Sandboxes

Processes are protected from each other through virtual memory isolation (page tables)



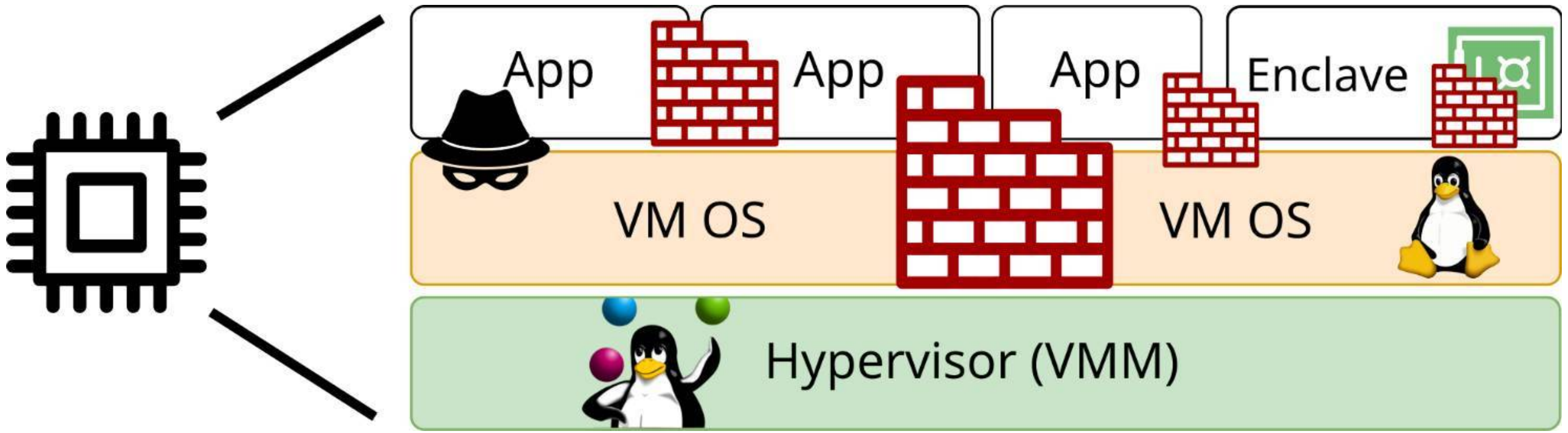
OS is protected from applications by privilege level

Software can additionally enforce more fine-grained protection:

- Safe languages
- JVM
- WebAssembly
- …

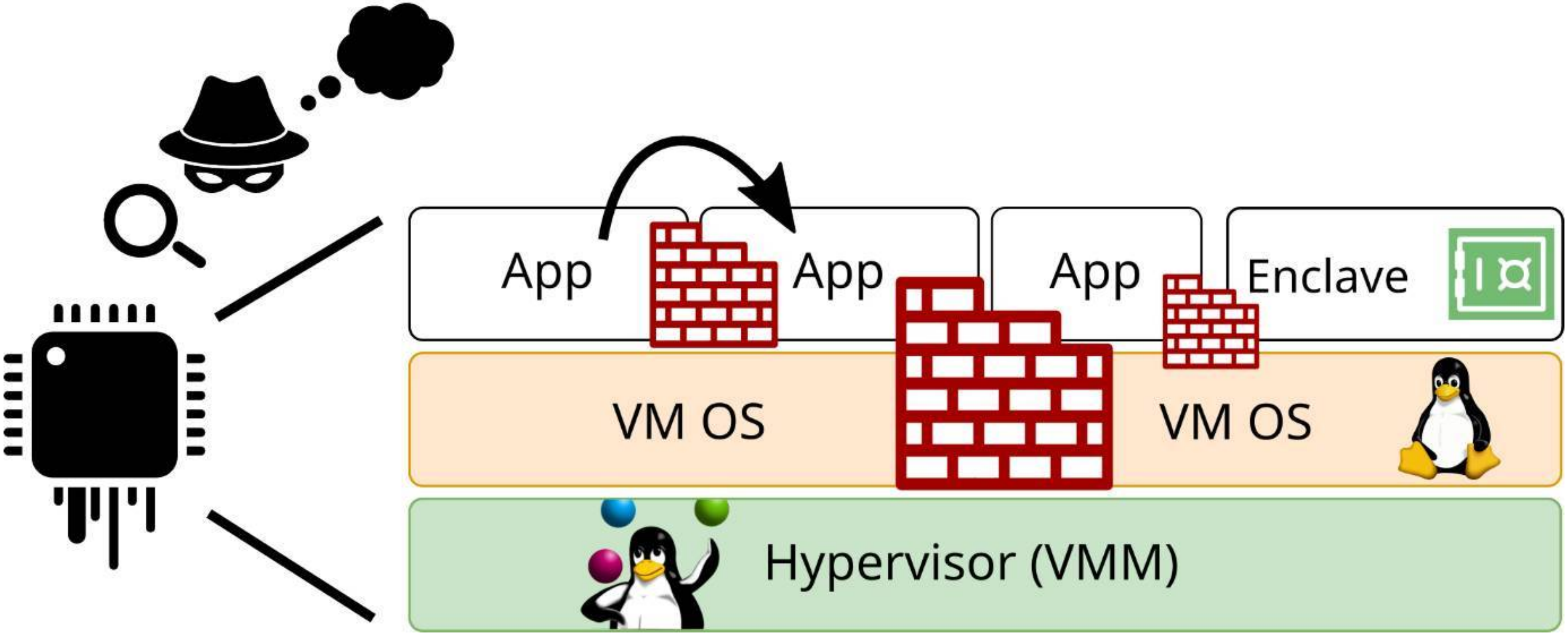
and can implement security checks such as bounds checks

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

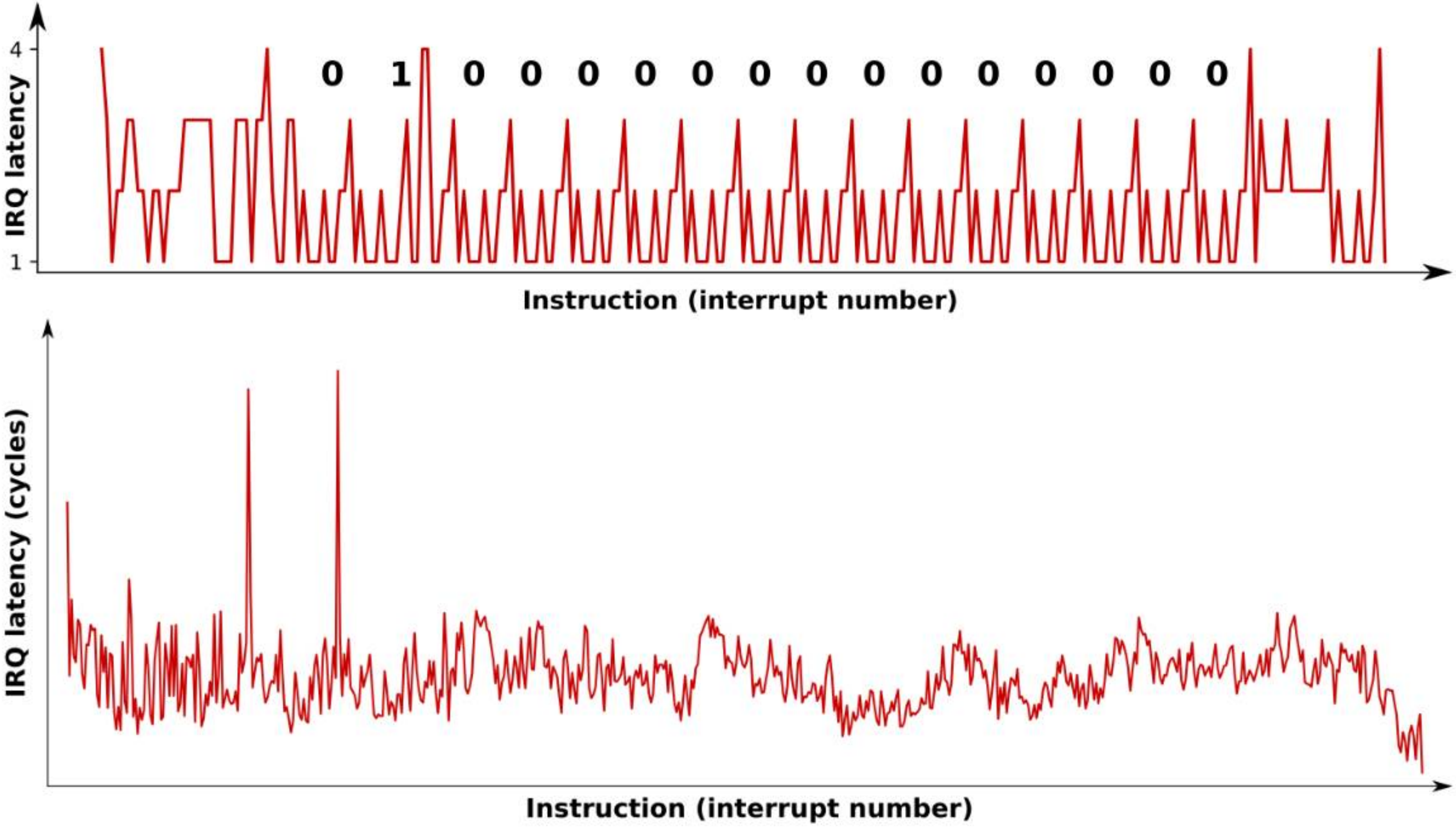


- Different software **protection domains:** Processes, virtual machines, (enclaves)
- CPU builds “walls” for **memory isolation** between apps and privilege levels
- ↔ *But architectural protection walls permeate microarchitectural side channels!*


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


0
35
30

Microarchitectural Timing Leaks in Practice



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

Architecture versus Microarchitecture

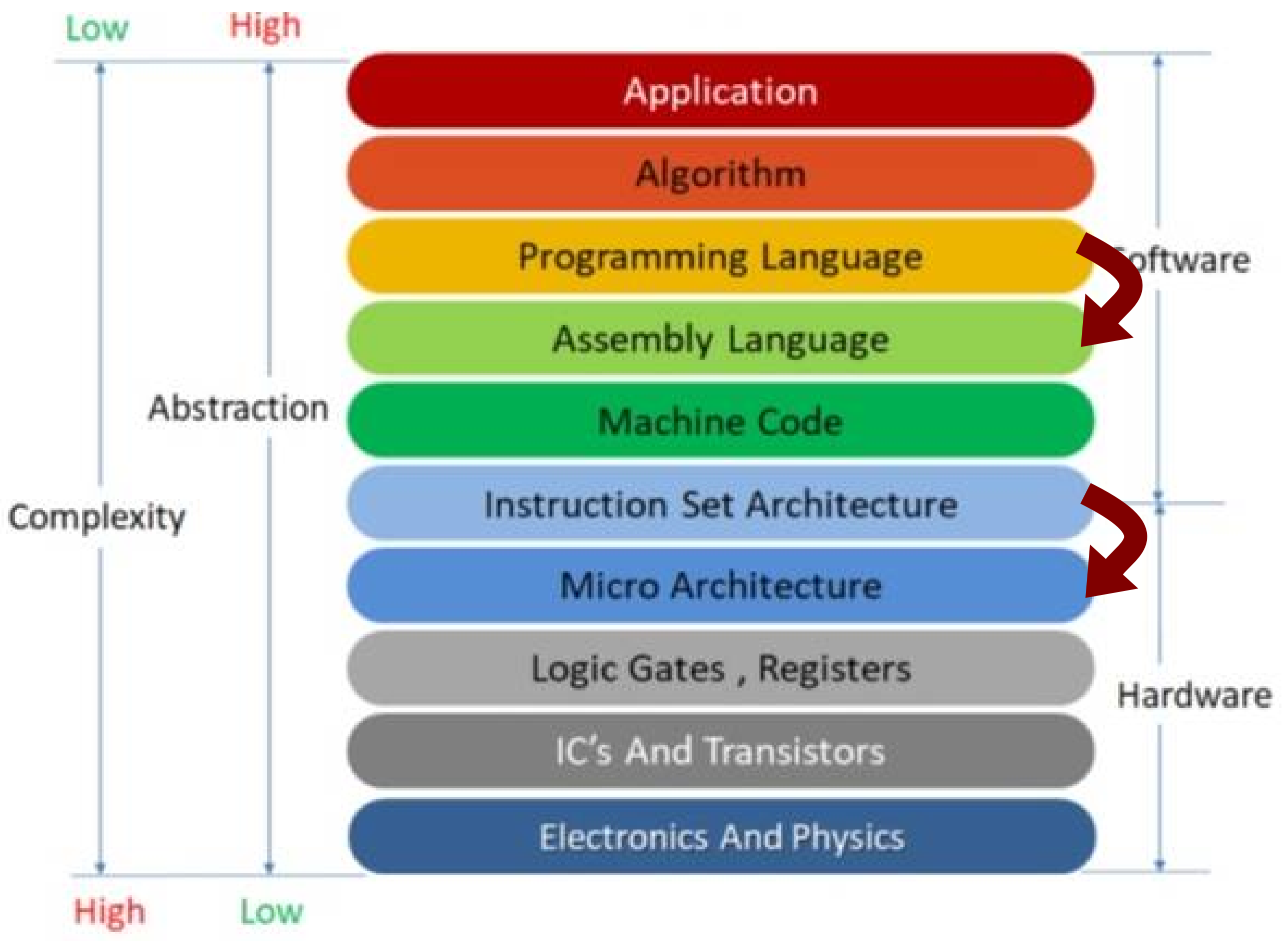
- The **Instruction Set Architecture (ISA)** defines behavior of the machine code:
  - Examples: x86, RISC-V, ARM, …
  - The ISA defines:
    - Architectural state: memory, registers, …
    - Instruction semantics

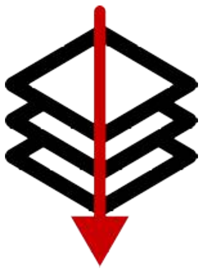
- The **microarchitecture** is the way the ISA is implemented in a particular processor:
  - Examples: single-cycle versus pipelined, in-order versus out-of-order, …
  - This can introduce additional state and behavior:
    - State: e.g., for performance improvements (caches, branch predictor state,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

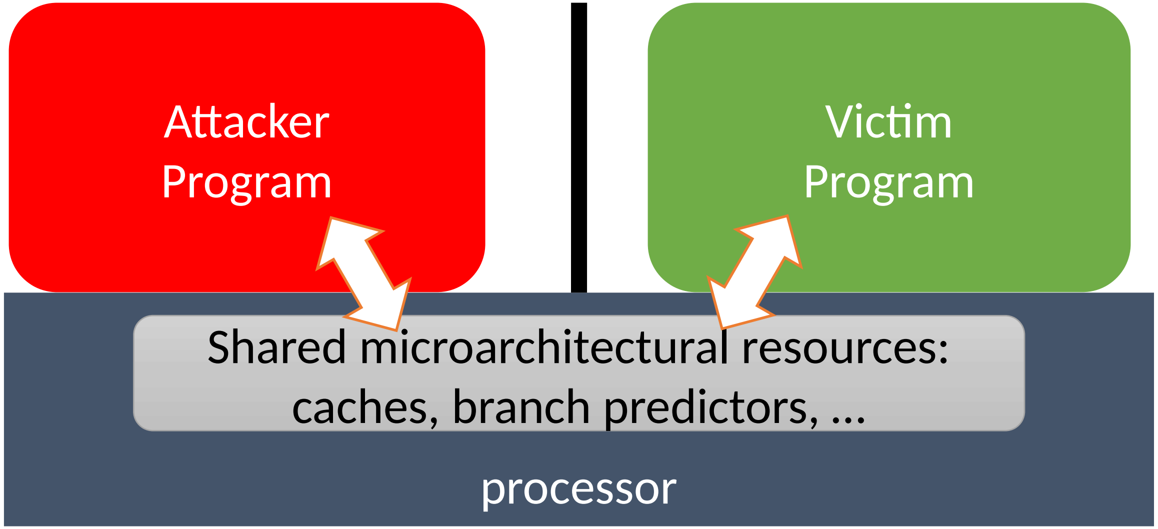
Microarchitectural attacks

Overview

1. System model
2. **Microarchitectural side-channel attacks**
   - Cache timing attacks
   - “Constant-time” software mitigations
3. Transient-execution attacks
4. Conclusions

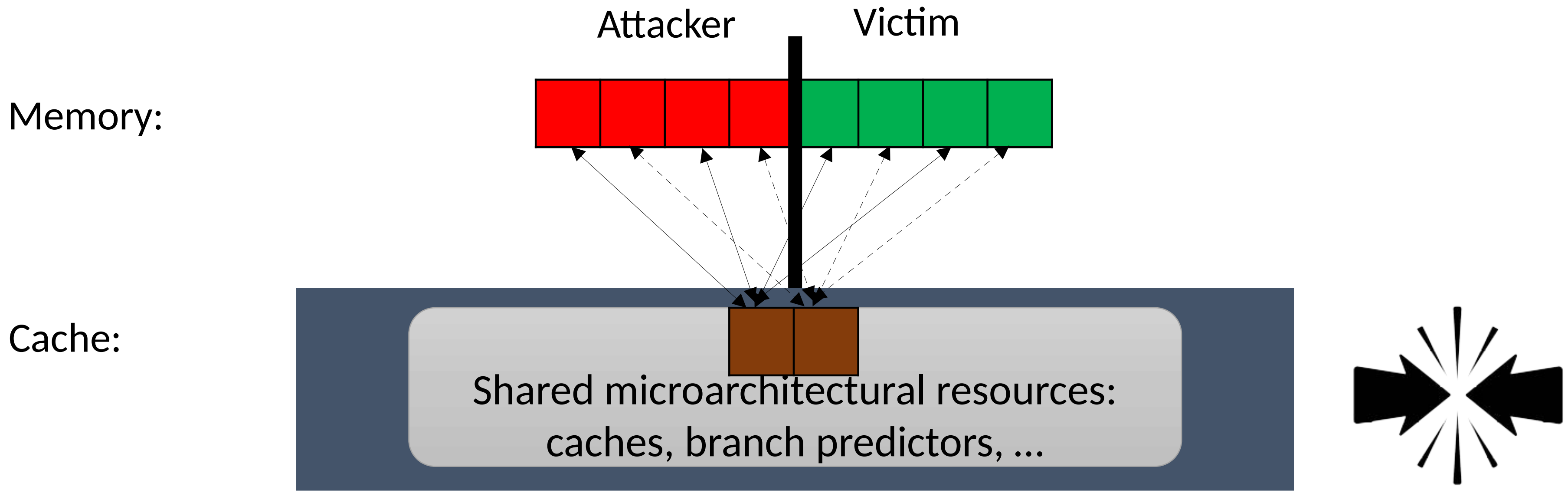
Idea: Microarchitectural Contention

- 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

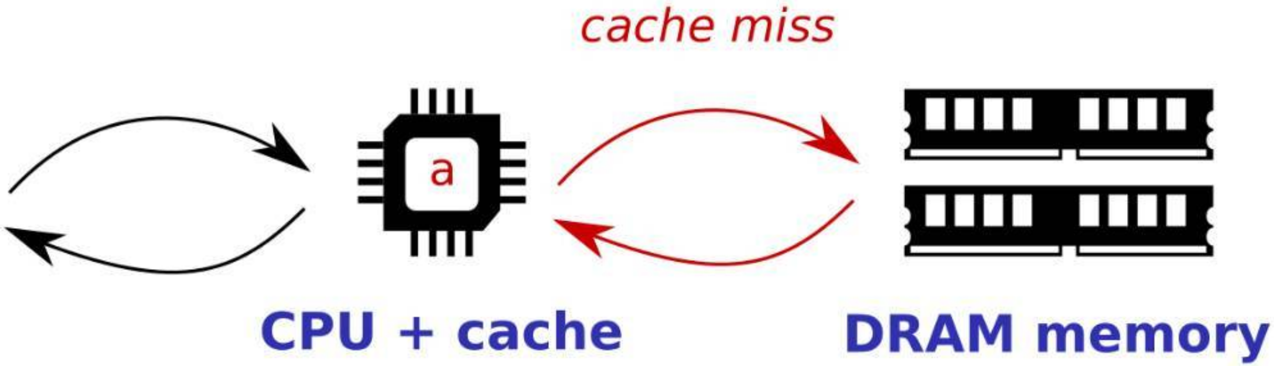
Cache principle: CPU speed $\gg$ DRAM $\rightarrow$ *cache code/data*

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

Example: CPU Cache Timing Side Channel

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

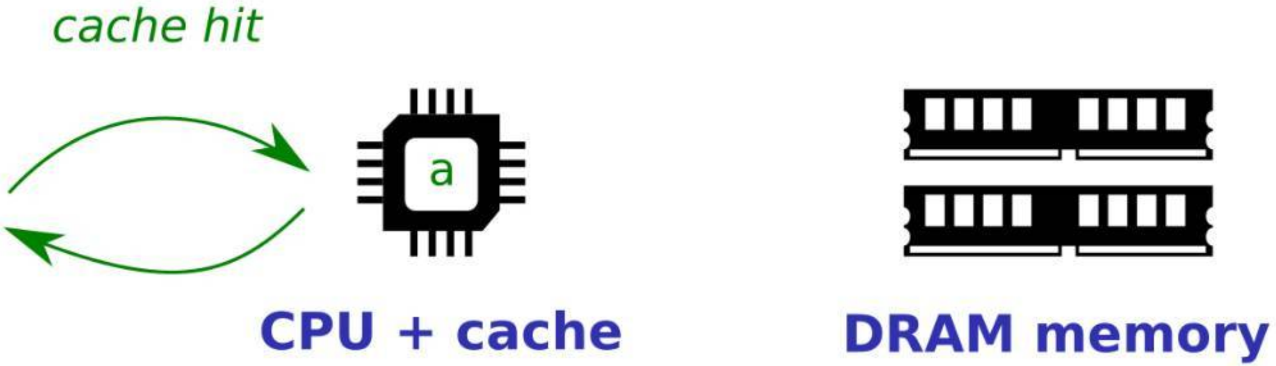
```
while true do
    maccess(&a);
endwh
```



Example: CPU Cache Timing Side Channel

Cache hit: No DRAM access required for subsequent uses

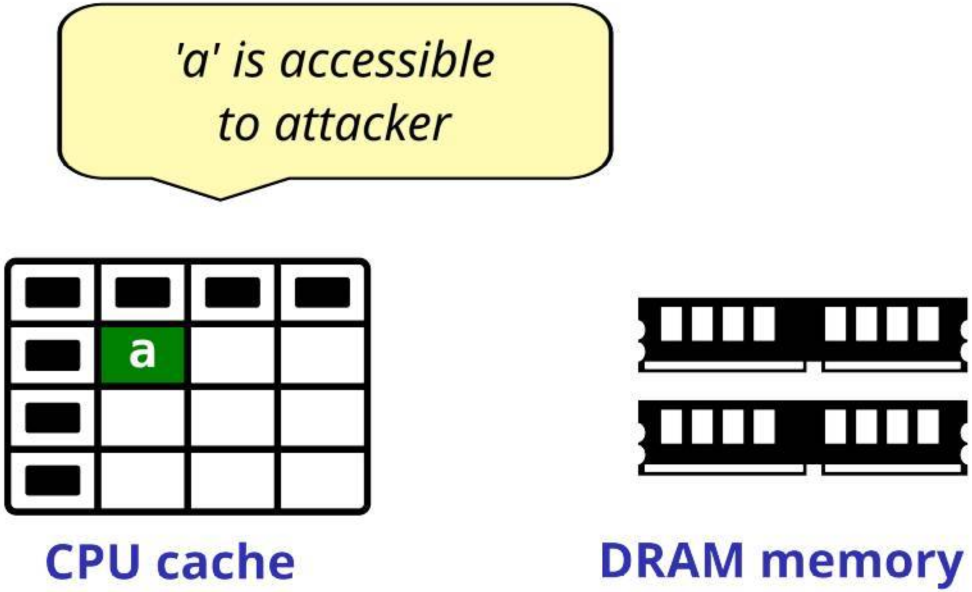
```
while true do
    maccess(&a);
endwh
```



Cache Timing Attacks in Practice: Flush+Reload

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

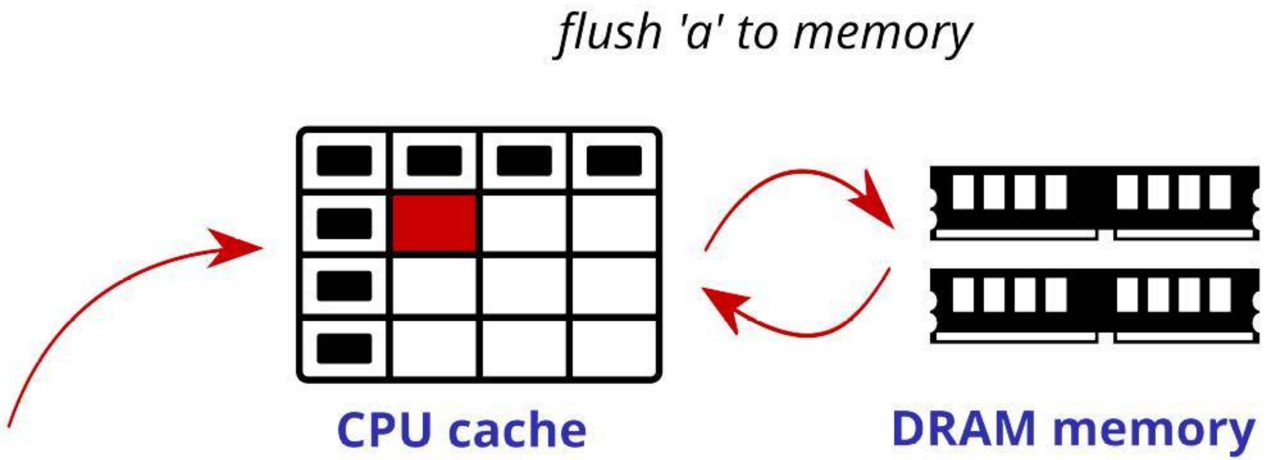
```
flush(&a);
start_timer
   maccess(&a);
end_timer
```



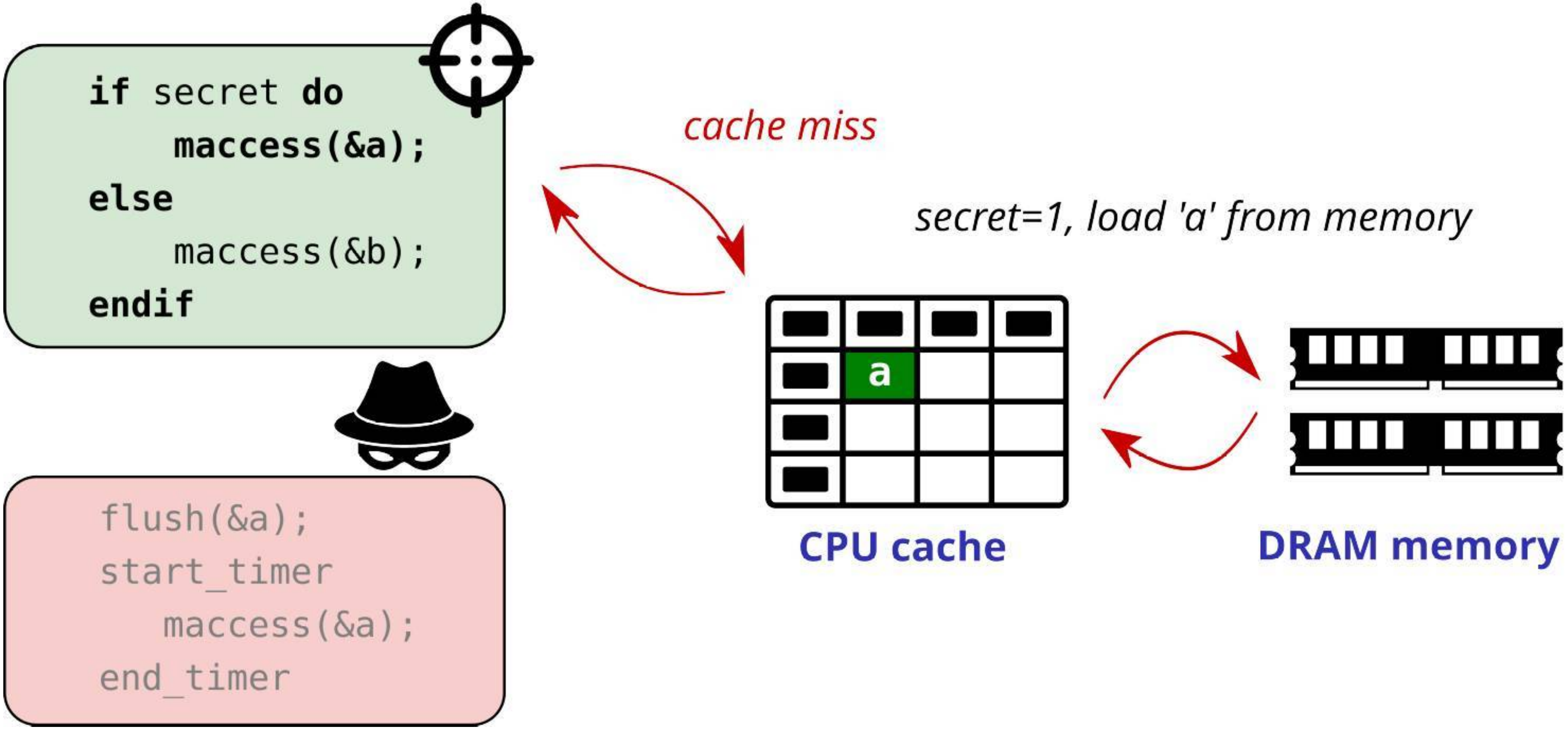
Cache Timing Attacks in Practice: Flush+Reload

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

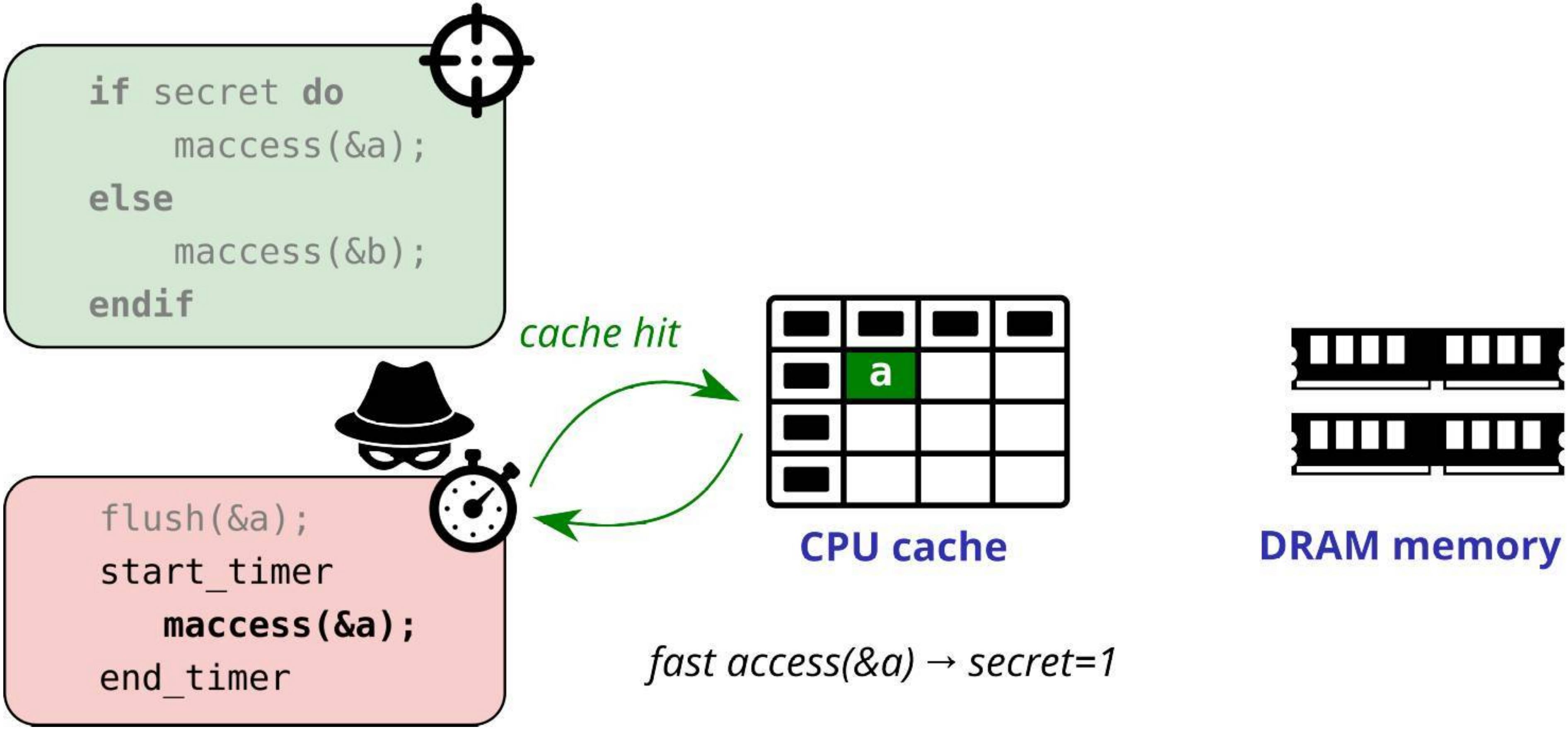
```
flush(&a);
start_timer
   maccess(&a);
end_timer
```



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
```

```
super secret keystroke timings

F
```

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

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

We can communicate across protection walls using microarchitectural side channels!

Vulnerable Patterns: Secret-Dependent Code/Data Accesses

```c
void secret_vote(char candidate)
{
    if (candidate == 'a')
        vote_candidate_a();
    else
        vote_candidate_b();
}
```

```c
int secret_lookup(int s)
{
    if (s > 0 && s < ARRAY_LEN)
        return array[s];
    return -1;

}
```

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

Leak only metadata

Software Mitigation: “Constant-Time” Programming

```c
void check_pwd(char *input)
{
    for (int i=0; i < PWD_LEN; i++)
        if (input[i] != pwd[i])
            return 0;

    return 1;
}
```

```c
void check_pwd(char *input)
{
    int rv = 0x0;
    for (int i=0; i < PWD_LEN; i++)
        rv |= input[i] ^ pwd[i];

    return (result == 0);
}
```

Rewrite program such that execution time does <u>not</u> depend on secrets

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

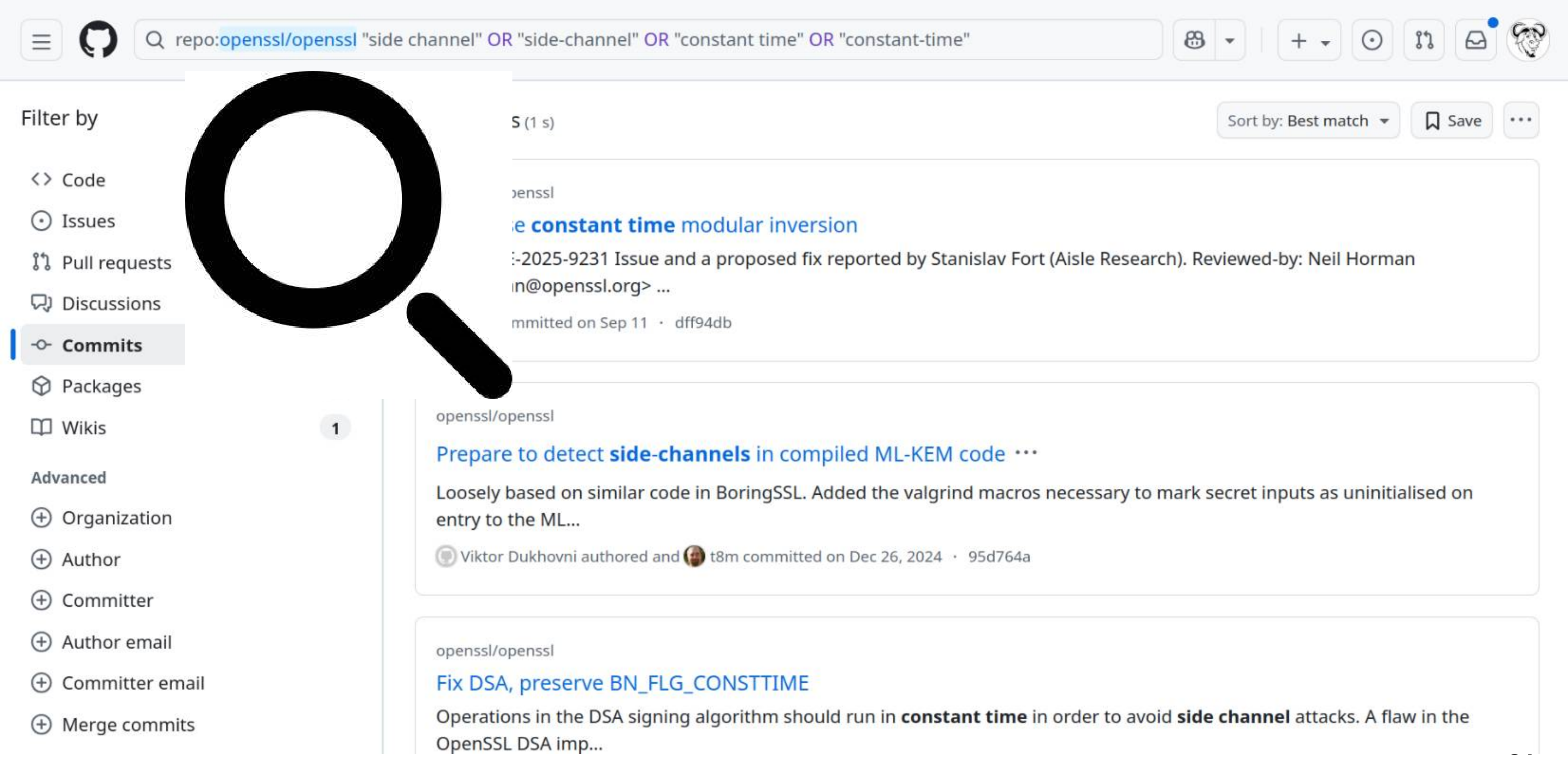
Software Mitigation: "Constant-Time" Programming

- **"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




WHAT IF I TOLD YOU
YOU CAN CHANGE RULES MID-GAME

Overview

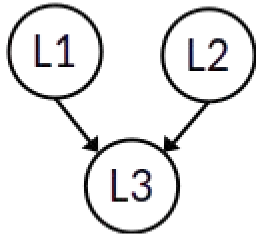
1. System model
2. Microarchitectural side-channel attacks
3. **Transient-execution attacks**
   - Spectre, Meltdown, Foreshadow
   - Hardware-Software Defenses
4. Conclusions

Background: Out-of-Order and Speculative Execution

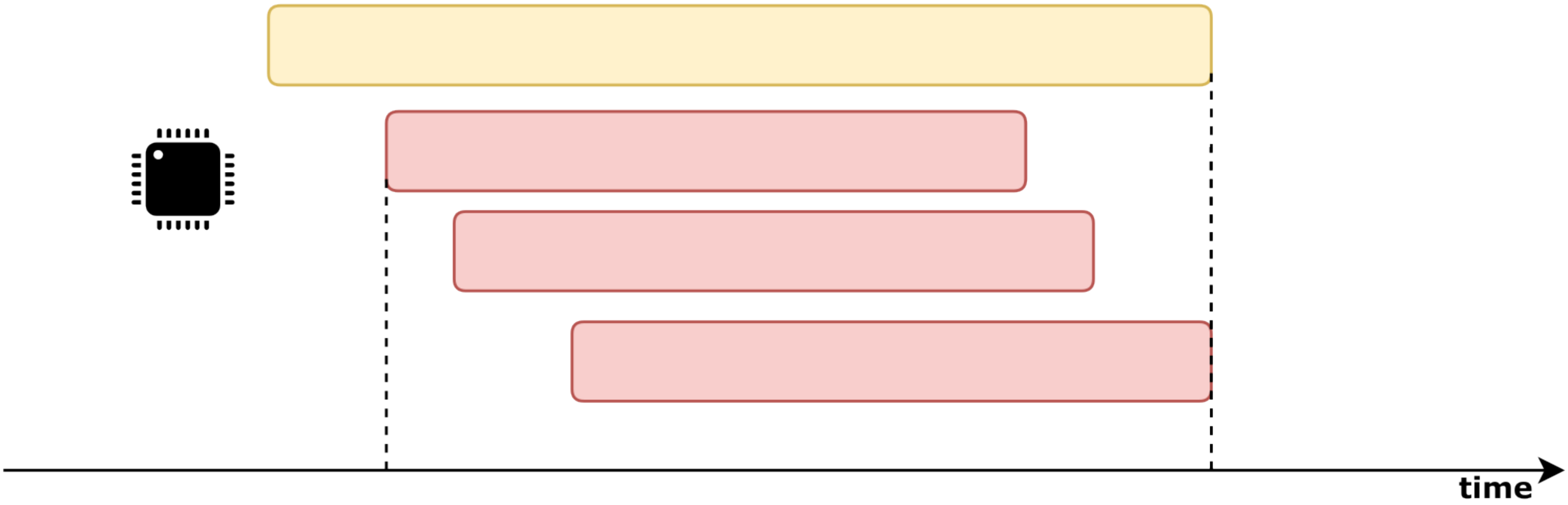
- Modern CPUs have deep out-of-order (OoO) pipelines:
  - Rather than executing one instruction at a time, **fetch** many instructions into a reorder buffer (ROB) of *in-flight instructions*
  - **Execute** instructions from this buffer, possibly *out-of-order*

    → This avoids having to wait while, for instance a slow memory load is happening
  - **Commit** the effect of the instructions to the architectural state *in order*
- Prediction and speculation are used to speed things up
  - For instance, fetching instructions beyond a branch requires prediction

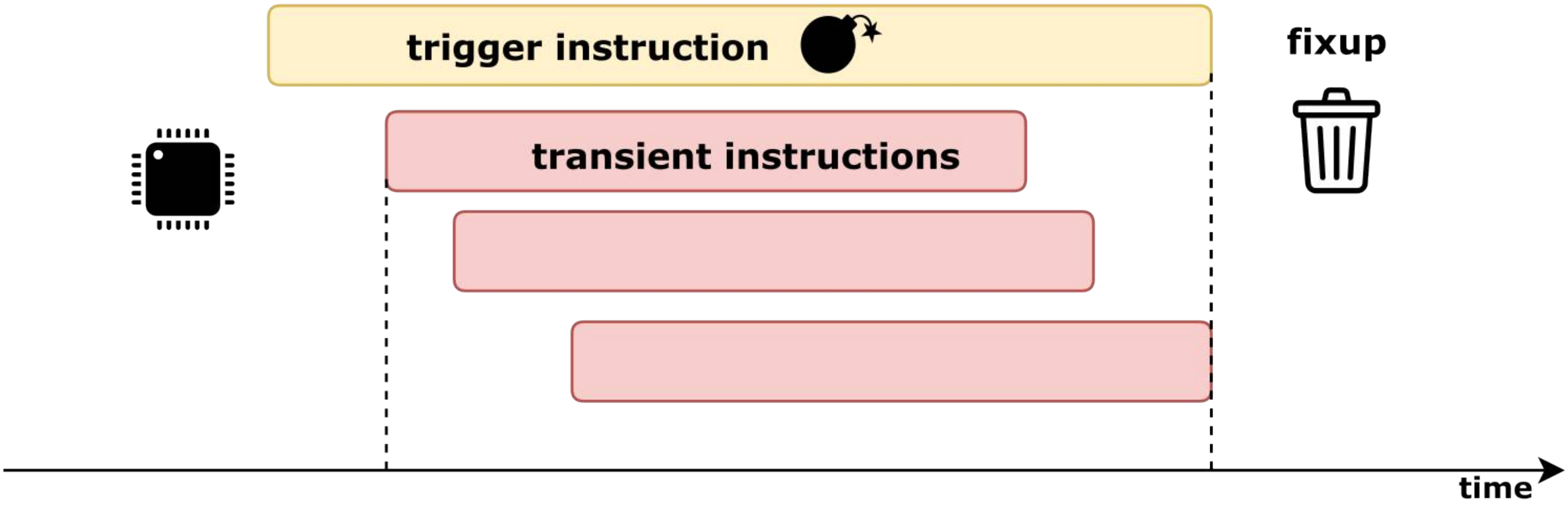
```
int a = *uncached_mem; // L1
int b = c + d;         // L2
if (a) { b++; }        // L3
```



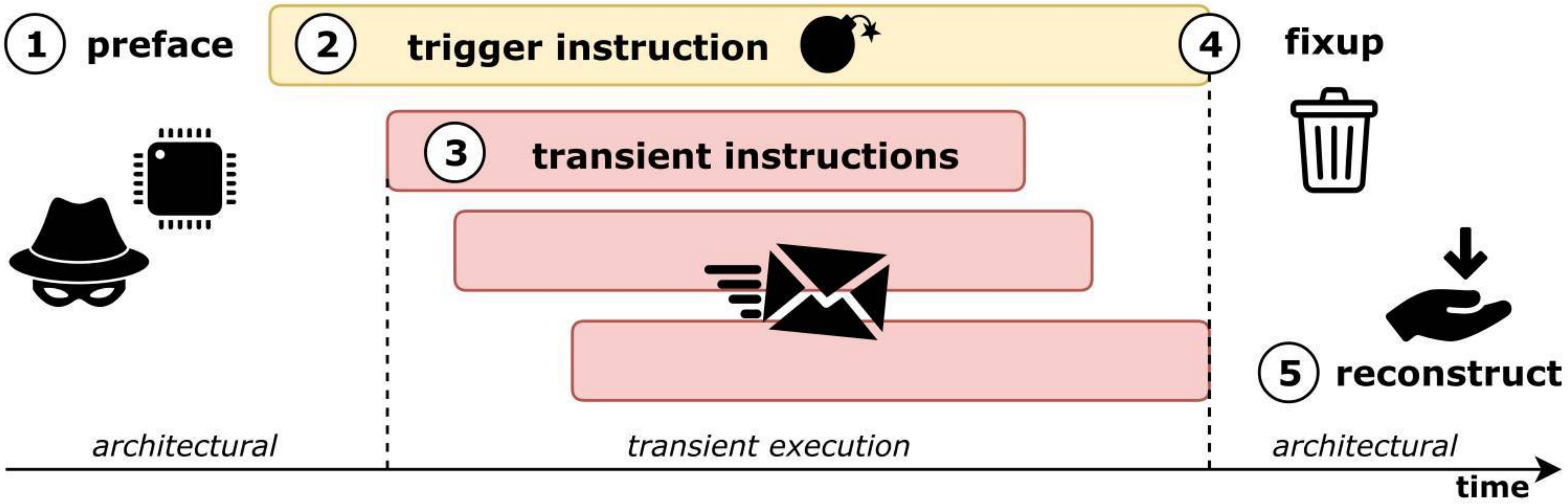
General Transient-Execution Attack Structure



General Transient-Execution Attack Structure



General Transient-Execution Attack Structure



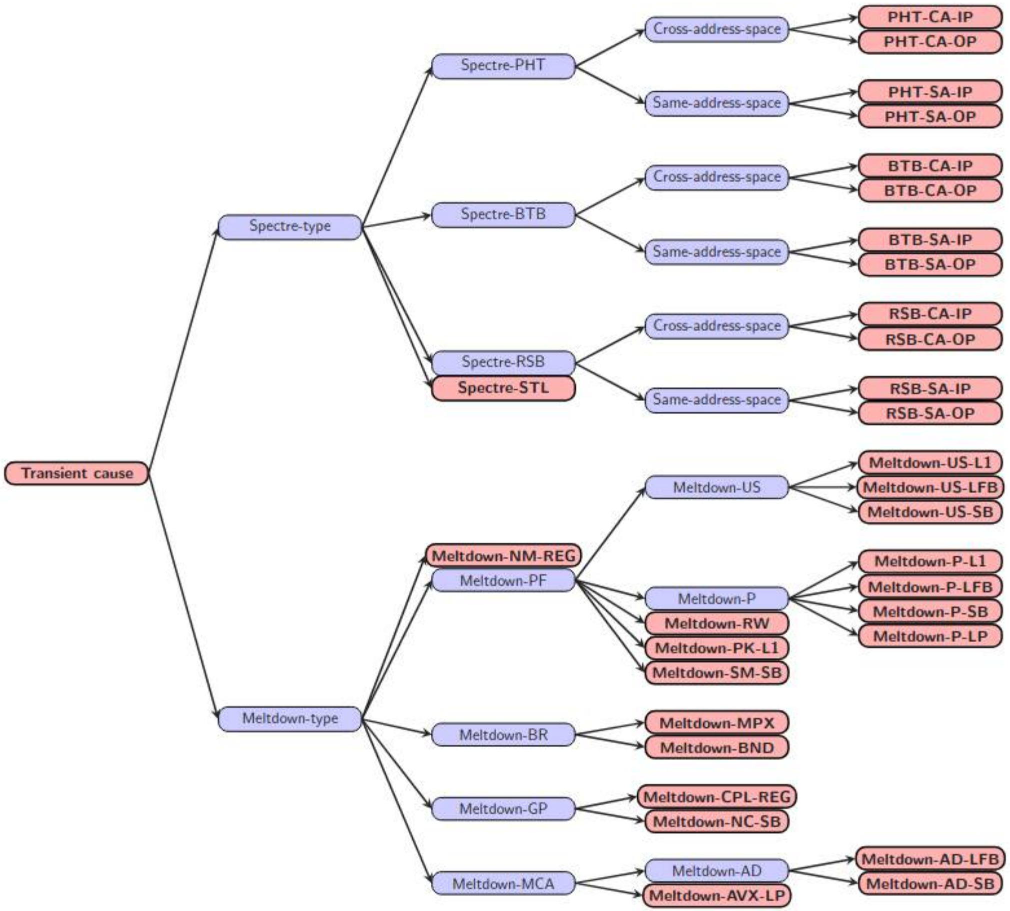
Idea: Transiently executed instructions can also leak information to the attacker

→ On rollback, architectural effects are discarded, but microarchitectural effects remain…

Transient Execution: Welcome to the Word of Fun!

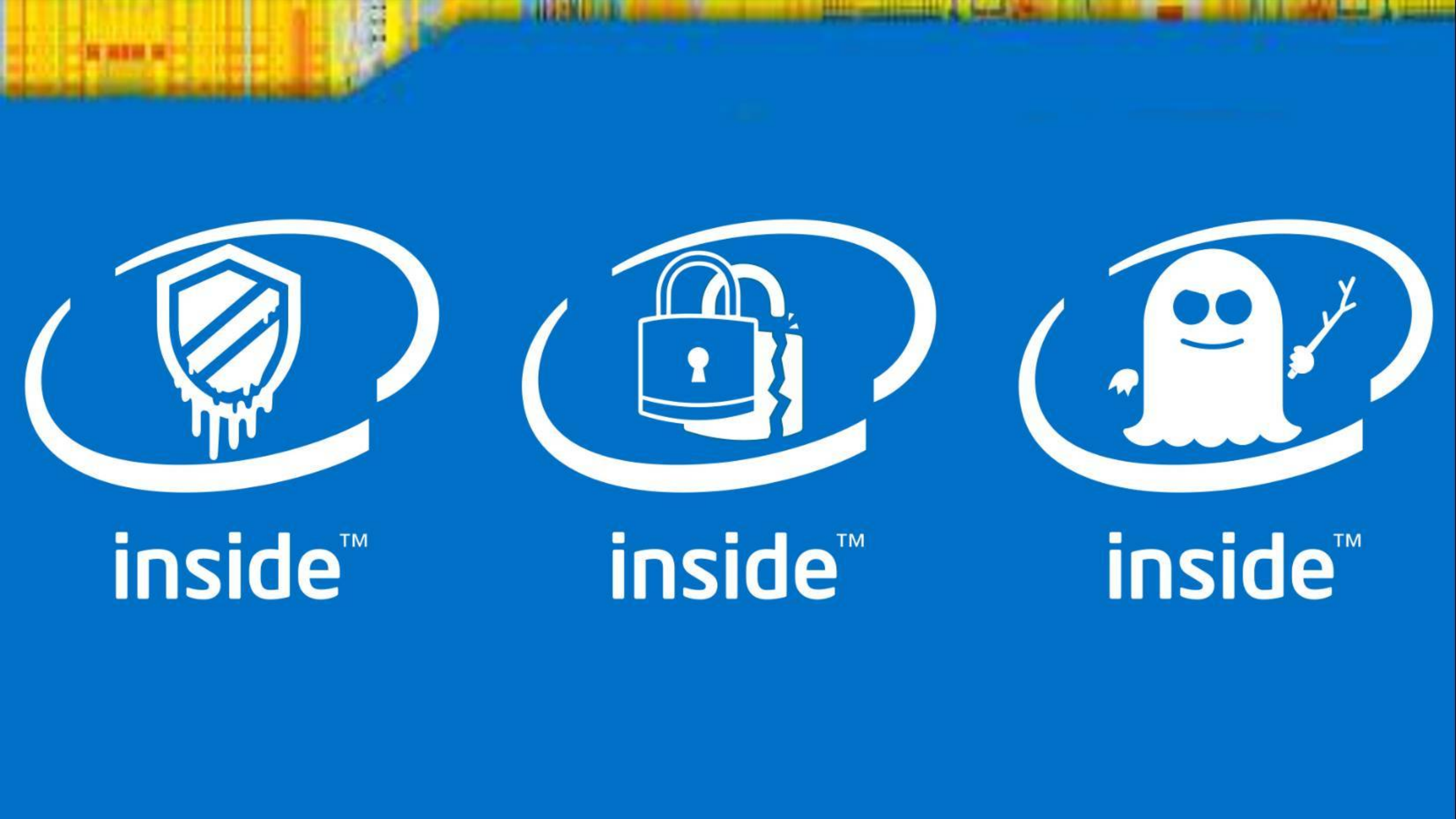
The Transient-Execution Zoo

https://transient.fail/

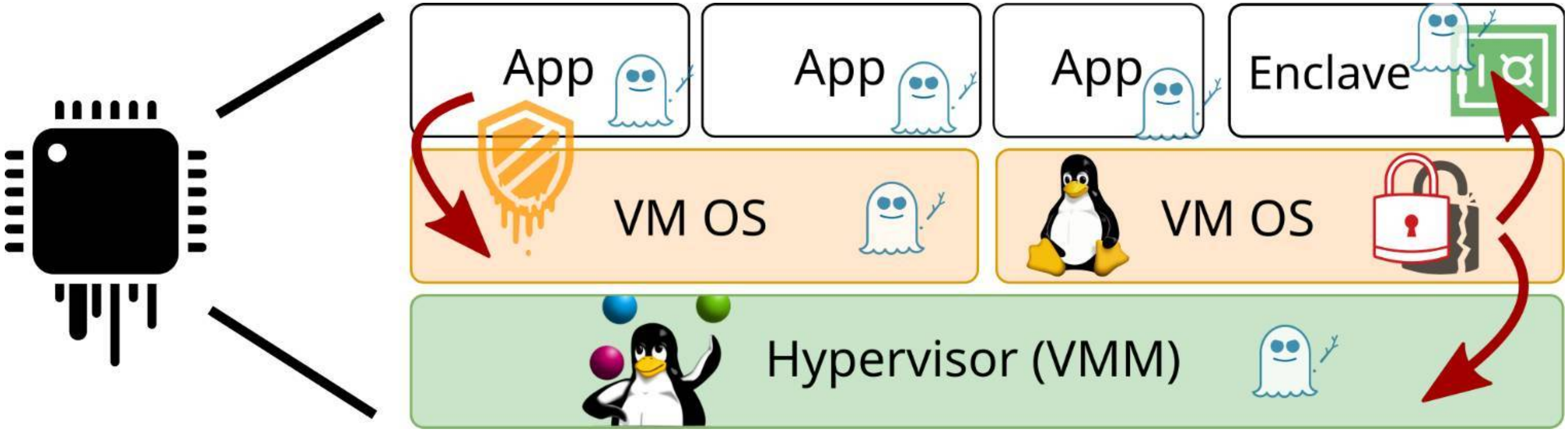


Idea: Transient instructions can access information expected to be inaccessible:

- Because the information is *protected by software*
  → "Spectre"-style attacks
- Because it is in another *hardware protection domain*
  → "Meltdown"-style attacks

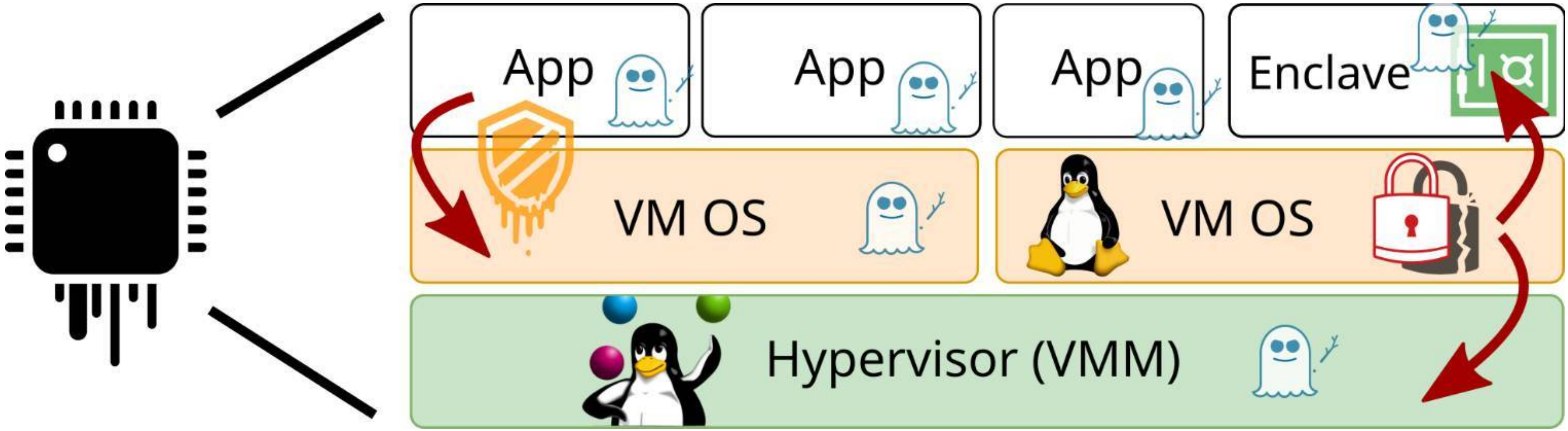

inside™
inside™
inside™

Breaking Architectural Isolation with Transient Execution



- Meltdown breaks *user/kernel* isolation
- Foreshadow breaks SGX *enclave and virtual machine* isolation
- Spectre breaks *software-defined* isolation on various levels
- … many more – but all exploit the same underlying insights!

Breaking Architectural Isolation with Transient Execution



- Meltdown breaks *user/kernel* isolation
- Foreshadow breaks SGX *enclave and virtual machine* isolation
- Spectre breaks *software-defined* isolation on various levels
- ... many more – but all exploit the same underlying insights!

} *HW fixes*

→ HW-SW *fixes*

Meltdown: Transiently Encoding Unauthorized Memory

Unauthorized access → Transient out-of-order window → **Exception handler**

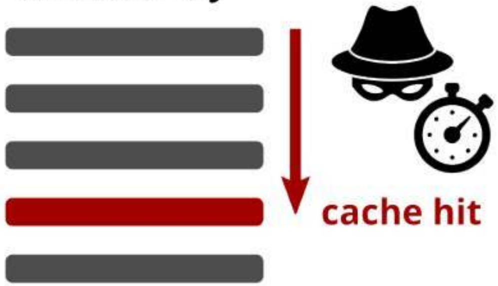
Listing 1: x86 assembly.

```
meltdown:
  // %rdi: oracle
  // %rsi: secret_ptr

  movb (%rsi), %al
  shl $0xc, %rax
  movq (%rdi, %rax), %rdi
  retq
```

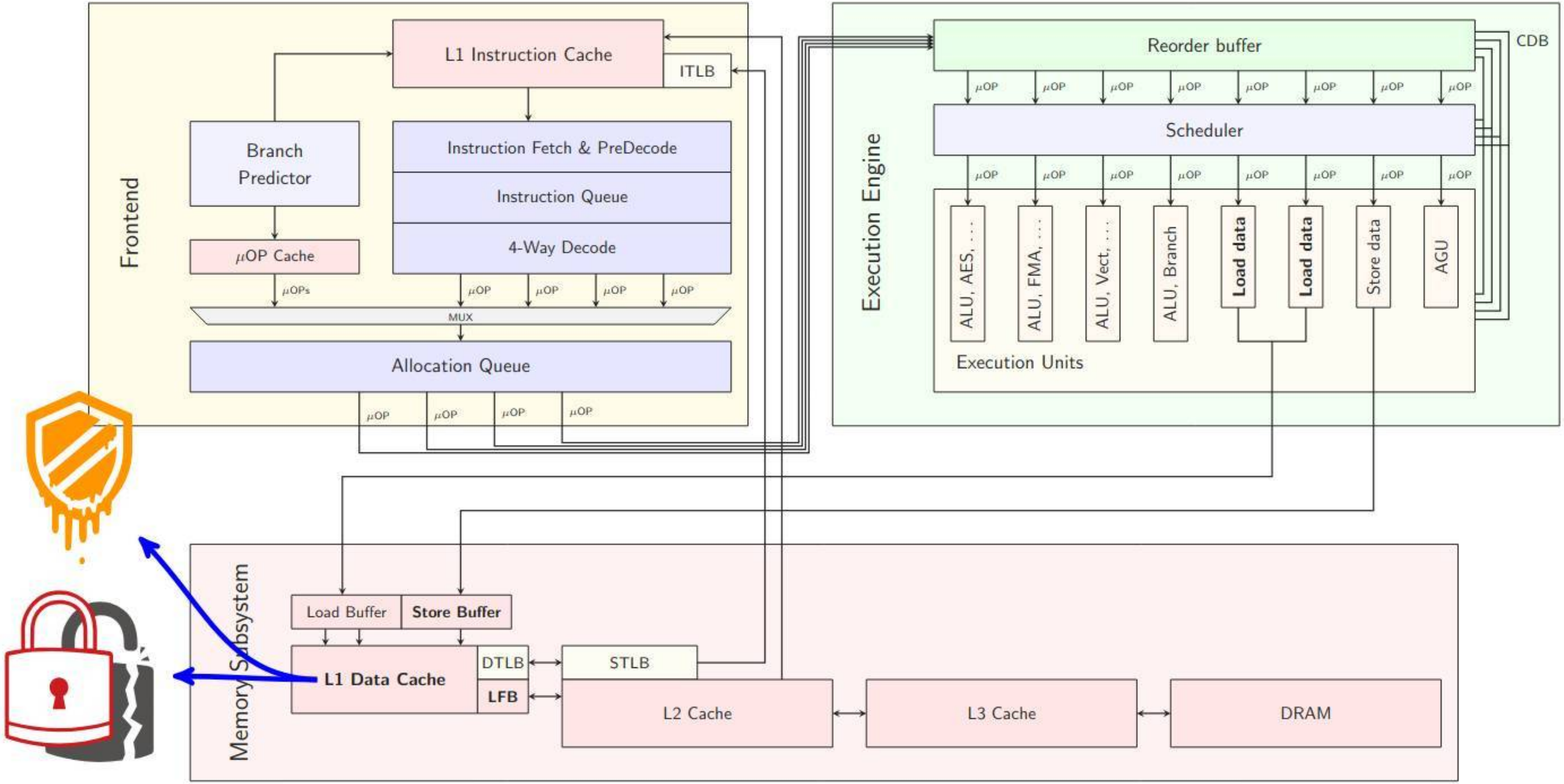
Listing 2: C code.

```
void meltdown(
        uint8_t *oracle,
        uint8_t *secret_ptr)
{
  uint8_t v = *secret_ptr;
  v = v * 0x1000;
  uint64_t o = oracle[v];
}
```

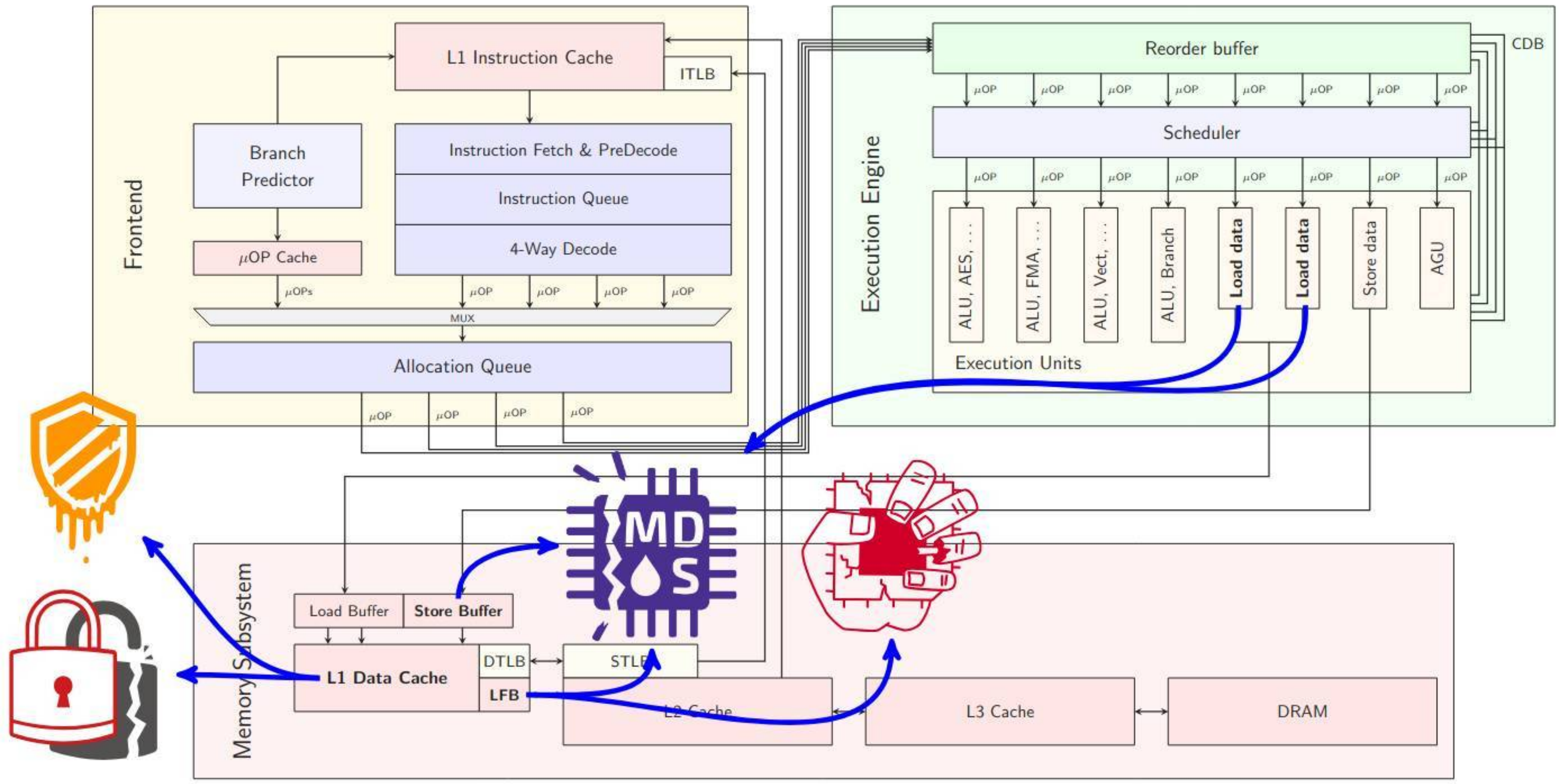



inside™
inside™
inside™

Meltdown Variants: Escaping Virtual Memory (~2018)



Meltdown Variants: Microarchitectural Buffers (~2019)



```
jo@gropius:/sys/devices/system/cpu/vulnerabilities$ find . -type f -exec sh -c 'for file; do if grep -q
"Not affected" "$file"; then echo -n "\033[32m\033[1m$(basename "$file"): \033[0m"; else echo -n "\033[3
1m\033[1m$(basename "$file"): \033[0m"; fi; cat "$file"; done' sh {} +
spectre_v2: Mitigation: Enhanced / Automatic IBRS; IBPB: conditional; RSB filling; PBRSB-eIBRS: SW seque
nce; BHI: BHI_DIS_S
itlb_multihit: Not affected
vmscape: Mitigation: IBPB before exit to userspace
mmio_stale_data: Not affected
mds: Not affected
reg_file_data_sampling: Mitigation: Clear Register File
l1tf: Not affected
spec_store_bypass: Mitigation: Speculative Store Bypass disabled via prctl
tsx_async_abort: Not affected
spectre_v1: Mitigation: usercopy/swapgs barriers and __user pointer sanitization
gather_data_sampling: Not affected
retbleed: Not affected
spec_rstack_overflow: Not affected
srbds: Not affected
meltdown: Not affected
```

- "Meltdown-type" attacks (mostly) mitigated in modern **hardware**...
- "Spectre-type" attacks (v1/PHT and v4/STL) need patches in **software**...


inside™
inside™
inside™

Spectre v1: Speculative Buffer Over-Read


```
if (idx < LEN)
{
  s = buffer[idx];
  t = lookup[s];
  ...
}
```

- Programmer *intention:* no out-of-bounds accesses

Spectre v1: Speculative Buffer Over-Read


```
if (idx < LEN)
{
  s = buffer[idx];
  t = lookup[s];
  ...
}
```

- Programmer *intention:* no out-of-bounds accesses
- **Mistrain gadget** to speculatively "ahead of time" execute with `idx ≥ LEN` in the transient world

Spectre v1: Speculative Buffer Over-Read


```
if (idx < LEN)
{
   s = buffer[idx];
   t = lookup[s];
   ...
}
```

- Programmer *intention:* no out-of-bounds accesses
- **Mistrain gadget** to speculatively "ahead of time" execute with `idx ≥ LEN` in the transient world
- **Side channels** may leave traces after roll-back!

Spectre v1: Speculative Buffer Over-Read


```
if (idx < LEN)
{
  asm("lfence\n\t");
  s = buffer[idx];
  t = lookup[s];
  ...
}
```

- Programmer *intention:* no out-of-bounds accesses
- **Mistrain gadget** to speculatively "ahead of time" execute with `idx ≥ LEN` in the transient world
- **Side channels** may leave traces after roll-back!
- Insert explicit **speculation barriers** to tell the CPU to halt the transient world…

Spectre v1: Speculative Buffer Over-Read


```
if (idx < LEN)
{
  asm("lfence\n\t");
  s = buffer[idx];
  t = lookup[s];
  ...
}
```

- Programmer *intention:* no out-of-bounds accesses
- **Mistrain gadget** to speculatively "ahead of time" execute with `idx ≥ LEN` in the transient world
- **Side channels** may leave traces after roll-back!
- Insert explicit **speculation barriers** to tell the CPU to halt the transient world…

Manual, error-prone effort(!)

Overview

1. System model

2. Microarchitectural side-channel attacks

3. Transient-execution attacks

4. **Conclusions**

Conclusions and Take-Away

- **Microarchitectural attacks** break architectural isolation “walls”

  → *New dangerous class of transient-execution attacks*

- **Short-term defenses** include patches across the system stack:

  → *Hardware / operating system / compiler*

- **Long-term defenses** are the subject of current research

  → Fundamentally new *hardware-software co-design* may be required...

Thank you! Questions?