ARM MTE: The End of Memory Corruption? Not Yet.

Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung,
Youngjoo Lee, Taesoo Kim, Byoungyoung Lee

Samsung Research

Whoami

Juhee Kim

Ph.D Student at CompSec Lab,

Seoul National University

kimjuhi96@snu.ac.kr

Focuses on

- Software and Systems security
- Bug finding, Attack mitigation
- Linux kernel, Web browser, GPU/ML systems

Contributors

Samsung Research

Jinbum Park
- Security researcher at Samsung Research
- System security, Confidential Computing
- Published in USENIX Security and ASPLOS

Sihyeon Roh
- Ph.D Student at CompSec Lab
- Hardware side-channels

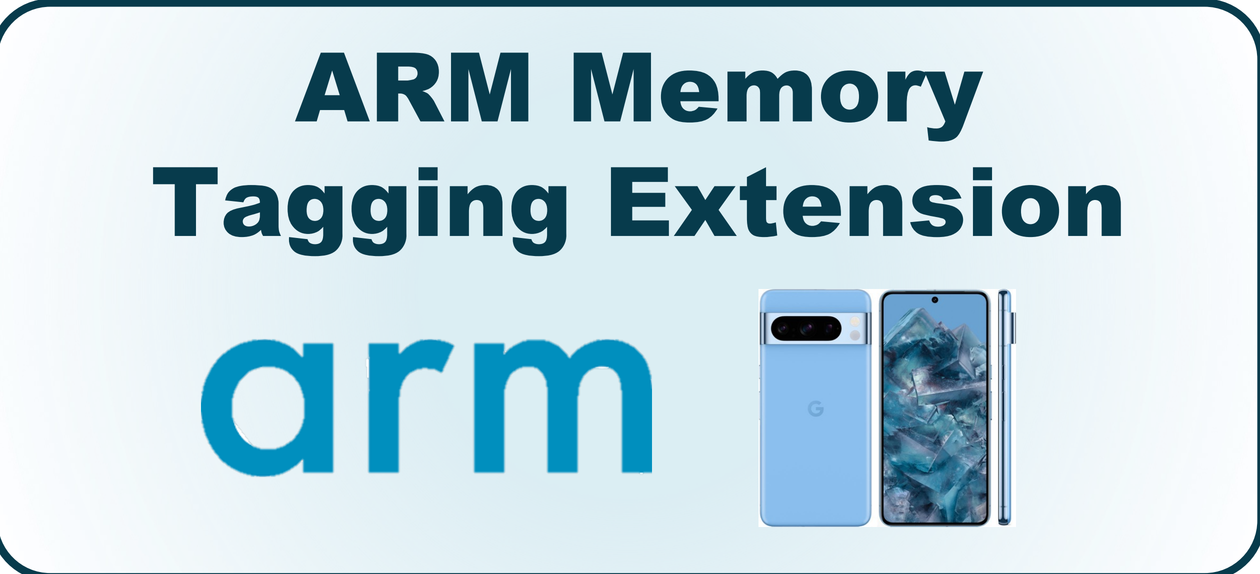
Jaeyoung Chung
- Ph.D Student at CompSec Lab
- System Security
- CTF player

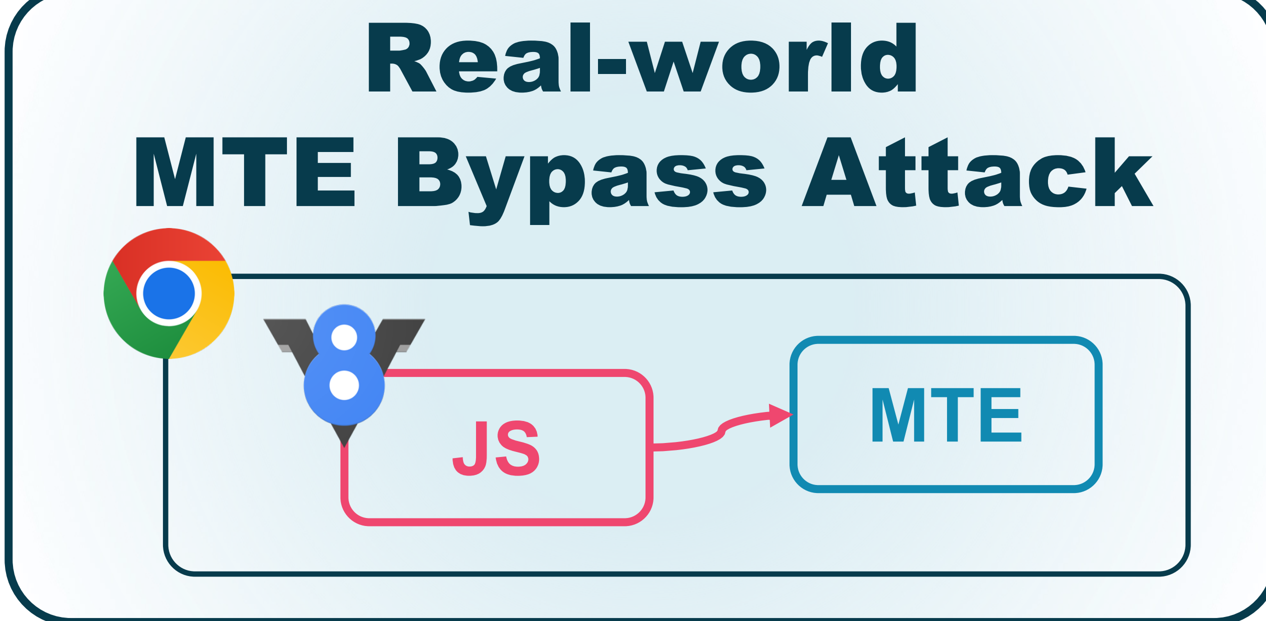
Youngjoo Lee
- Ph.D Student at CompSec Lab
- Fuzzing, Browser security, Bug gounty
- CTF player

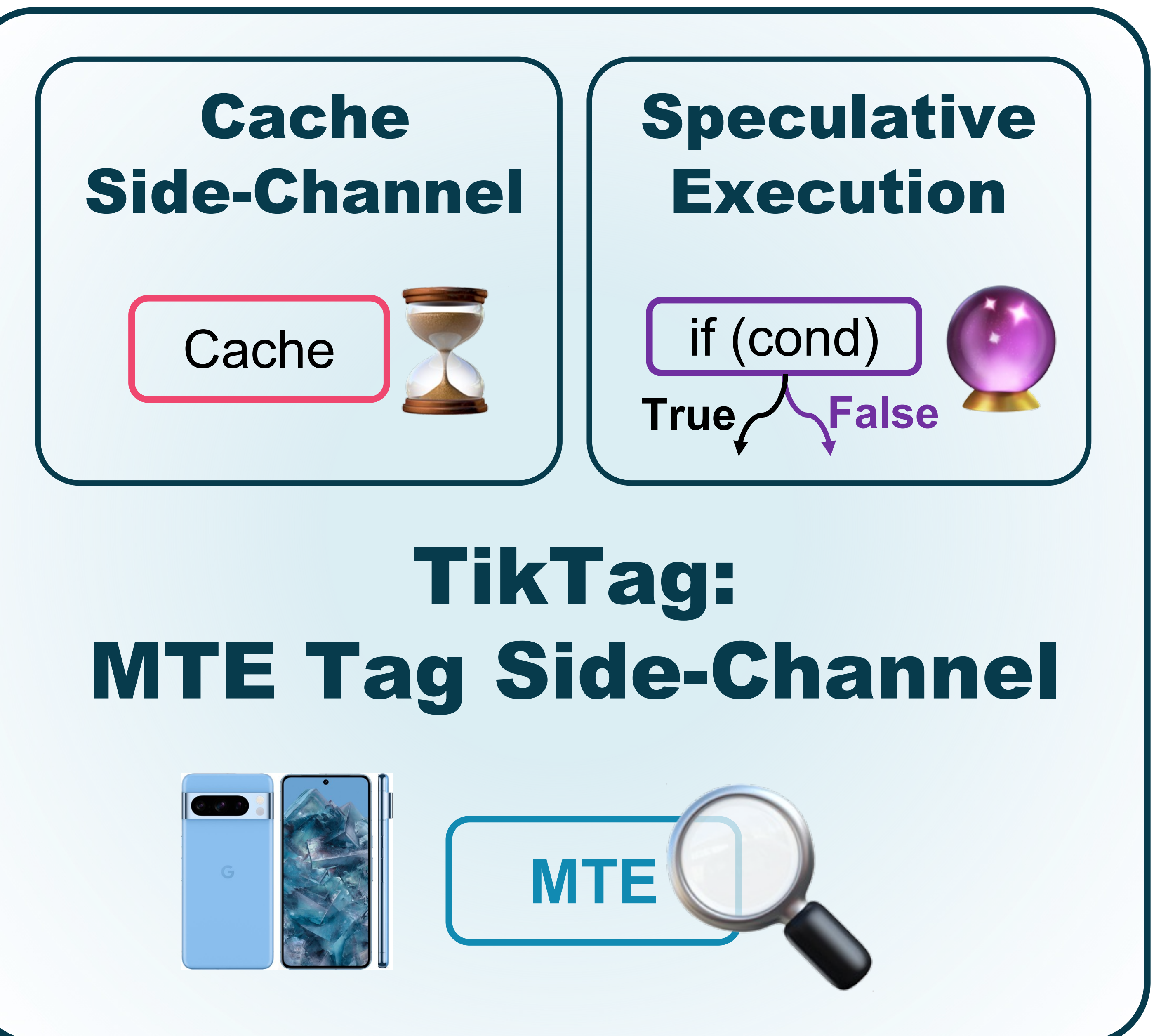
Taesoo Kim
- Vice president of Samsung Research
  Professor of Georgia Tech
- Won several best paper awards from USENIX Security, EuroSys

Byoungyoung Lee
- Professor of Seoul National University
- Leads CompSec Lab
- System security, Confidential computing
- Previous CTF player
- Spoken at Black Hat

Roadmap







Memory corruption attacks

have been the most pervasive and dangerous security threats

Heartbleed (2014)

OpenSSL information leak

Bad Binder (2019)

Bad Binder: Android In-The-Wild Exploit

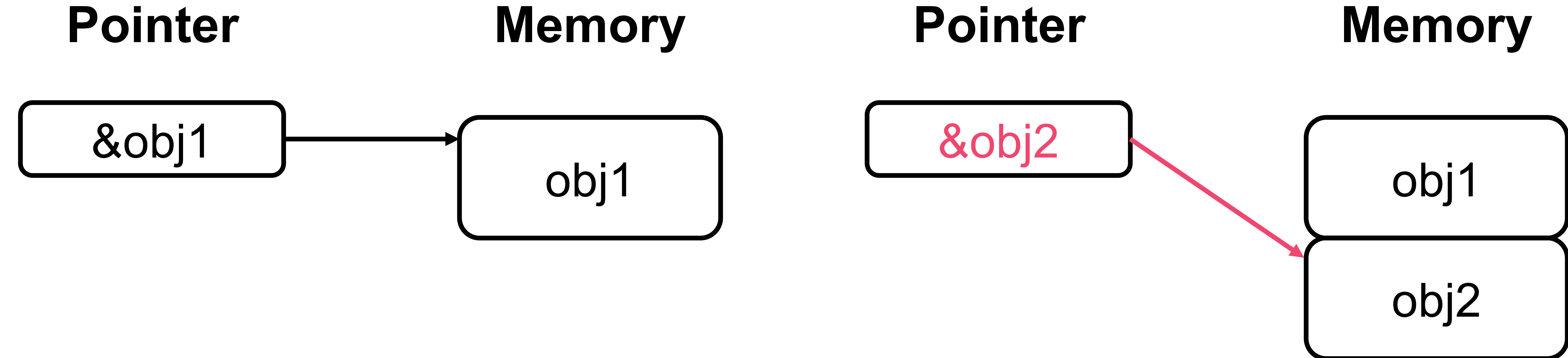
reggreSSHion (2024)

regreSSHion: Remote Unauthenticated Code Execution Vulnerability in OpenSSH server

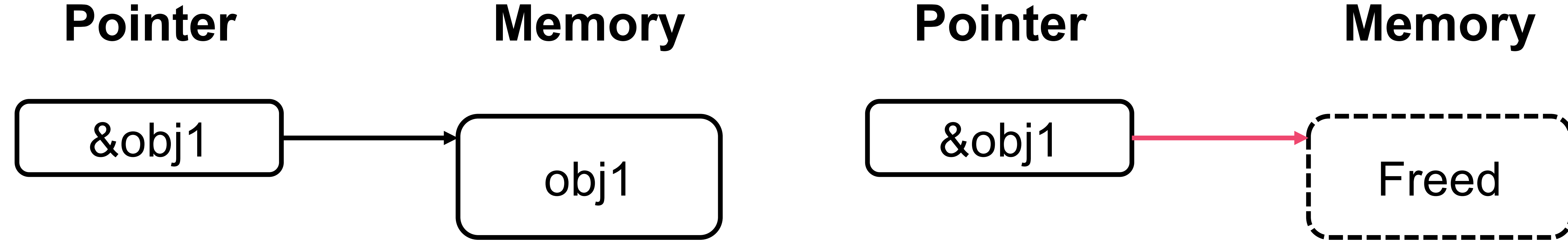
BLASTPASS (2023)

NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild

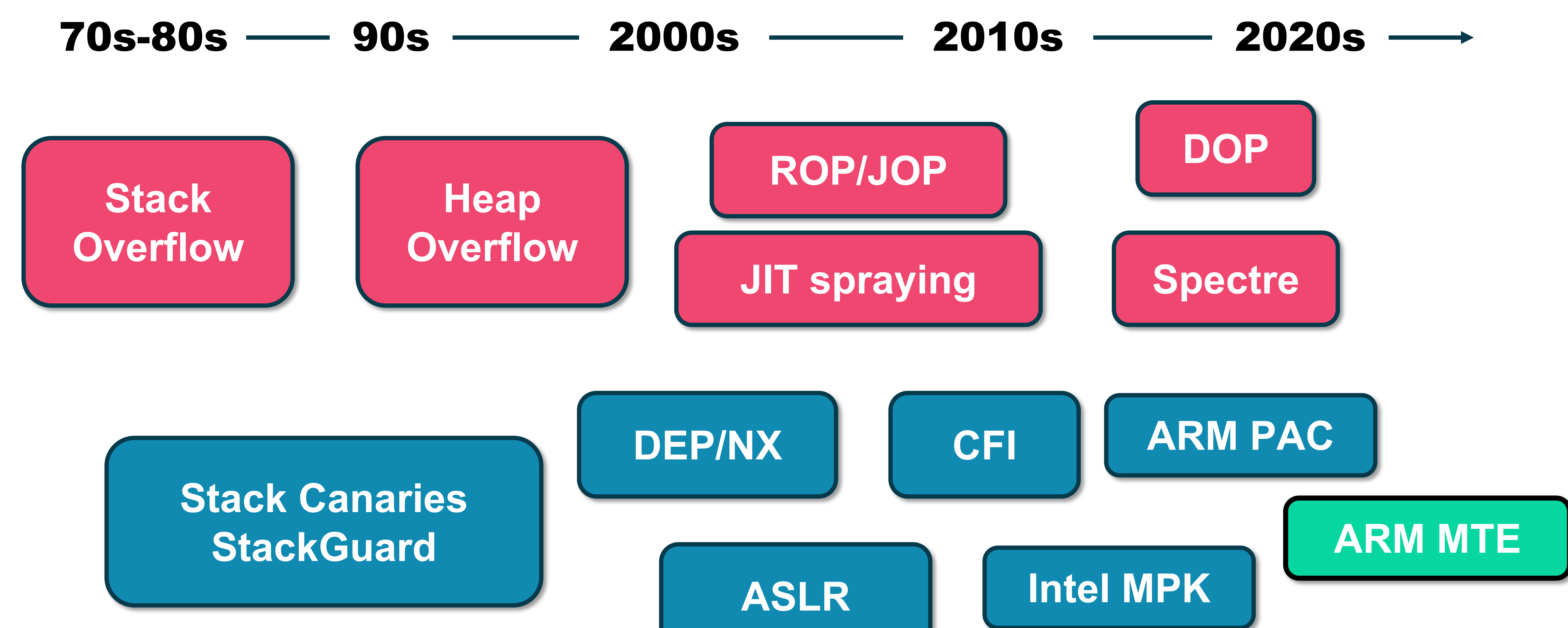
What is Memory Corruption?



What is Memory Corruption?



Attack and Defense Techniques



Google Pixel 8 / 8 pro — First MTE hardware released in Sep. 2023

*"**MTE** being one **key** feature that is delivering **secure mobile experiences**"*
- Arm (Feb 2023)

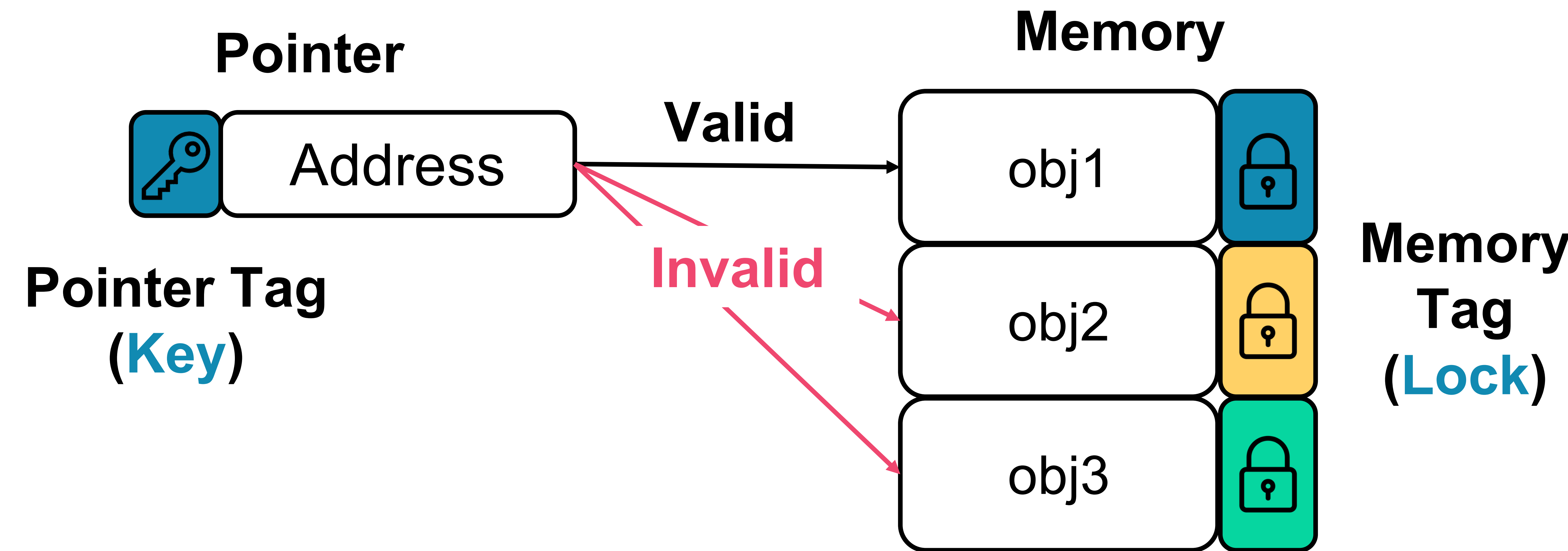
*"**MTE** is still by far the most promising path forward for improving C/C++ software security"*
- Google Project Zero (Aug 2023)

*"**Memory tagging** has the potential to provide good value both for **discovering vulnerabilities** and as **a mitigation for vulnerabilities**"*
- Microsoft (Mar 2020)

Why is MTE so Special?

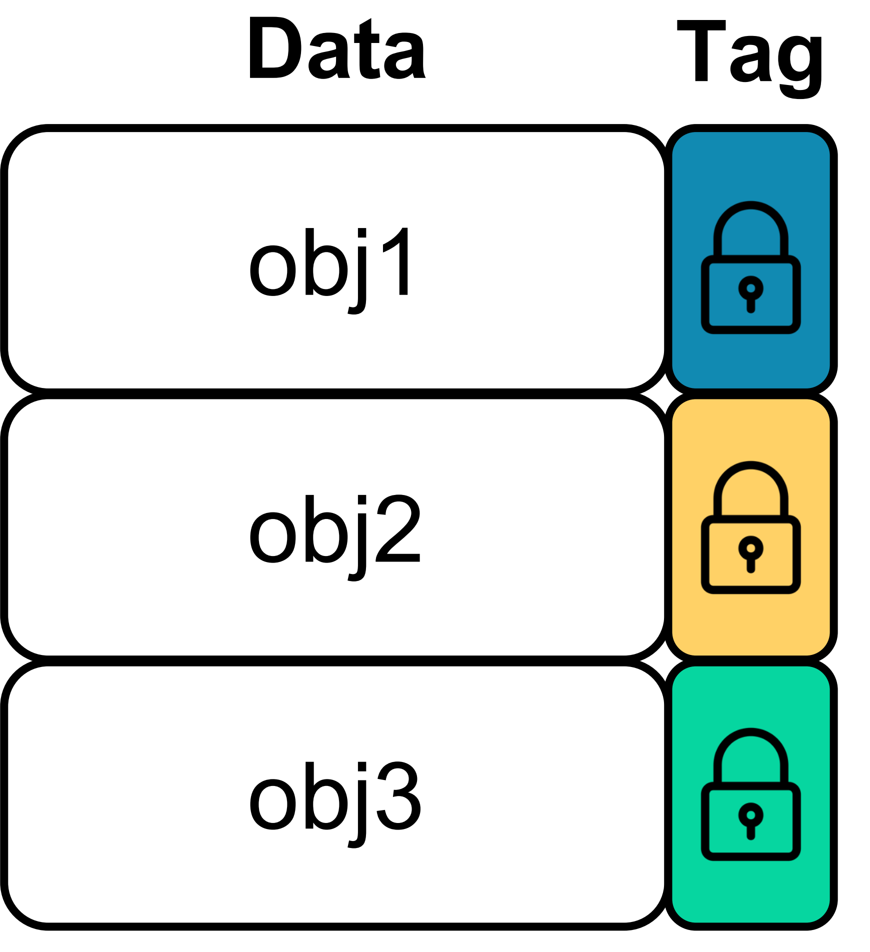
Hardware-based
Memory Corruption Detection
Fast and Compatible

ARM Memory Tagging Extensions



(1) Memory Tag

Dedicated memory region stores a 4-bit tag per 16-byte data



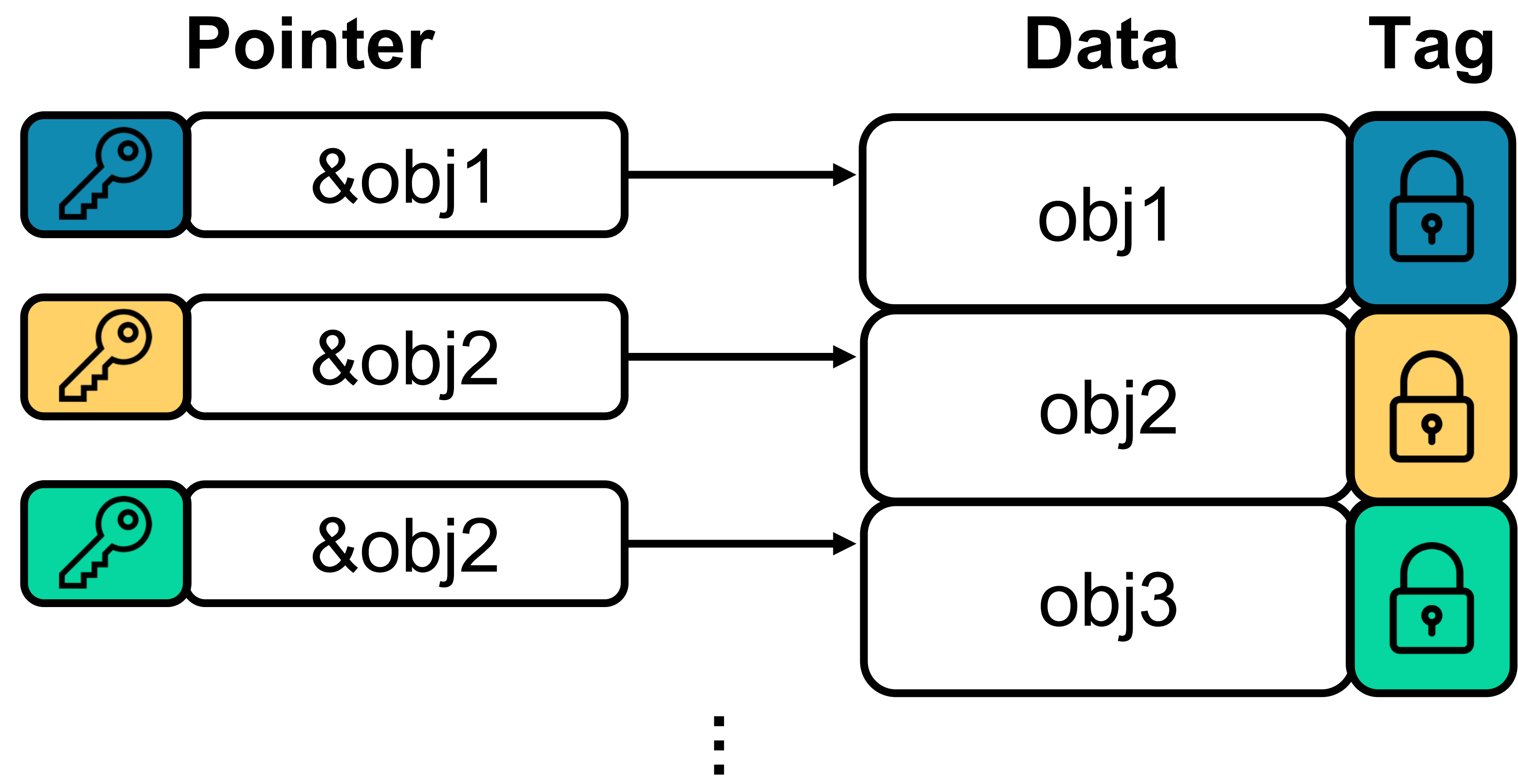
(2) Pointer Tag

Pointer Tag
(Key)

A pointer stores a 4-bit tag in its unused space

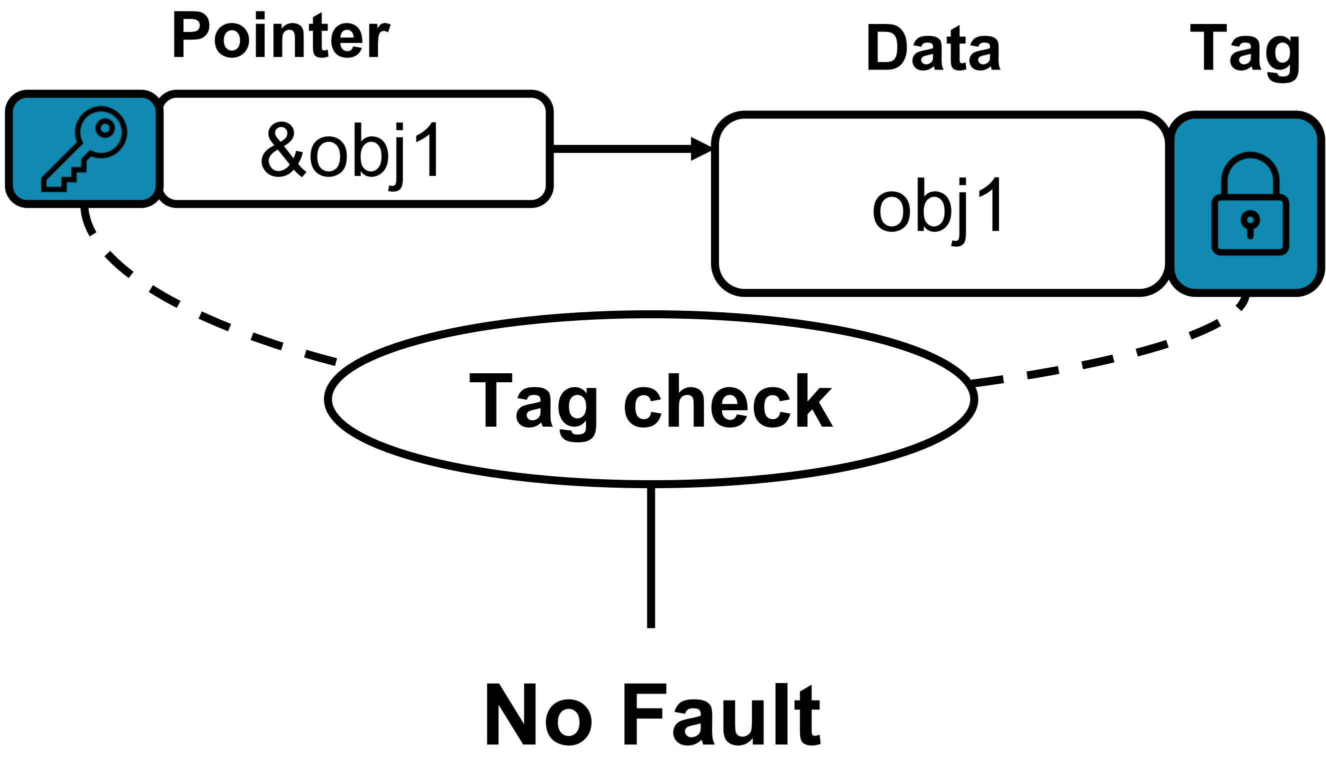
(3) Tag Allocation

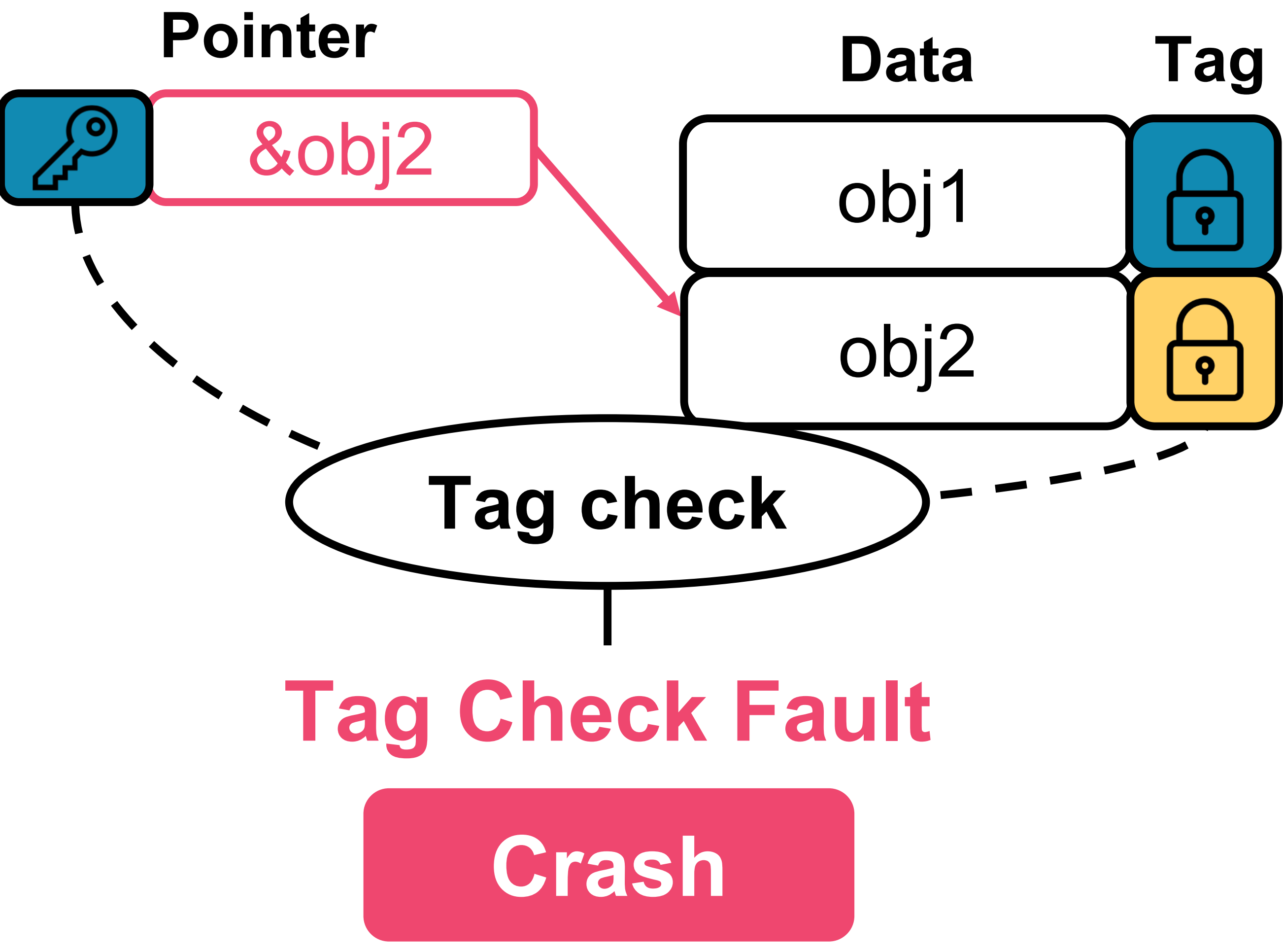
New instructions to create a random tag and load/store memory tags



(4) Tag Check

Transparently done by hardware

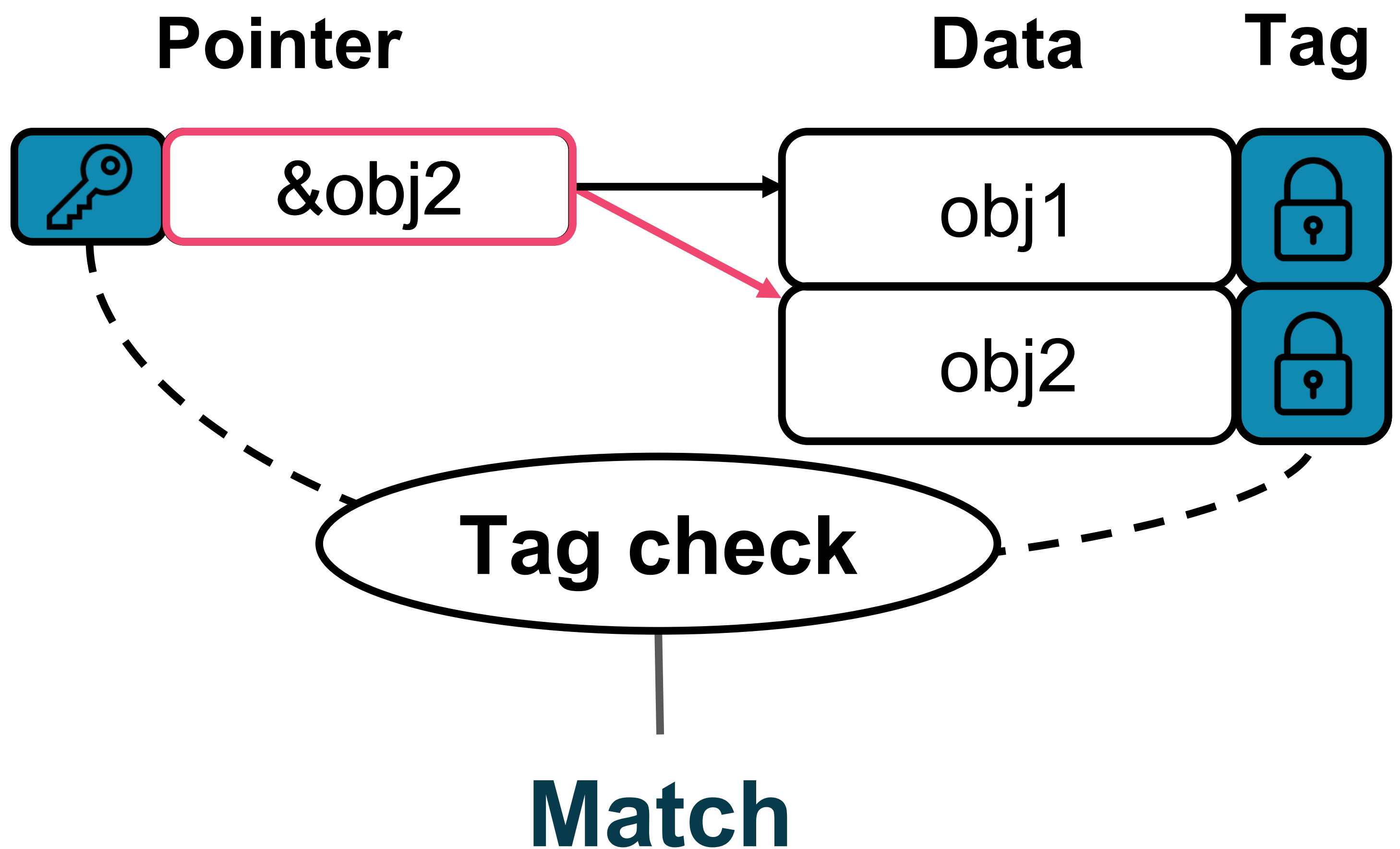




How to Bypass MTE?

(1) Tag Collision (16 possible tags)

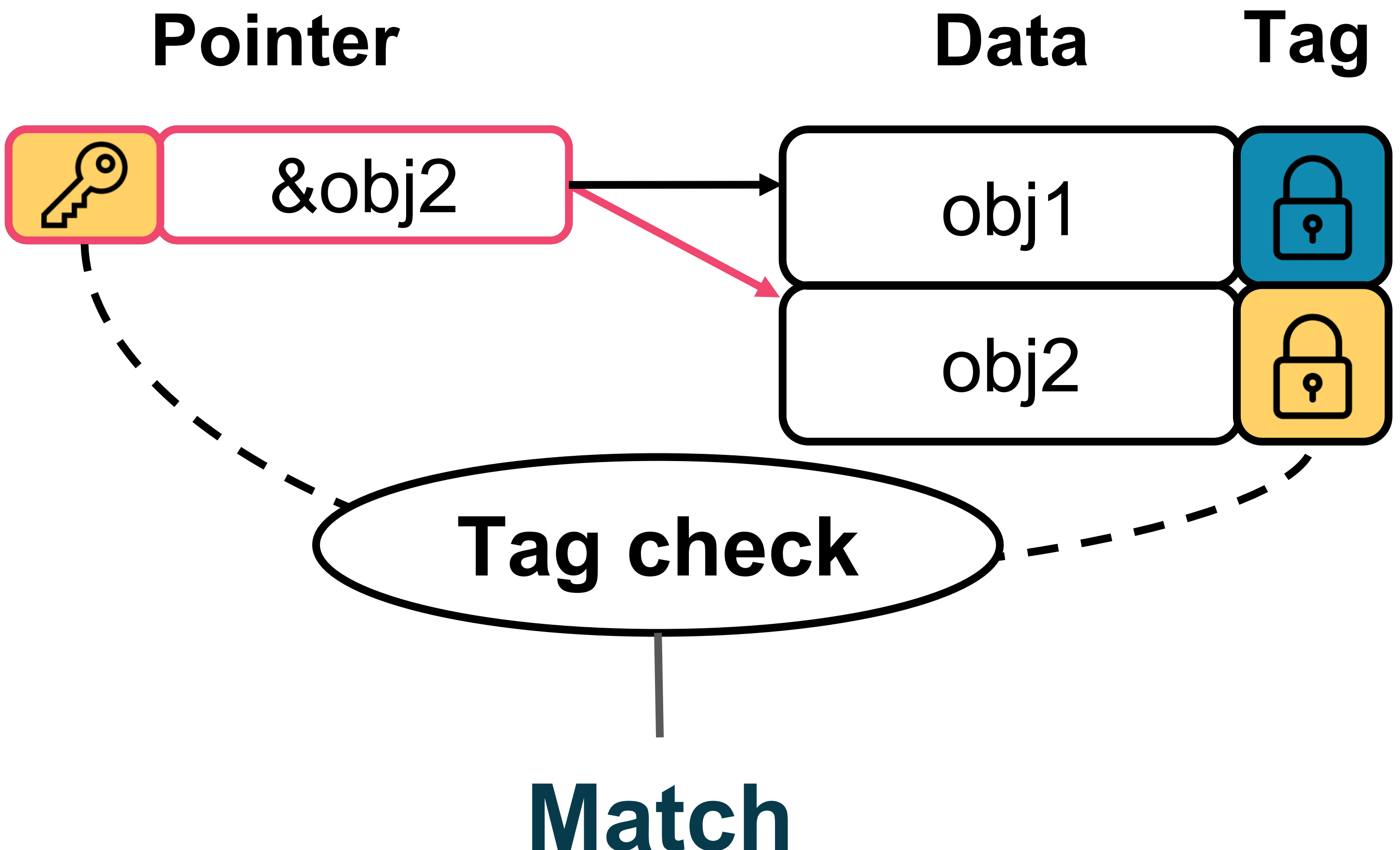
Wait until the pointer tag matches the target memory tag



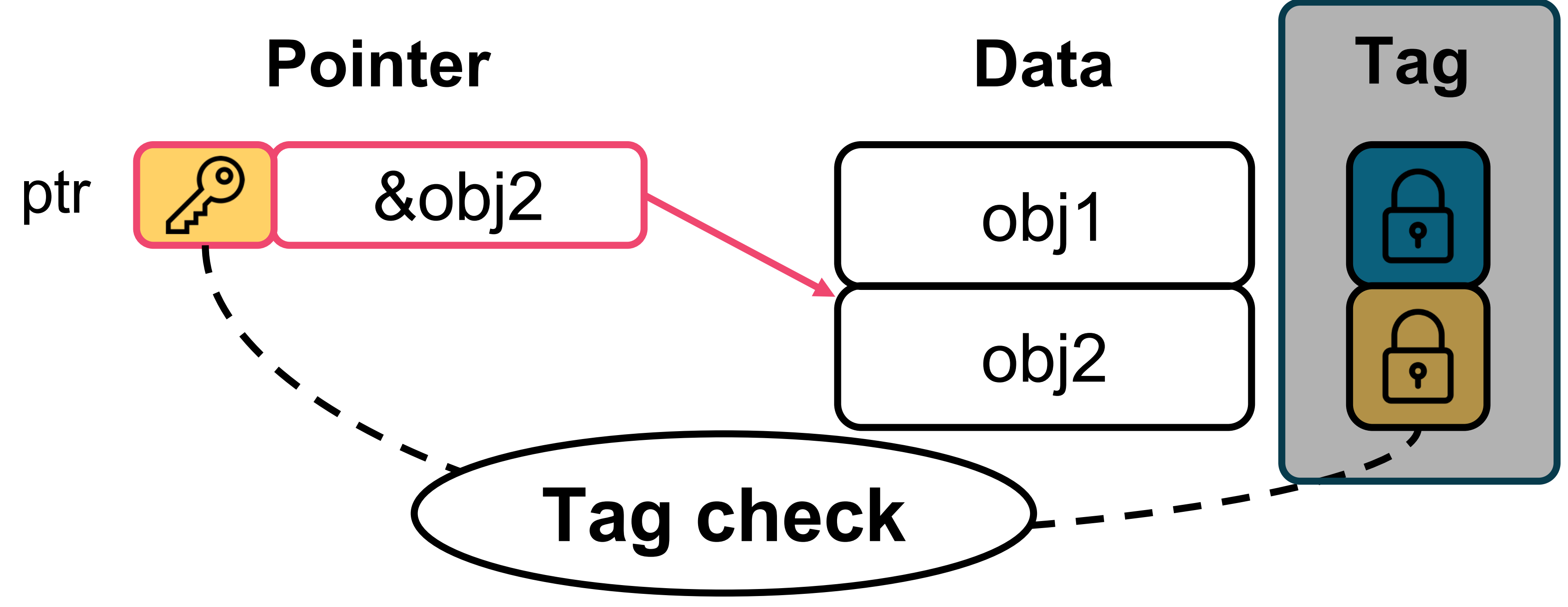
How to Bypass MTE?

(2) Pointer Tag Corruption

Corrupt the pointer tag to the target memory tag

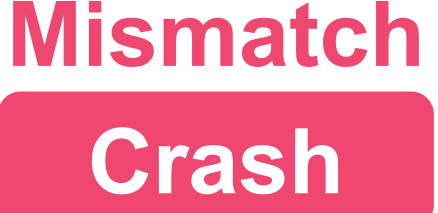


Challenge: Random Tags



Match ⇒ ***Attack Succeeds 1/16 (6%)***

Mismatch



⇒ ***Attack Fails 15/16 (94%)***

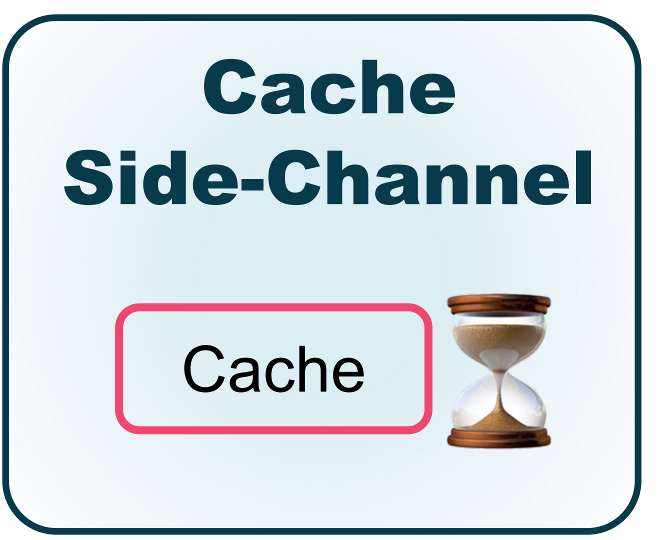
MTE Bypass Requirement

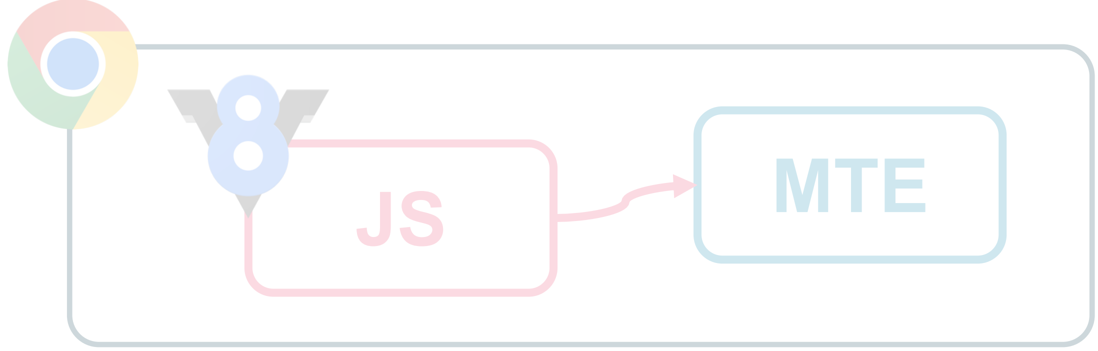
A reliable way to leak the MTE tag of any address

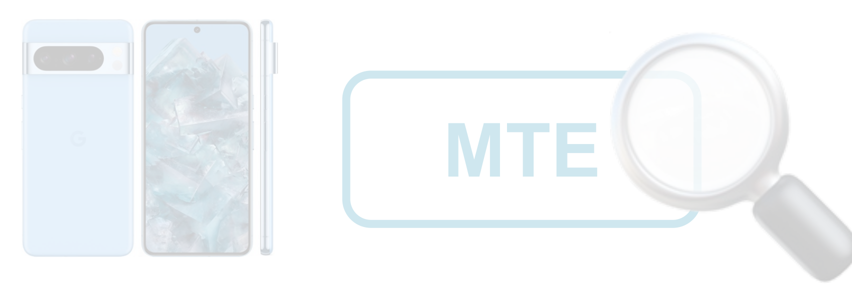
Approach

- Leak tag check result from Cache Side-channel
- Exploit Speculative Execution to avoid crash

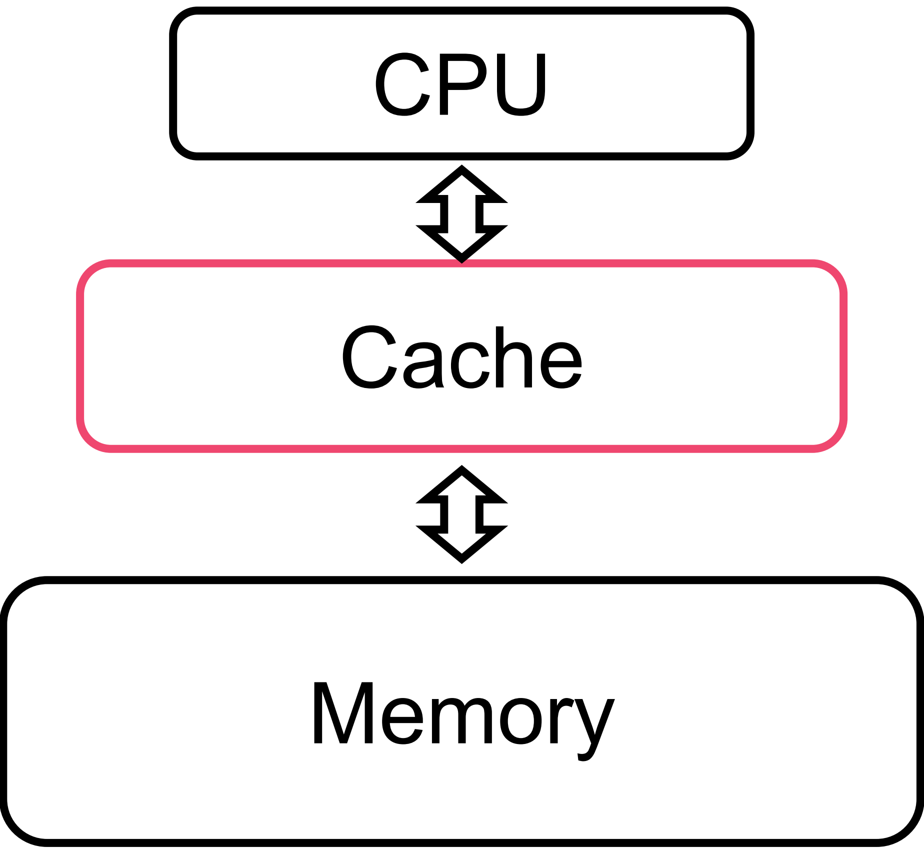
Roadmap







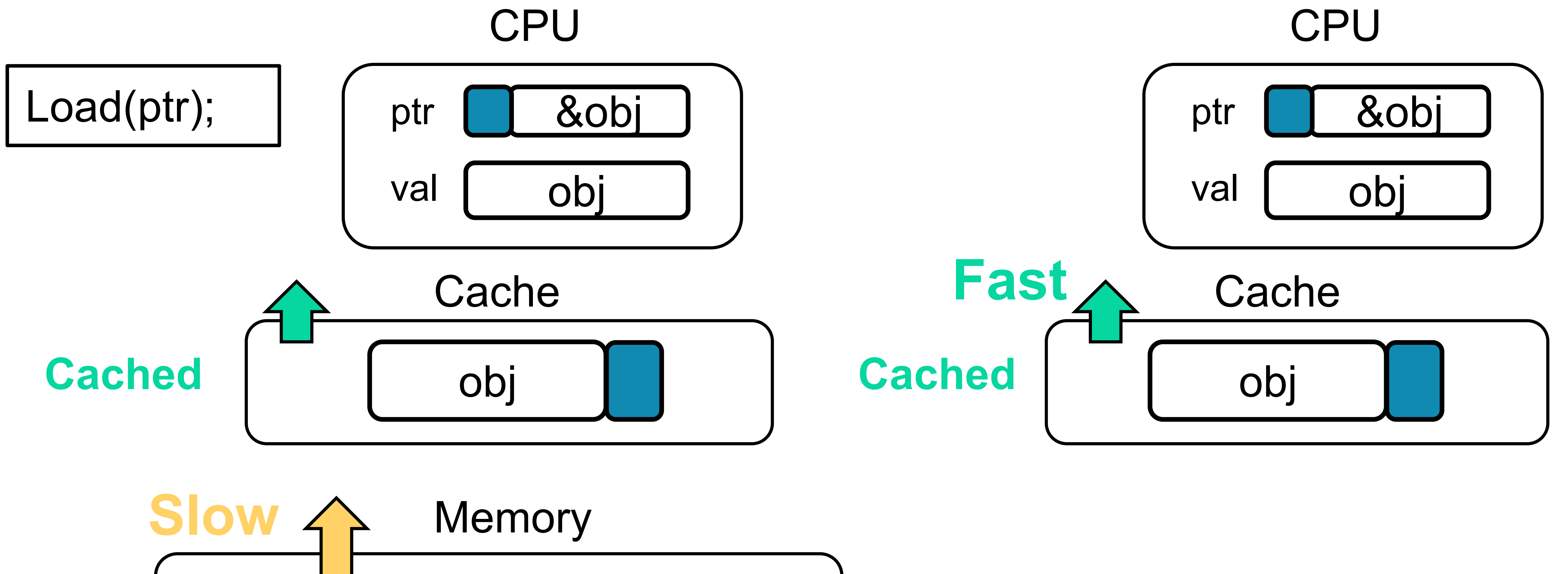
What is Cache?



What is Cache?

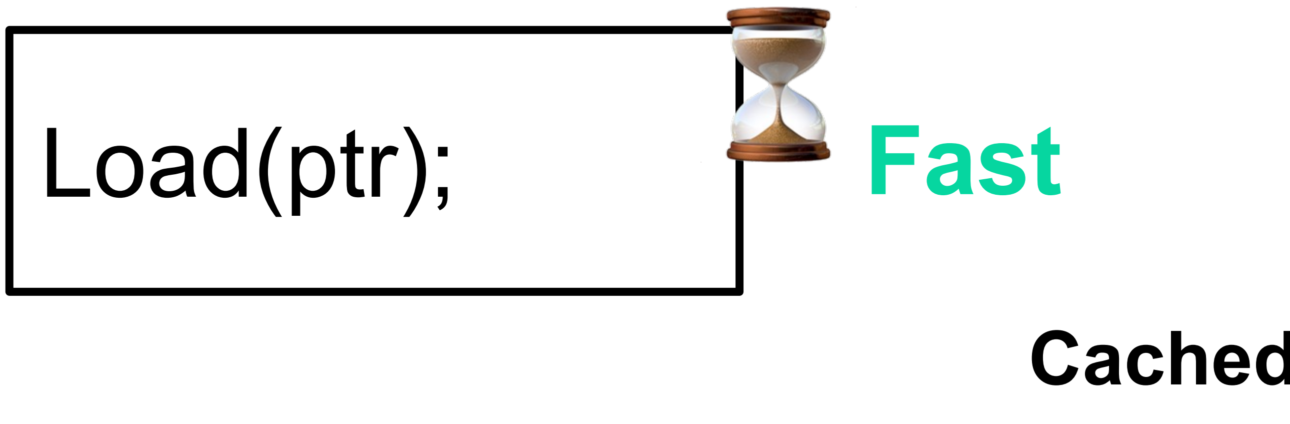
First Access : Slow

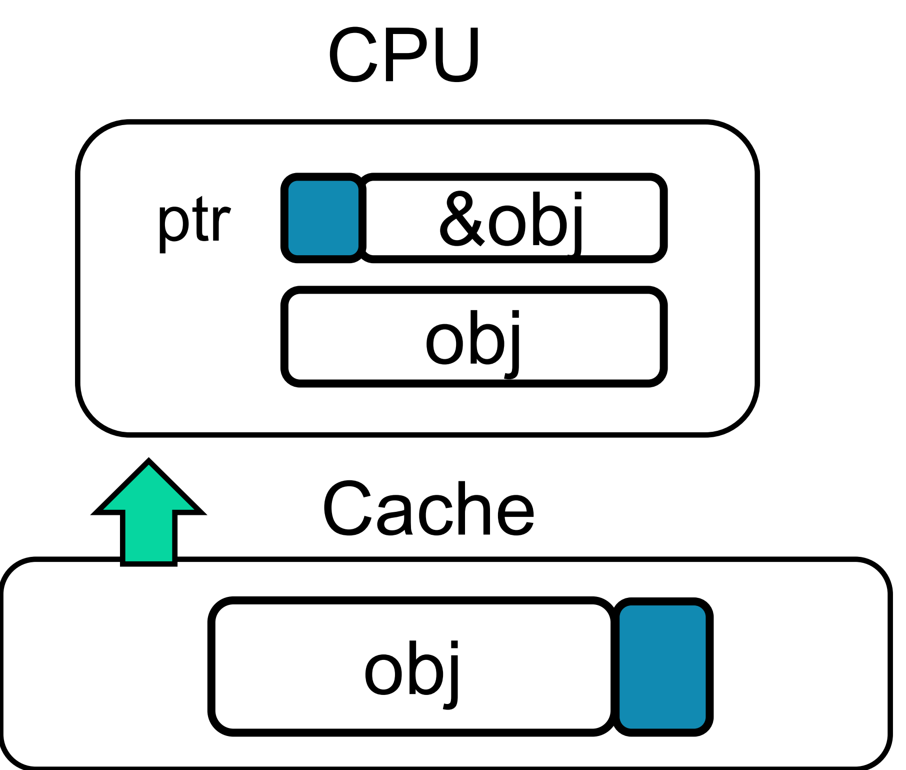
Second Access : Fast



Cache Side-Channel

Q. Has obj been accessed?





A. ptr has been accessed!

What is Cache Side-Channel?

Q. Has obj been accessed?

```
Load(ptr);
```

Slow

CPU

ptr &obj

obj

Cache

Not Cached

obj

A. ptr has NOT been accessed!

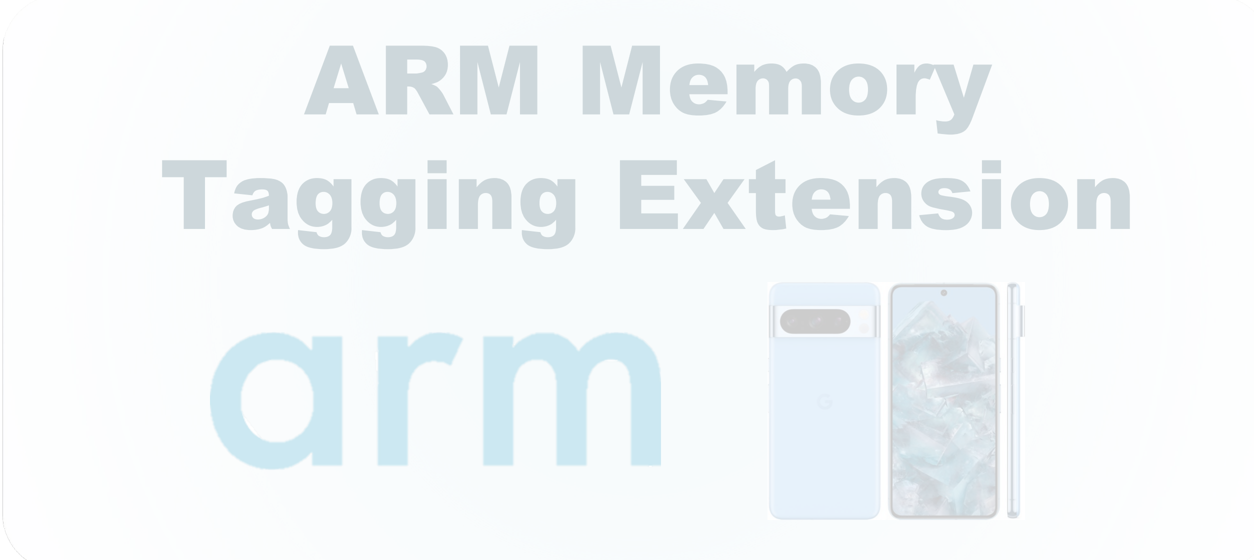
Memory

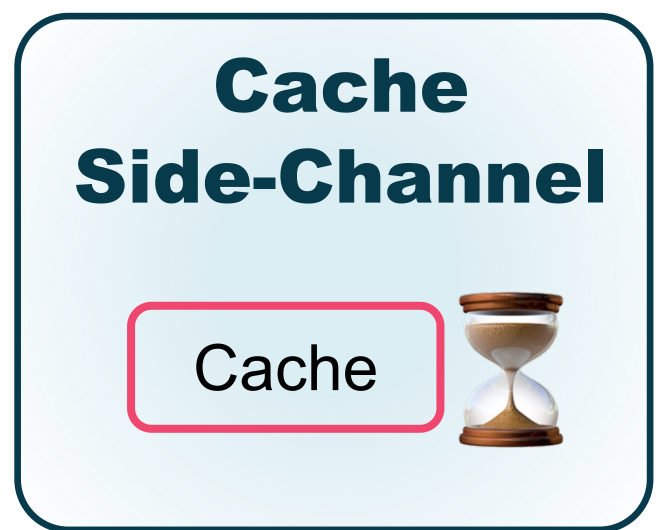
obj

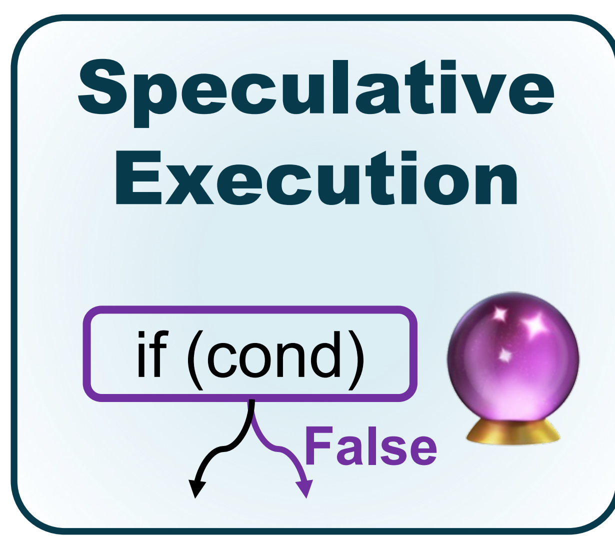
Exploit cache side-channel

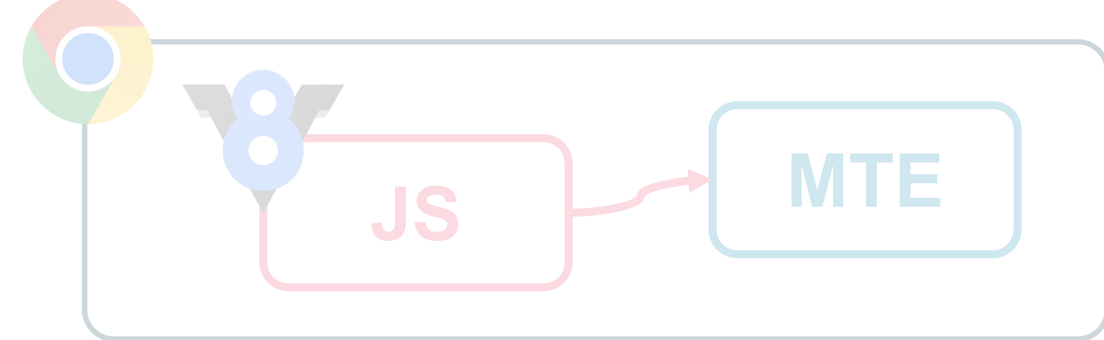
→ Leak whether an address is accessed

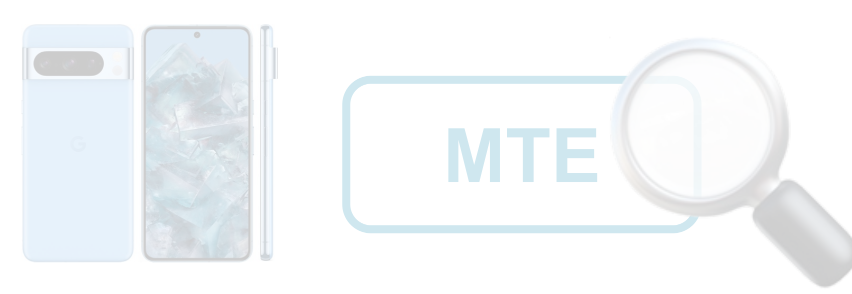
Roadmap





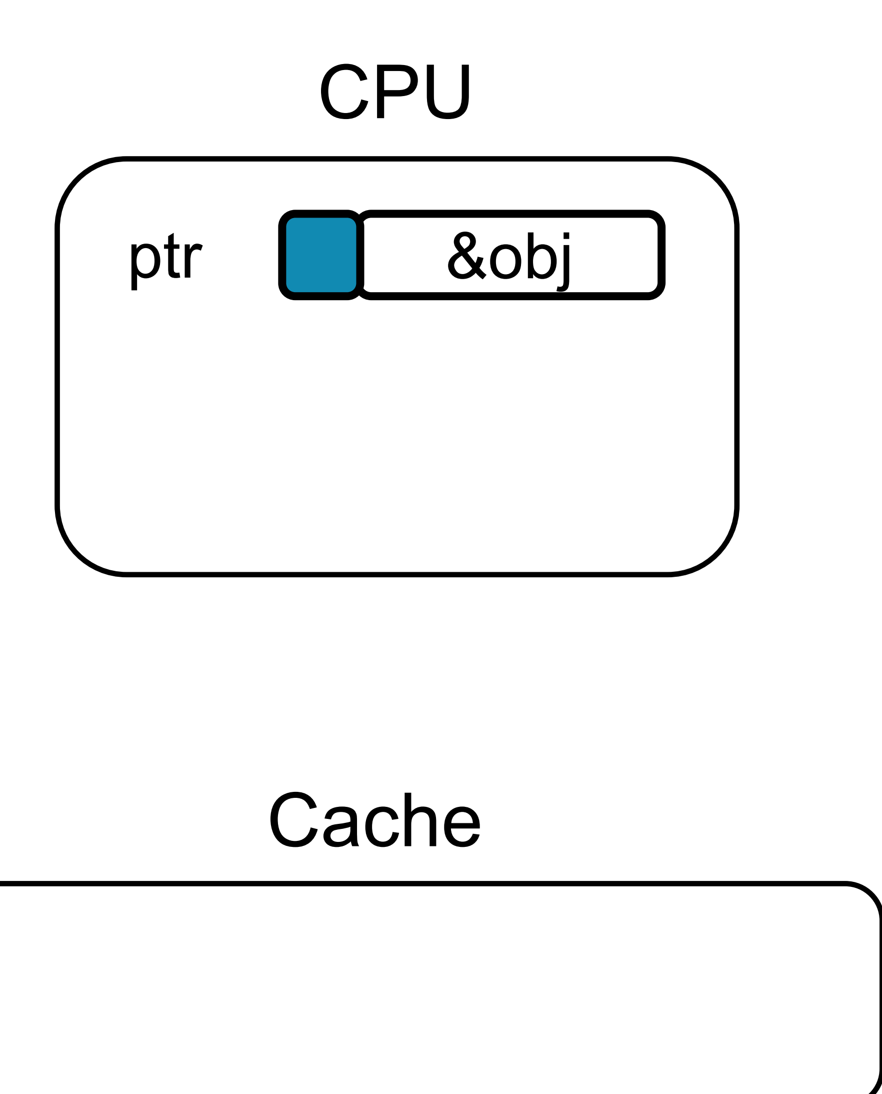




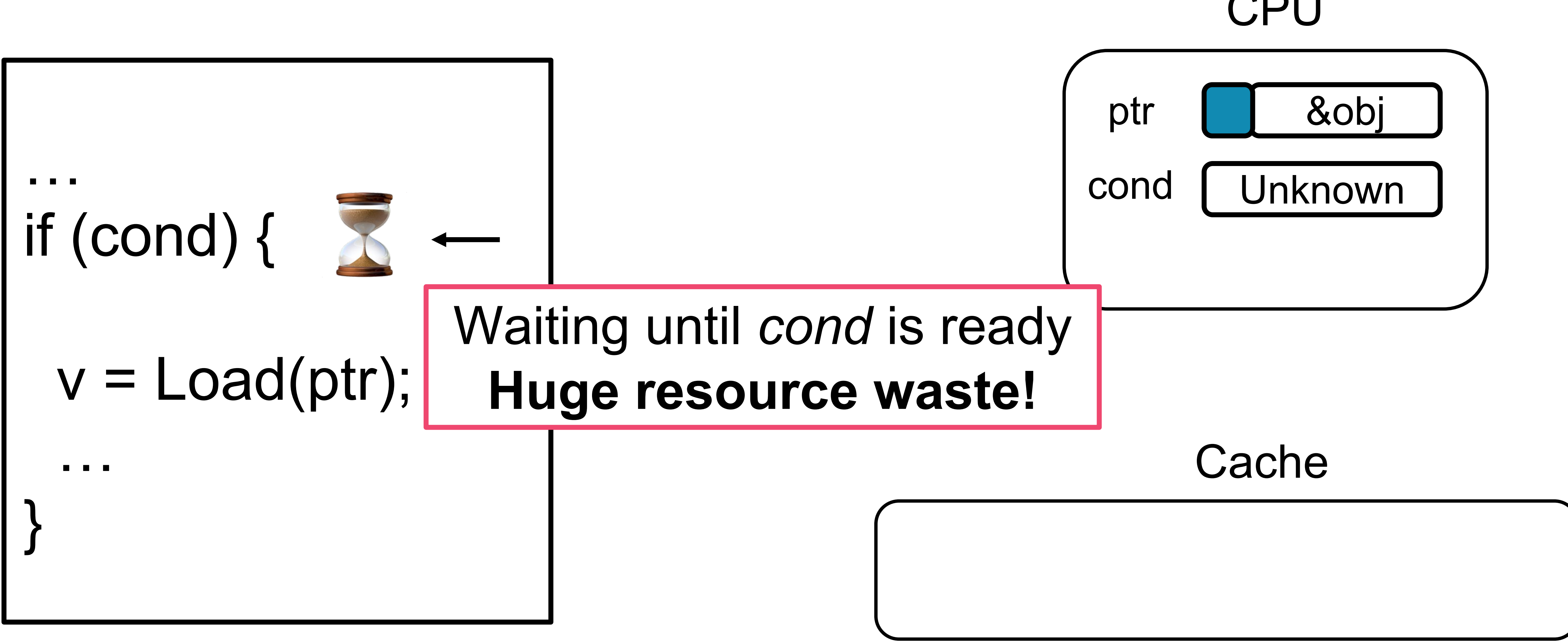


What is Speculative Execution?

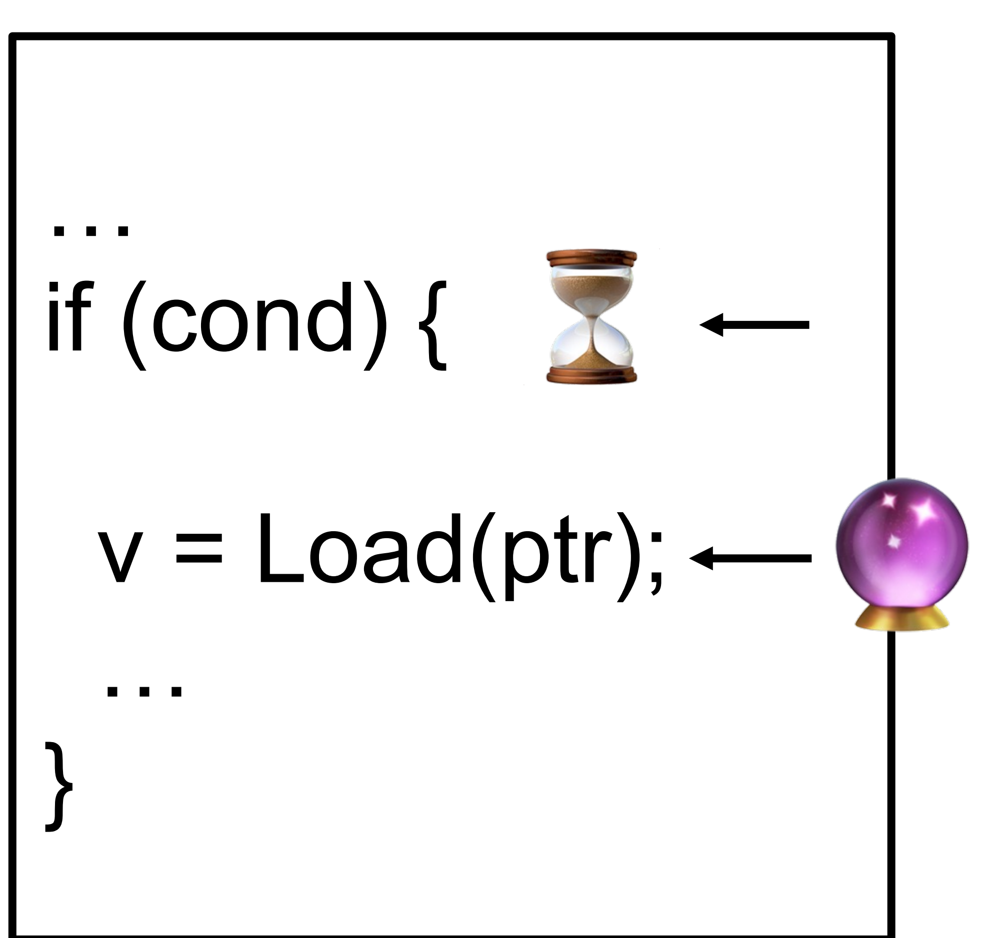
```
…
if (cond) {

  v = Load(ptr);
  …
}
```

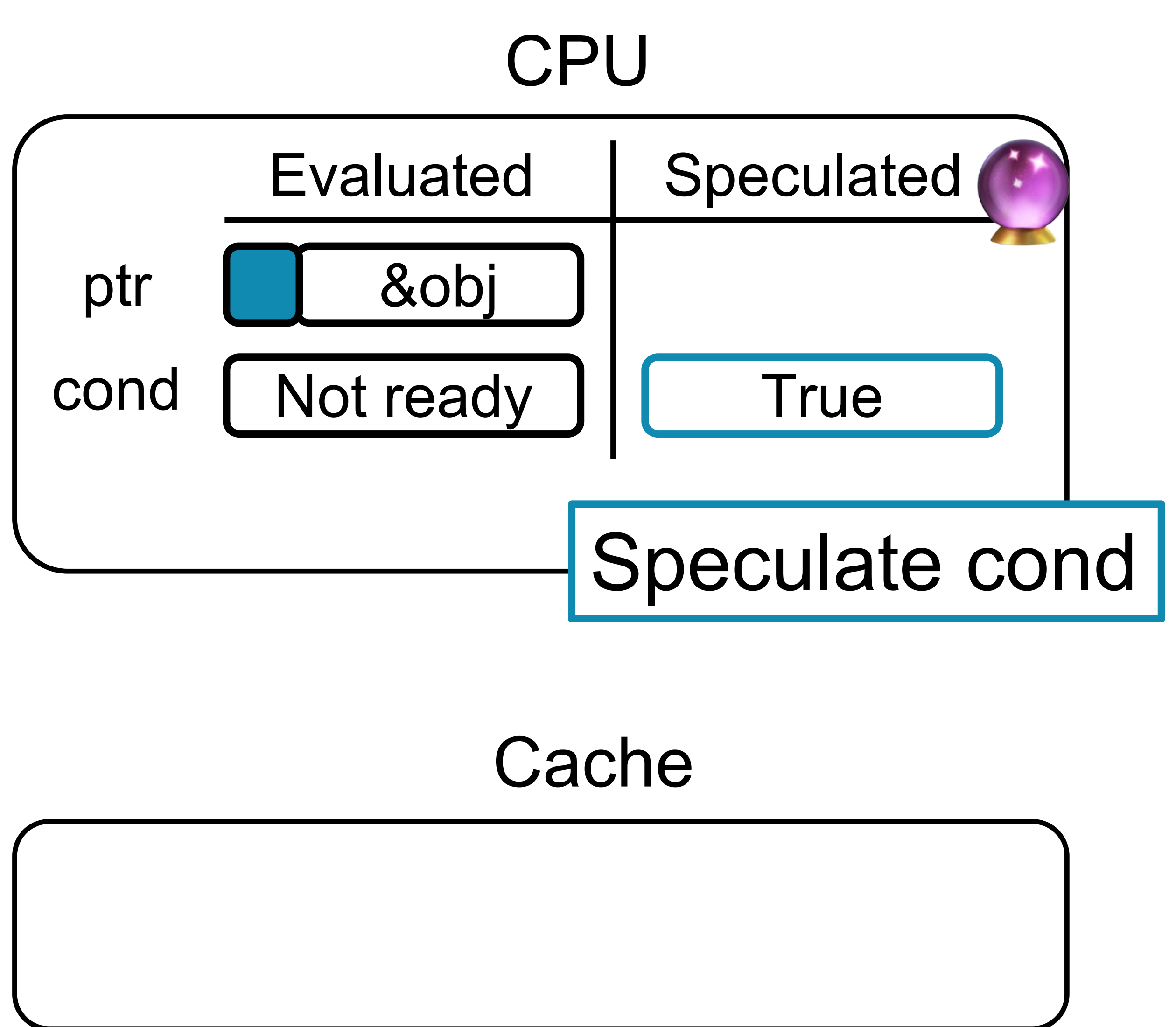


What is Speculative Execution?

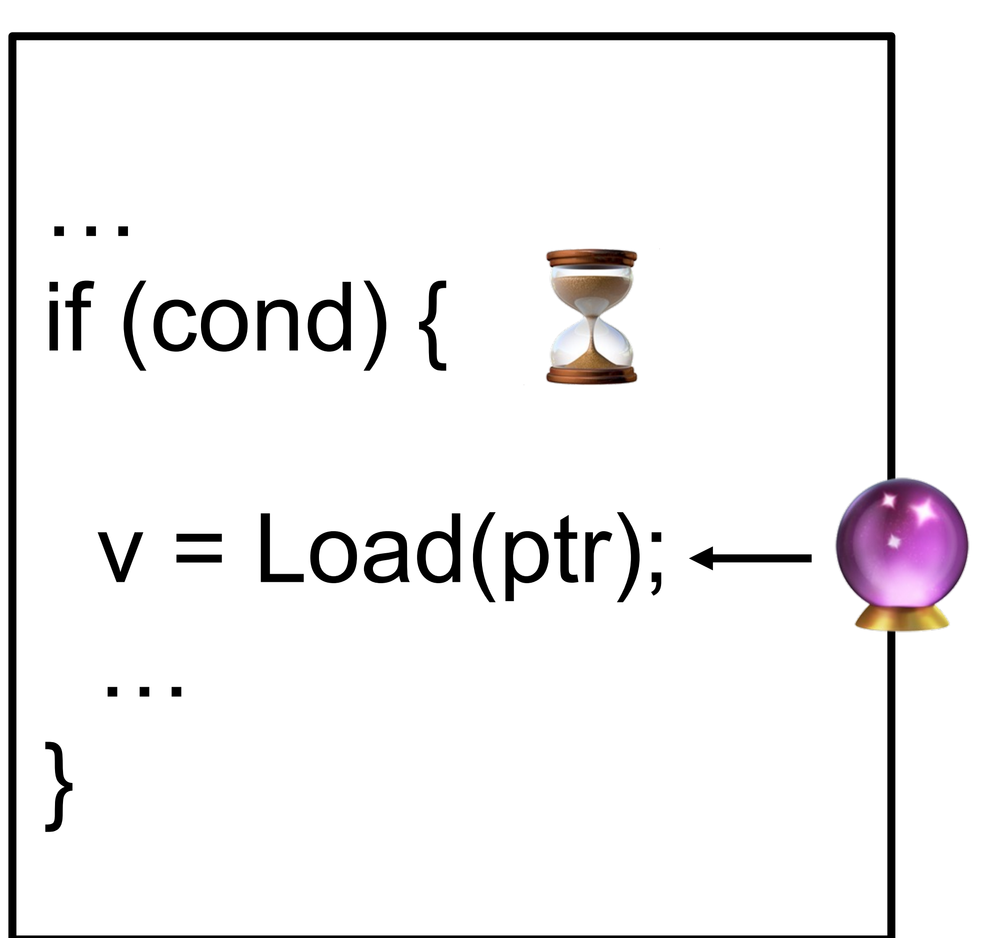


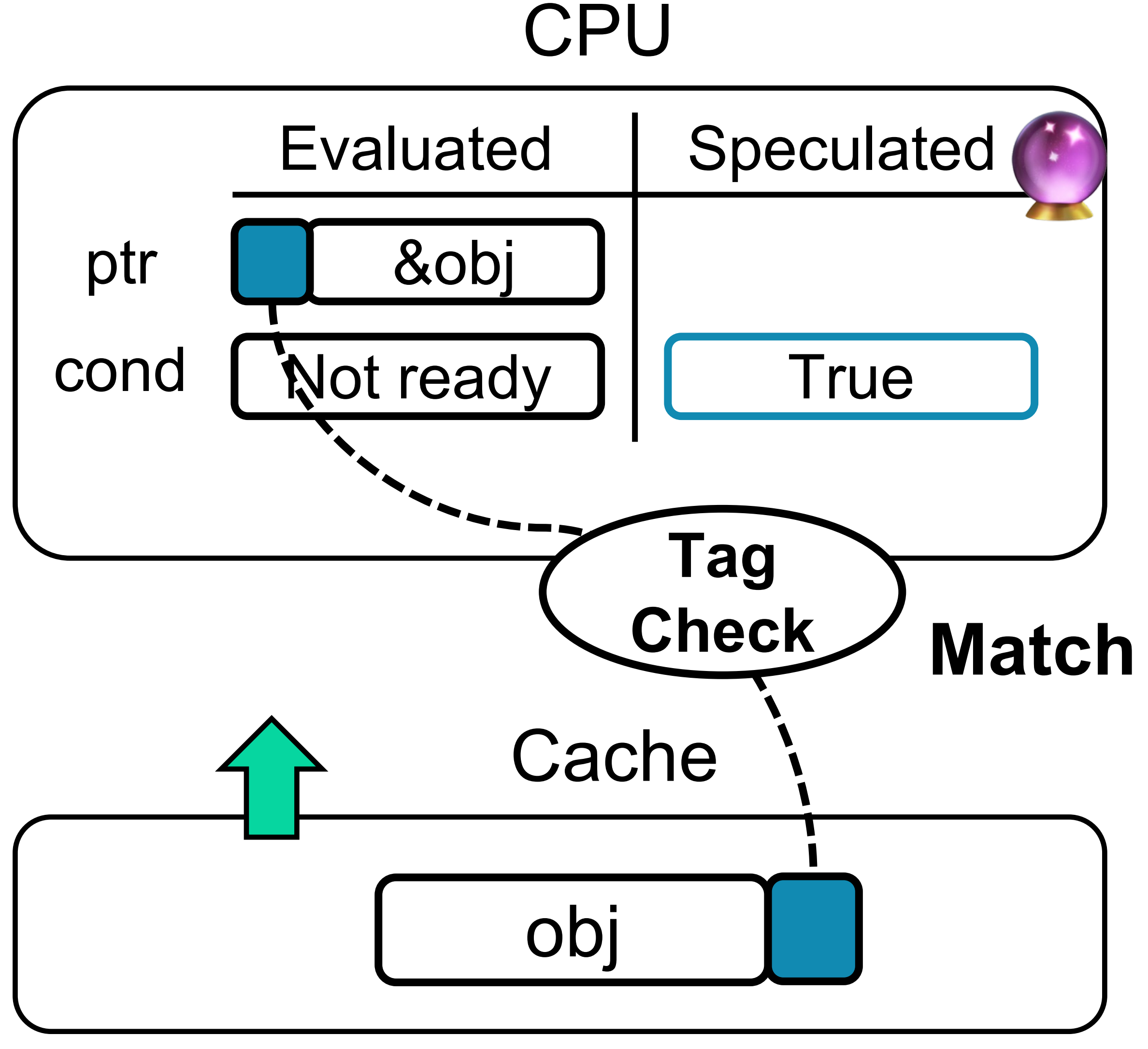
What is Speculative Execution?



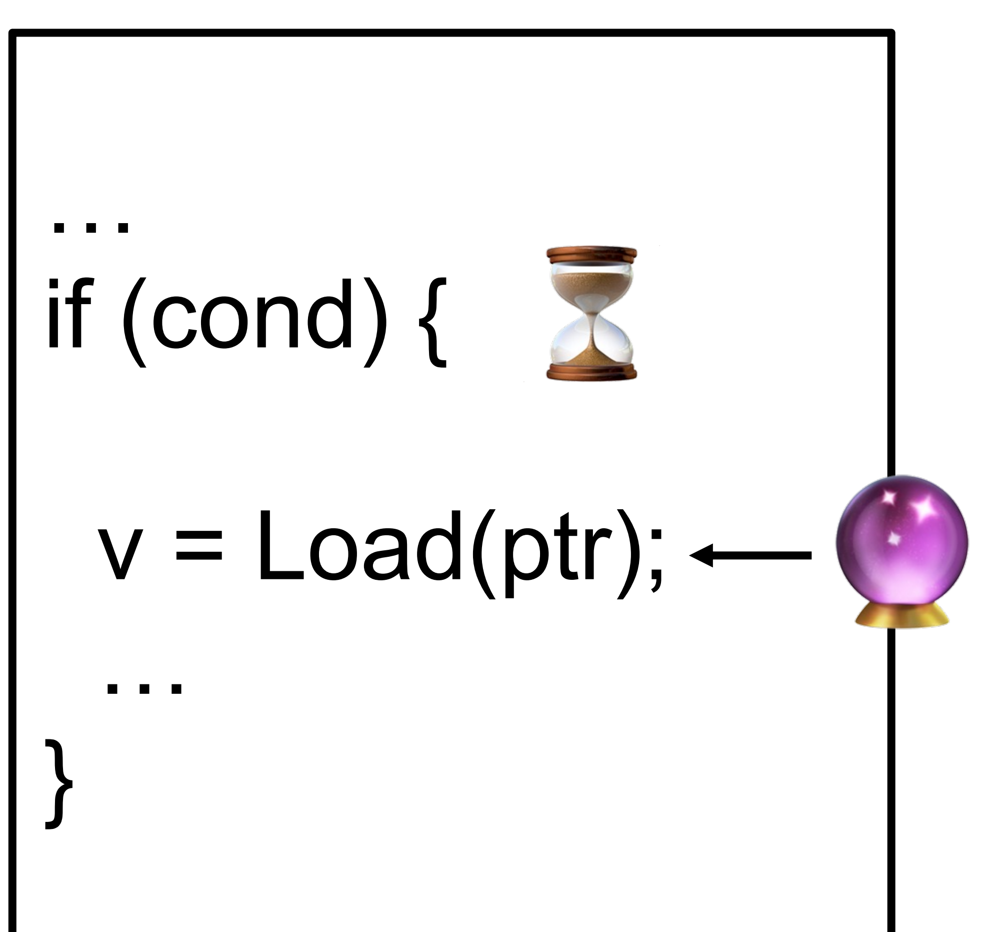


What is Speculative Execution?

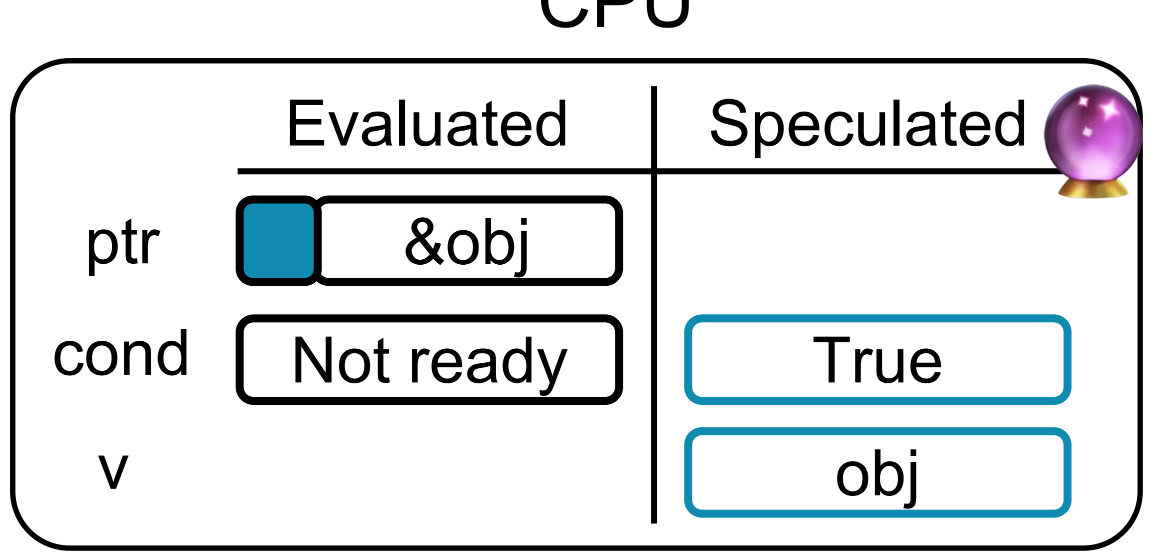




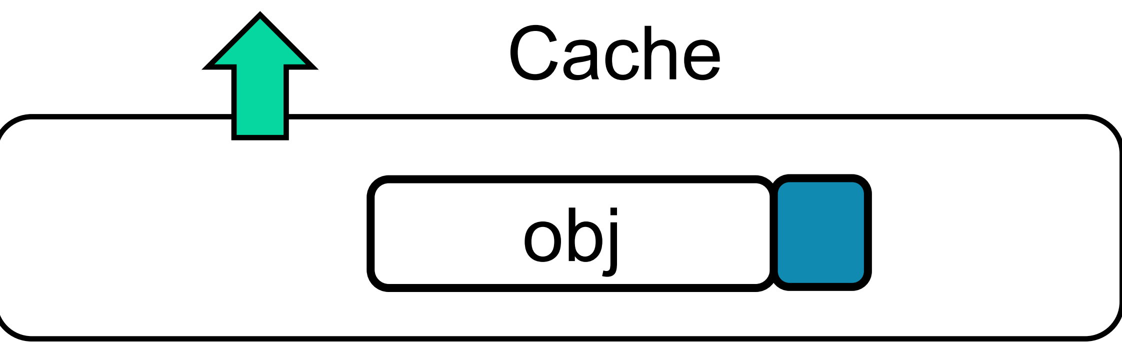
What is Speculative Execution?



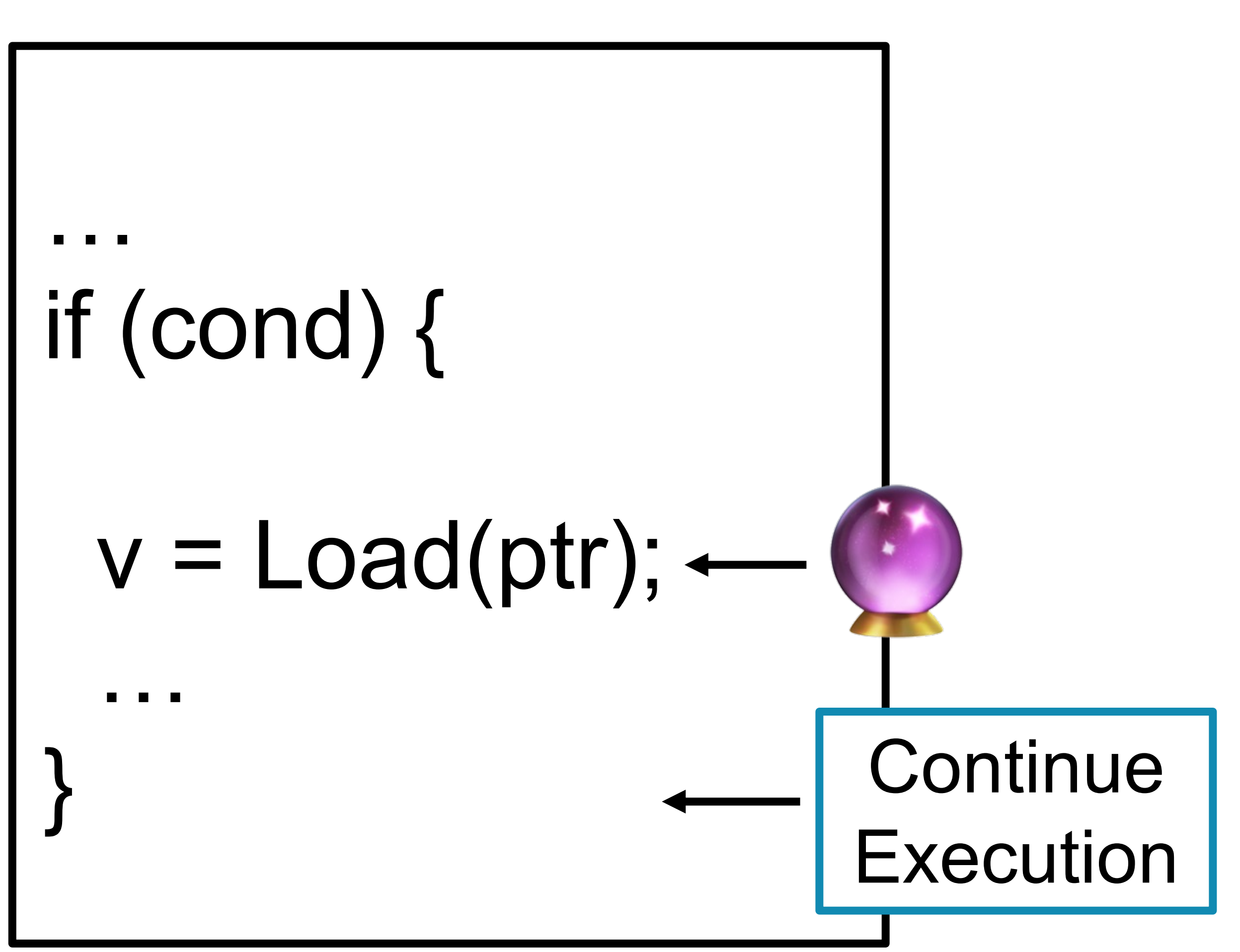
CPU

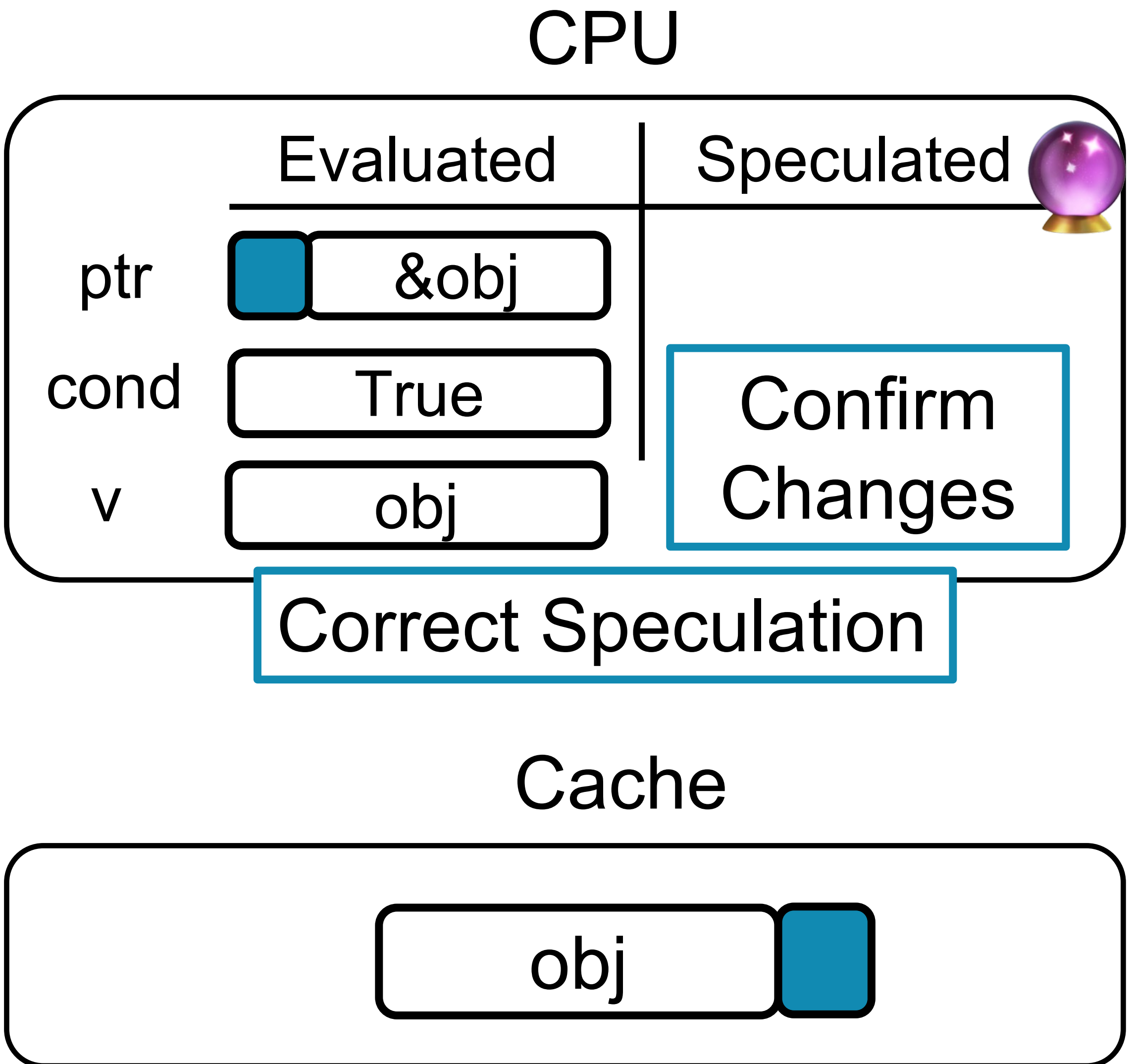


Cache

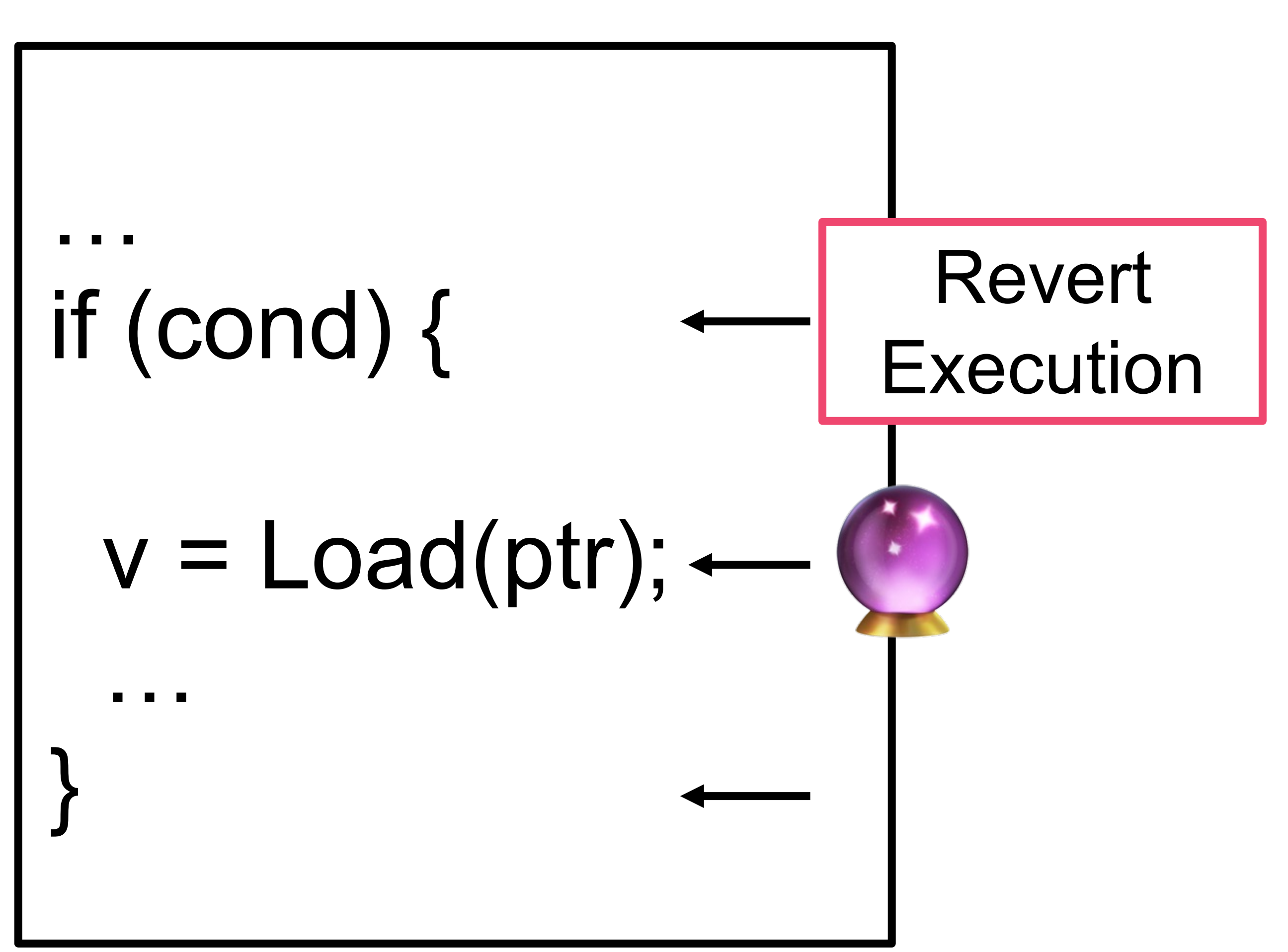


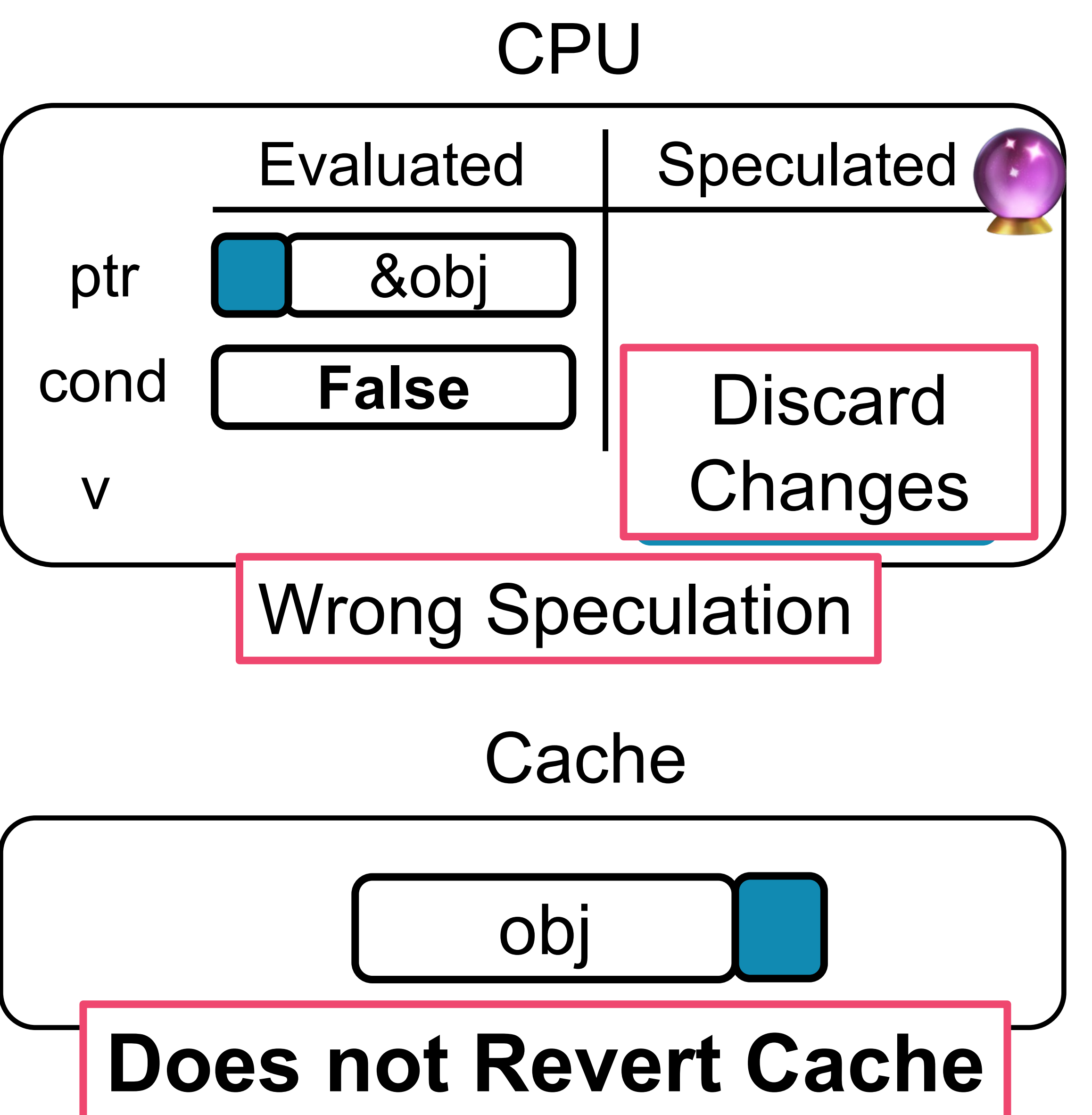
What is Speculative Execution?



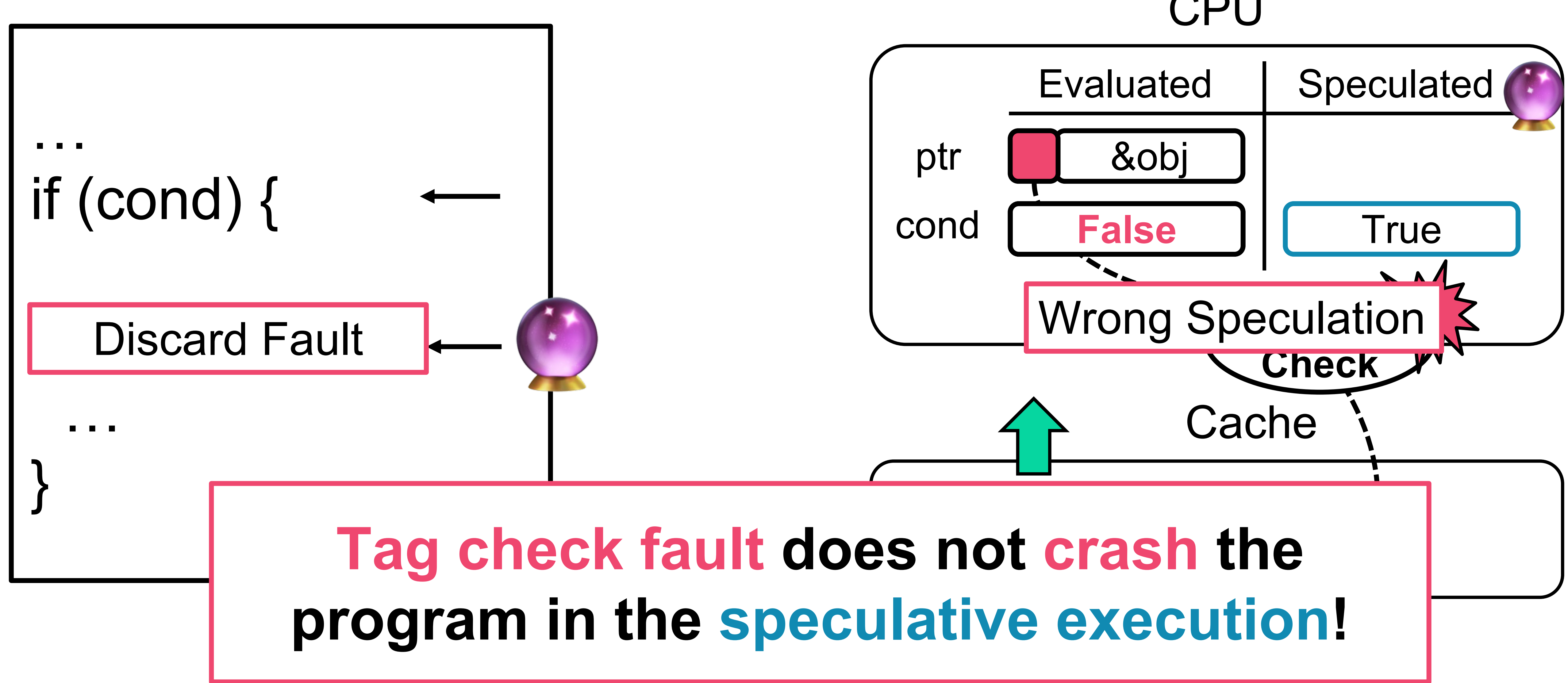


What is Speculative Execution?





Tag check fault on Speculative Execution?



Tag check fault does not crash the program in the speculative execution!

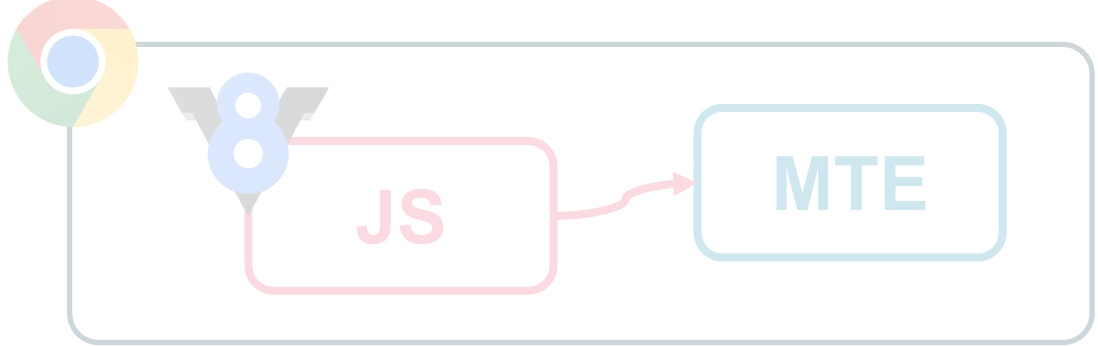
Exploit cache side-channel
→ Leak whether an address is accessed

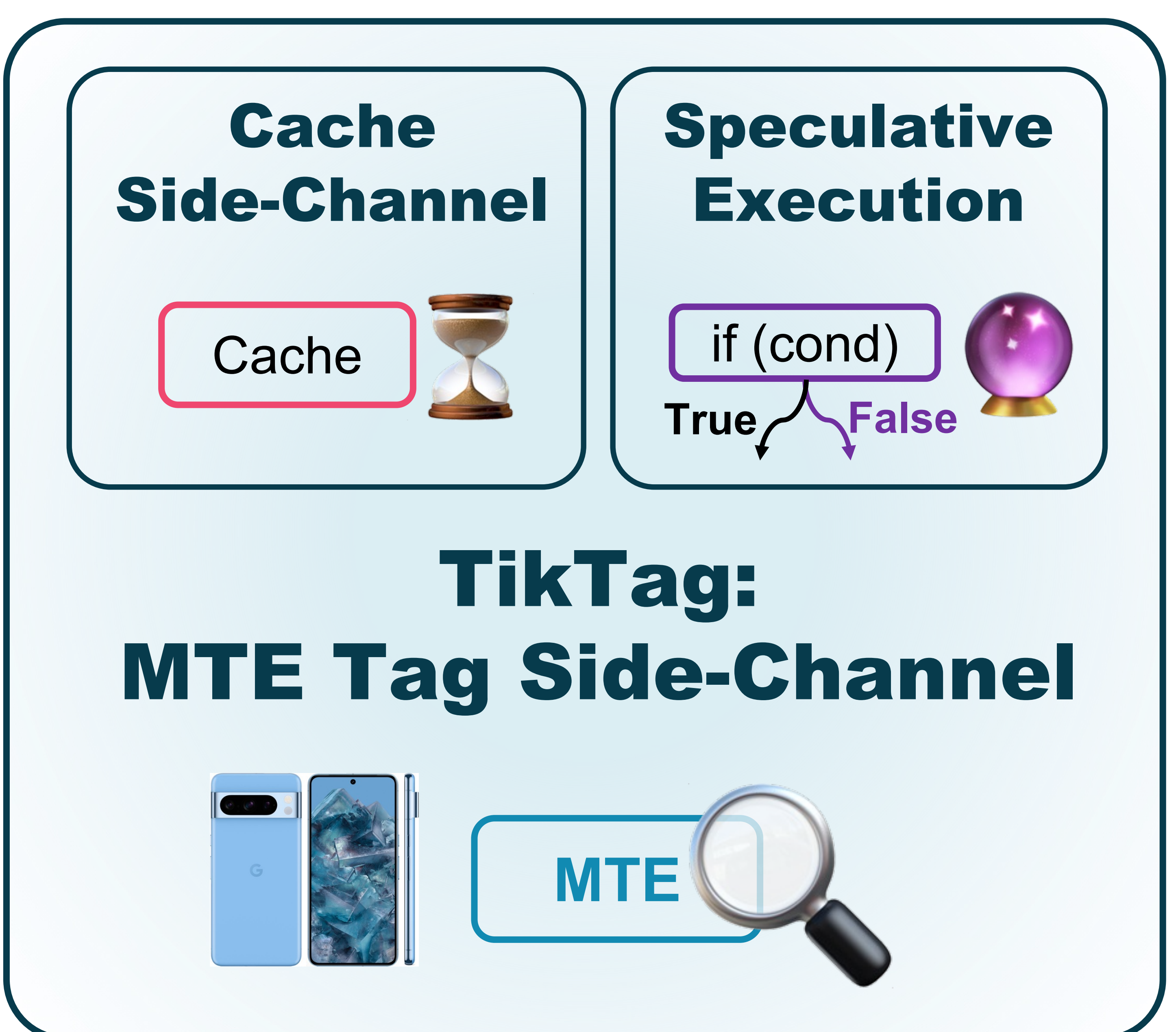
Exploit speculative execution
→ Avoid crash on tag check fault

Roadmap

ARM Memory Tagging Extension

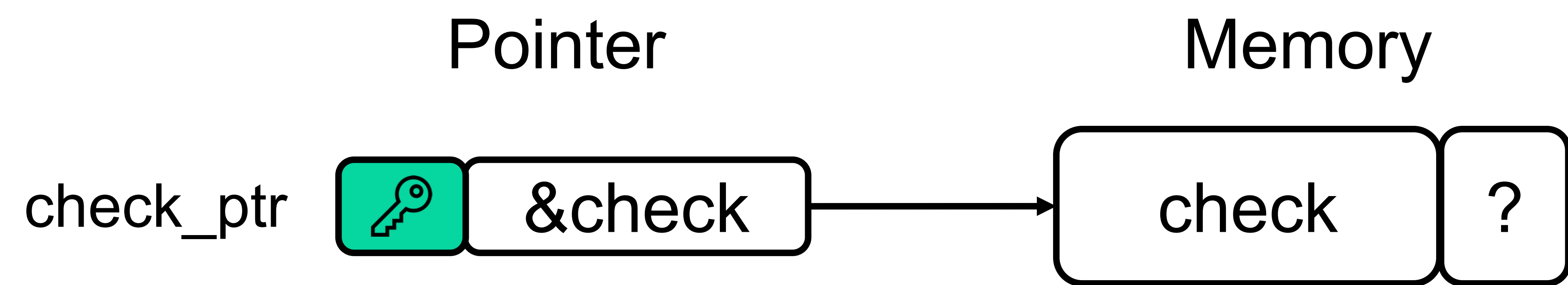
Real-world MTE Bypass Attack





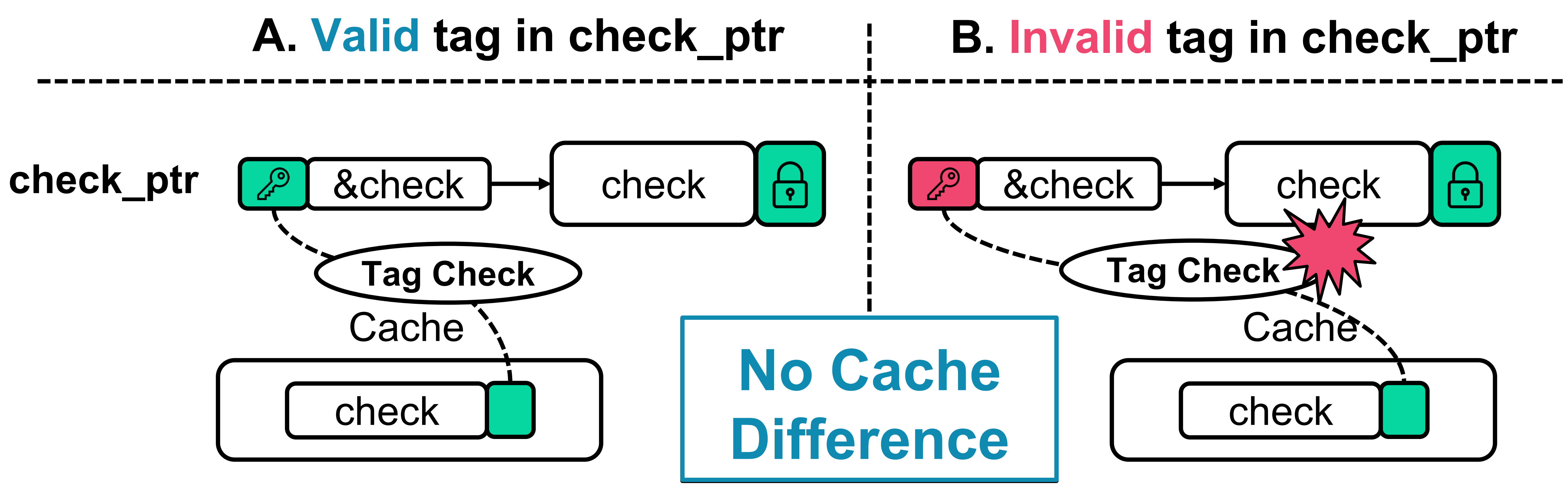
MTE Side-channel attack

Goal: Leak the memory tag given a pointer



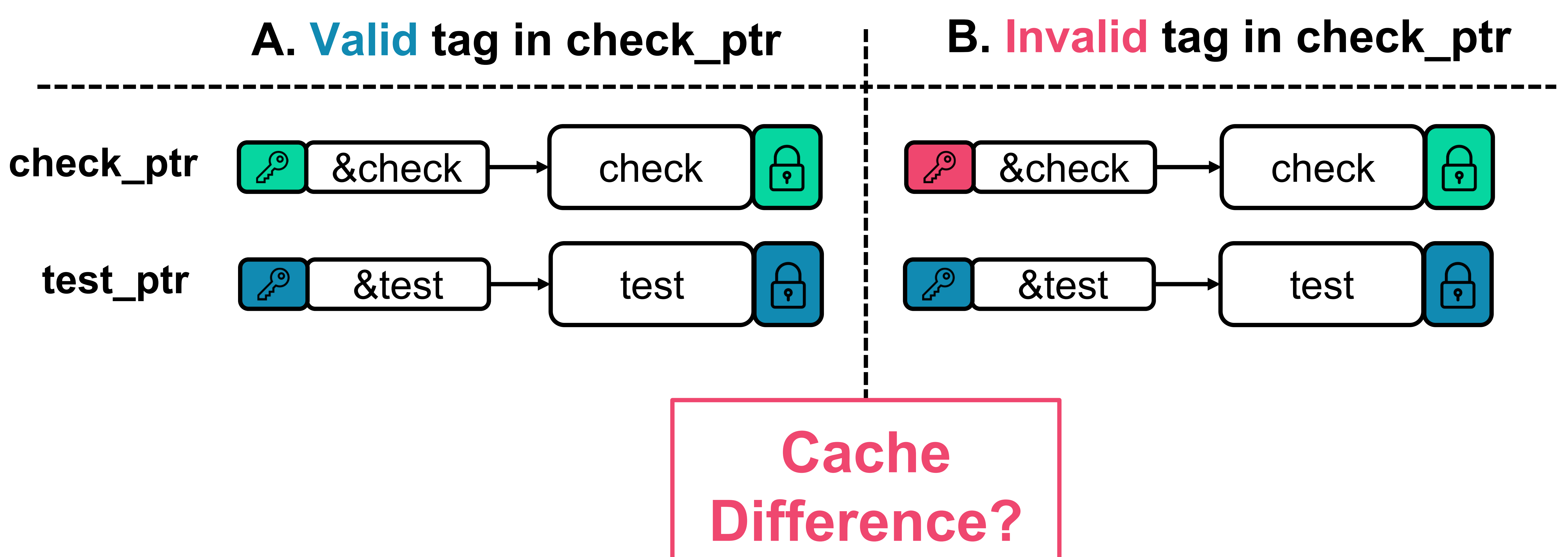
MTE Side-channel attack

Two test cases:

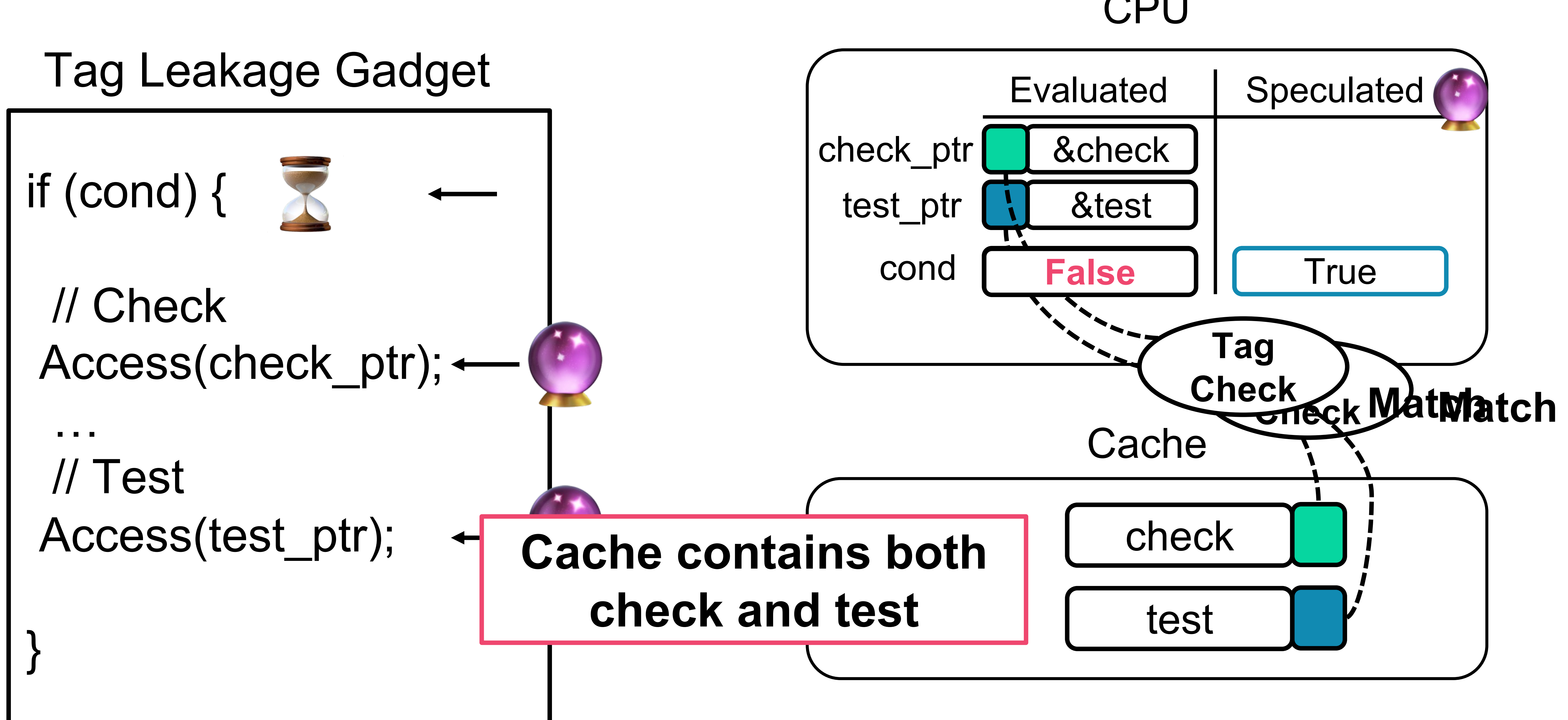


MTE Side-channel attack

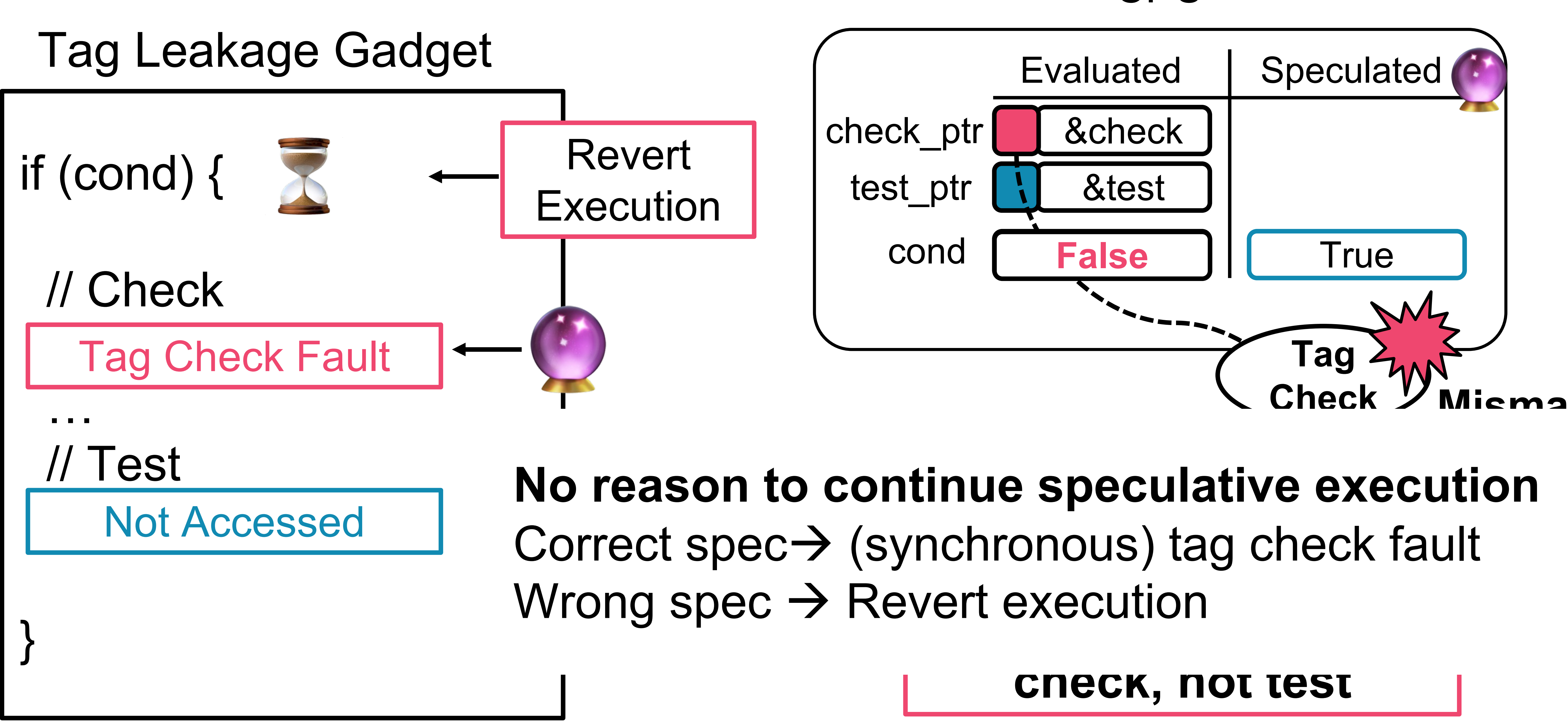
Two test cases:



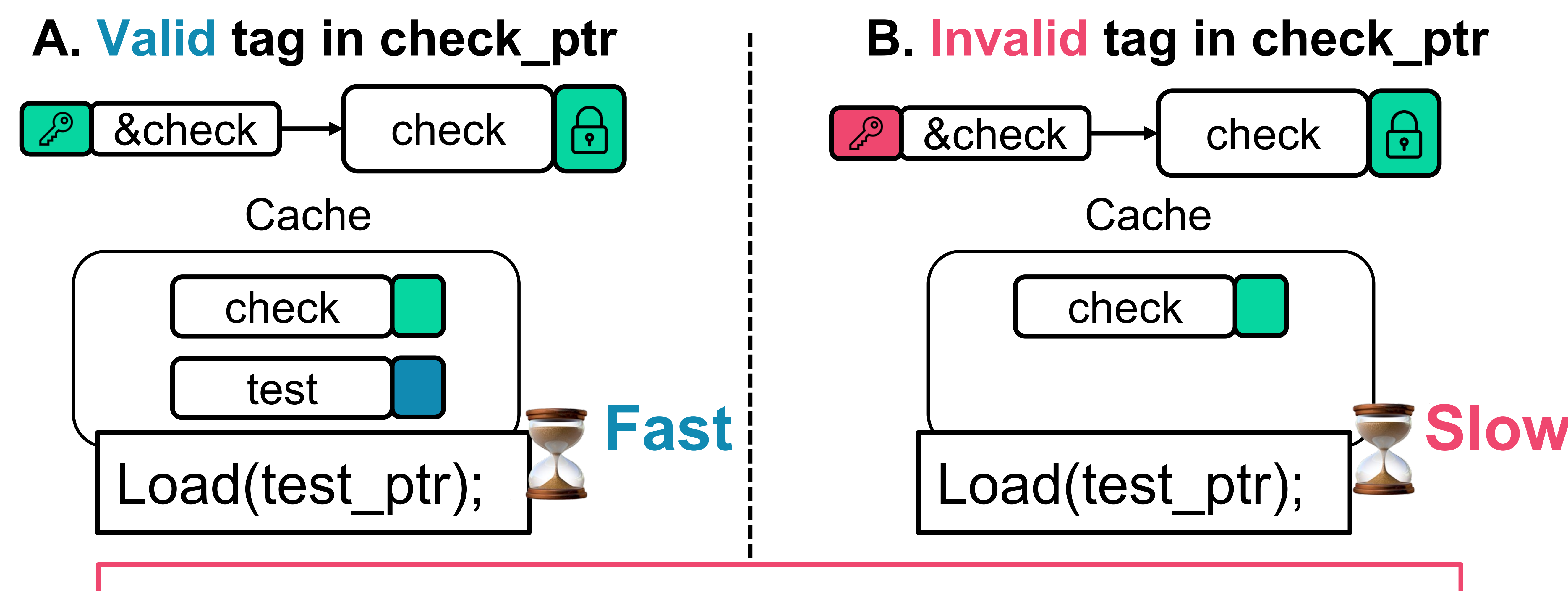
A. Valid tag in check_ptr



B. Invalid tag in check_ptr



Leak by Cache Side-Channel



Leak whether the tag is Valid/Invalid by test_ptr access latency!

Do new MTE chips contain the tag leakage side-channels?

PACMAN – ISCA 2022, DEF CON 30

- Discovered a Pointer Authentication Code (PAC) side-channel

MTE as Tested – Google Project Zero, POC 2023

- Attempted to find a MTE tag side-channel → **Failed**

Our work

- **Found 2 TikTag Gadgets + Root Causes**
- Gadget poc: **https://github.com/compsec-snu/tiktag**
- Paper (IEEE S&P 2025): **https://arxiv.org/abs/2406.08719**

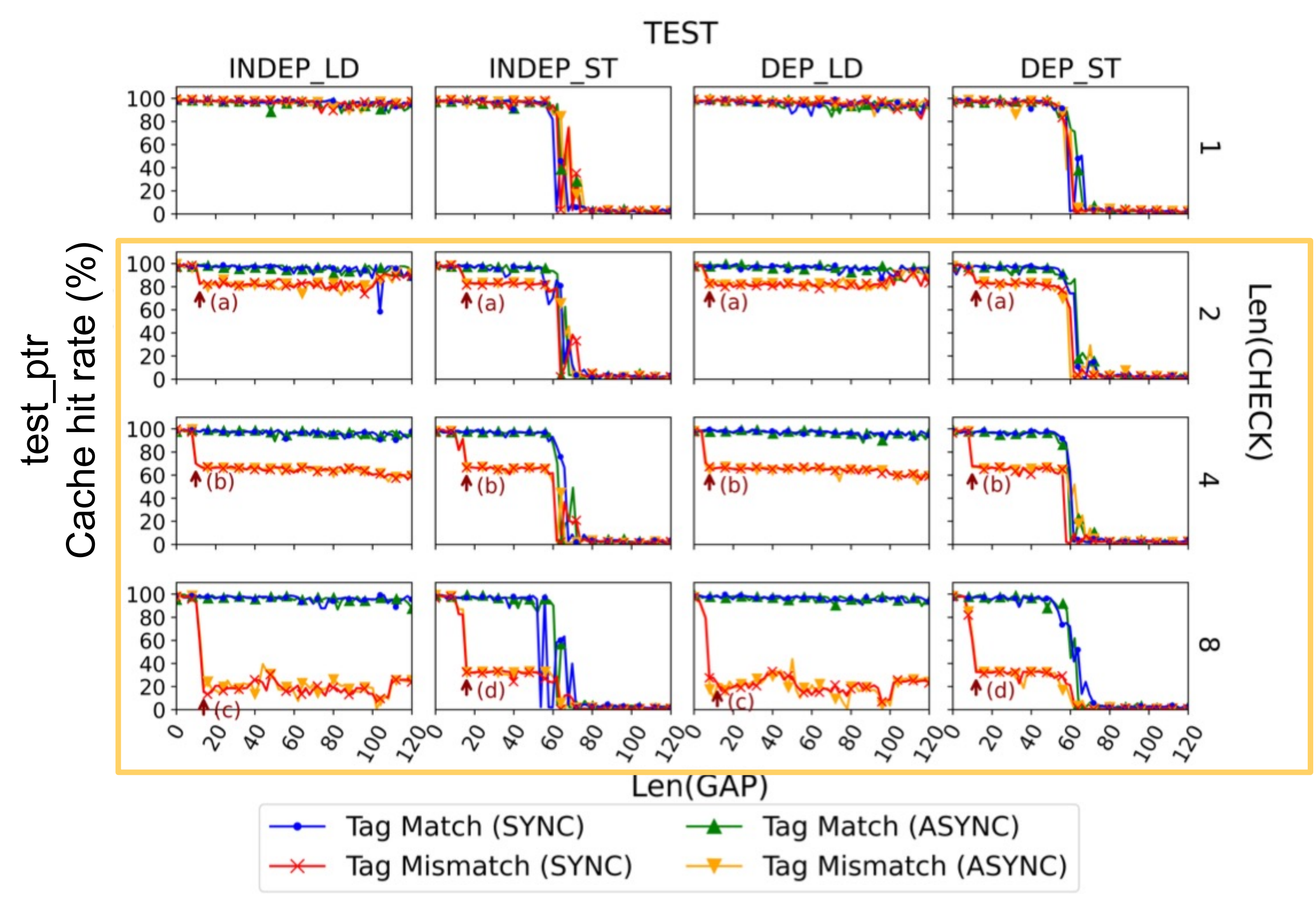
StickyTags – VUSec, IEEE S&P 2024

- Orthogonally found one of our tag leakage gadets

```
if (cond) {
  // Check
  k_ptr;

  Access(test_ptr);
}
```

TikTag-v1: Multiple Loads + Gap

```
if (cond) {
  // CHECK: 2+ loads
  *check_ptr;    Tag Check Fault
  *check_ptr;    Tag Check Fault
  …
  // GAP: 10+ cycles
  test_ptr = test_ptr | 0;
  …
  // TEST: dep/indep ld/st
  *test_ptr;     Not Accessed
}
```



2+ tag check faults + 10+ cycle gap (lf - TEST)
→ test_ptr cache hit rate dropped
= test_ptr is not accessed

TikTag-v1: Multiple Loads + Gap

```
if (cond) {
  // CHECK: 2+ loads
  *check_ptr;        Tag Check Fault
  *check_ptr;        Tag Check Fault
  …
  // GAP: 10+ cycles
  test_ptr = test_ptr | 0;
  …
  // TEST: dep/indep ld/st
  *test_ptr;         Not Accessed
}
```

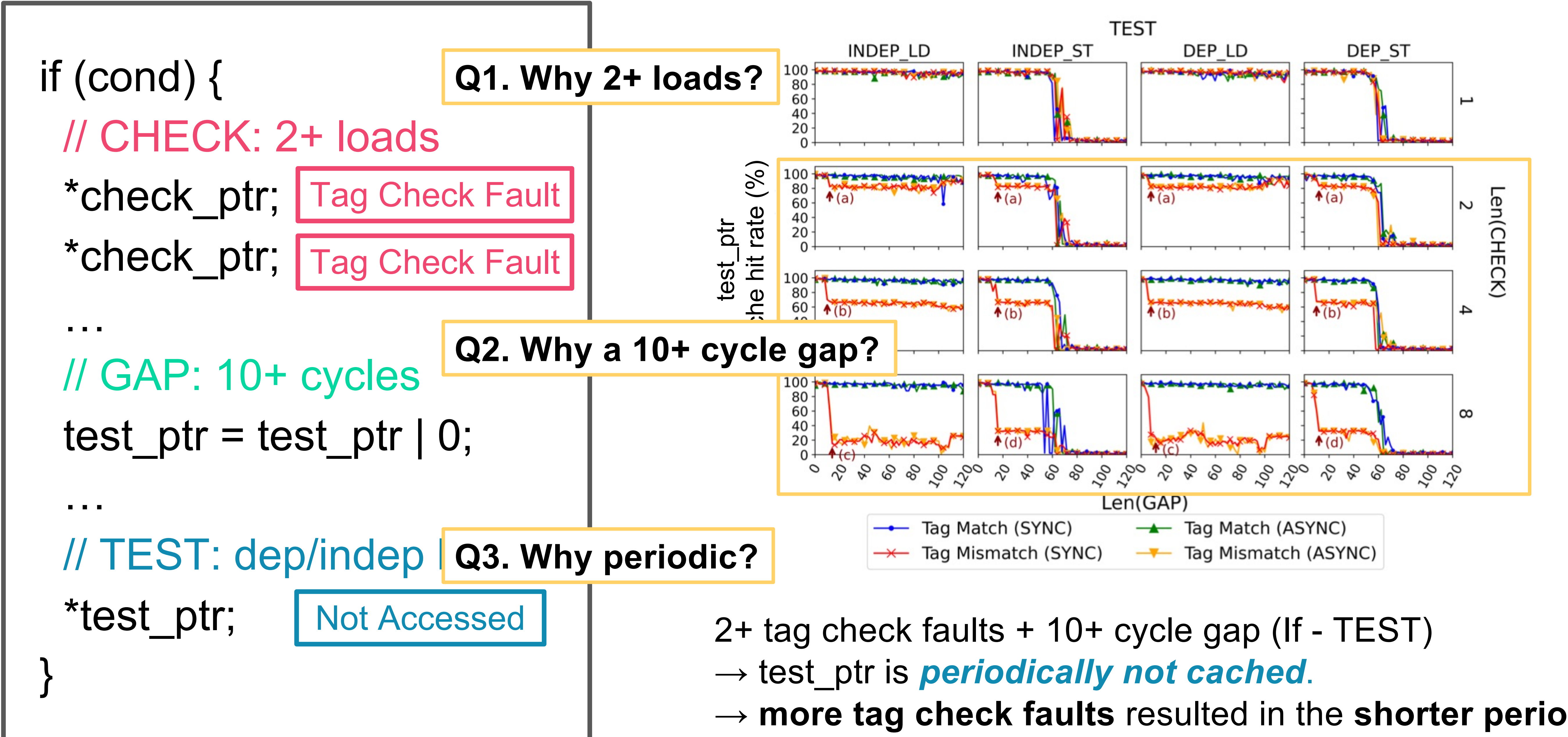
```
test_ptr access latency
Valid tag
38   36   36   36   36   36   36   36   36   362  36
6    38   38   38   38   38   38   38   36   36   36
6    36   36   36   36   36   36   36   36   36   38
1    36   36   36   36   36   36   36   294  36   36
6    36   36   36   36   36   36   36   38   36   36
6    36   36   36   36   36   36   36   36   36   36
6    36   36   36   332  36   38   36   36   36   36
6    36   36   36   36   36   36   36   36   36   36
6    36   36   36   38   36   36   36   36   36   36

Invalid tag
38   36   36   488  36   36   36   487  36   36   36
90   38   36   36   489  38   36   36   488  36   36
6    491  36   36   36   491  36   36   36   487  38
6    36   488  36   36   36   488  36   36   36   489
6    36   36   488  36   36   36   489  38   36   36
85   36   36   36   489  36   36   36   490  36   36
6    489  36   36   36   489  38   36   36   486  36
6    36   487  36   36   36   492  36   36   36   489
6    36   36   489  38   36   36   487  36   380  296
```

2+ tag check faults + 10+ cycle gap (If - TEST)

→ test_ptr is ***periodically not cached***.

→ **more tag check faults** resulted in the **shorter period**

TikTag-v1: Multiple Loads + Gap



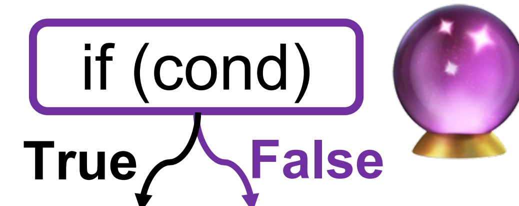
TikTag-v1: Root Cause Analysis

The CPU speculatively accesses memory by:

1) Speculative Execution (SE)

: Speculate branch results based on **branch history**

→ Hypothesis 1: **SE** stops on **tag check faults**



2) Hardware Data Prefeteching (DP)

: Prefetch based on **memory access history**

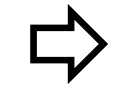
→ Hypothesis 2: **DP** stops on **tag check faults**

Accessed

Prefetch

m1 m2 m3 m1 m2 m3 m1 m2 m3

Challenge: Blackbox CPU



Method 1: Ablation Study

Method 2: Patent Review

TikTag-v1: Ablation Study

An **ablation study** investigates the performance of an AI system by removing certain components to understand the contribution of the component to the overall system. (Wikipedia)

- **Baseline (SP&DP)**
- **-SE (DP only)** : insert speculation barrier
- **-SE & -DP (None)** : insert speculation barrier & change access pattern
- **-DP (SE only)** : change access pattern

TikTag-v1: Ablation Study

```
// BR
if (*m1) {
  // CHECK
  *m2; *m2; …
  …
  // TEST
  *m3
}
```

Test	Branch training (*m1 = True) BR	CHECK	TEST	Speculative execution (*m1 = False) BR	CHECK	TEST	Result	Root cause
Baseline (SE&DP)	m1	m2	m3	m1	**m2**	m3	m3 hit	Not Speculative Execution
				m1	**m2**	m3	m3 miss (periodic)	Not Speculative Execution
-SE (DP)	m1	m2 **SB**	m3	m1	**m2** **SB**	m3	m3 hit	Not Speculative Execution
	(Speculation Barrier)			m1	**m2** **SB**	m3	m3 miss (periodic)	Data Prefetching
-SE&-DP (None)	m1	m2 **SB**	m3	m1	**m2** **SB**	m4	m4 miss (always)	Data Prefetching
				m1	**m2** **SB**	m4	m4 miss (always)	Speculative Execution
-DP (SE)	m1	m2	m3	m1	**m2**	m4	m4 hit	Speculative Execution
				m1	**m2**	m4	m4 miss (periodic)	Speculative Execution

TikTag-v1: Patent Review

(12) **United States Patent**
Cai et al.

(10) **Patent No.:** **US 11,526,356 B2**
(45) **Date of Patent:** **Dec. 13, 2022**

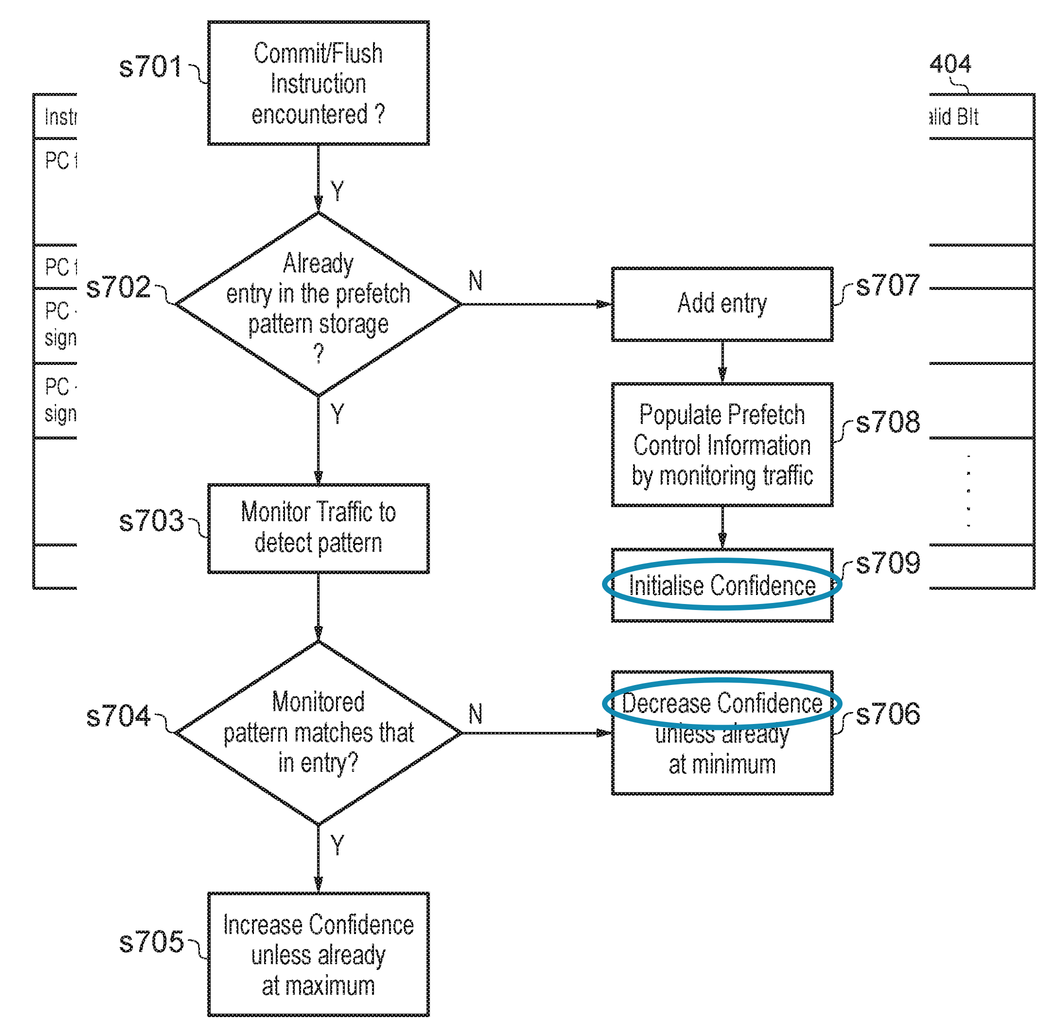
(54) **PREFETCH MECHANISM FOR A CACHE STRUCTURE**

(56) **References Cited**

U.S. PATENT DOCUMENTS

The wrong path event… provides a hint *that the processor pipeline may have fetched one or more* ***instructions that do not require execution****. … some examples are* ***invalid memory accesses****, …*

Improve performance on some cases e.g., Loop-end detection



Q1. Why 2+ loads? *2+ tag check faults treated as a wrong path event*

Q2. Why a 10+ cycle gap? *Remaining instructions in the pipeline & Latency of the wrong path event detection*

Q3. Why periodic? *Confidence decreased on wrong path events*

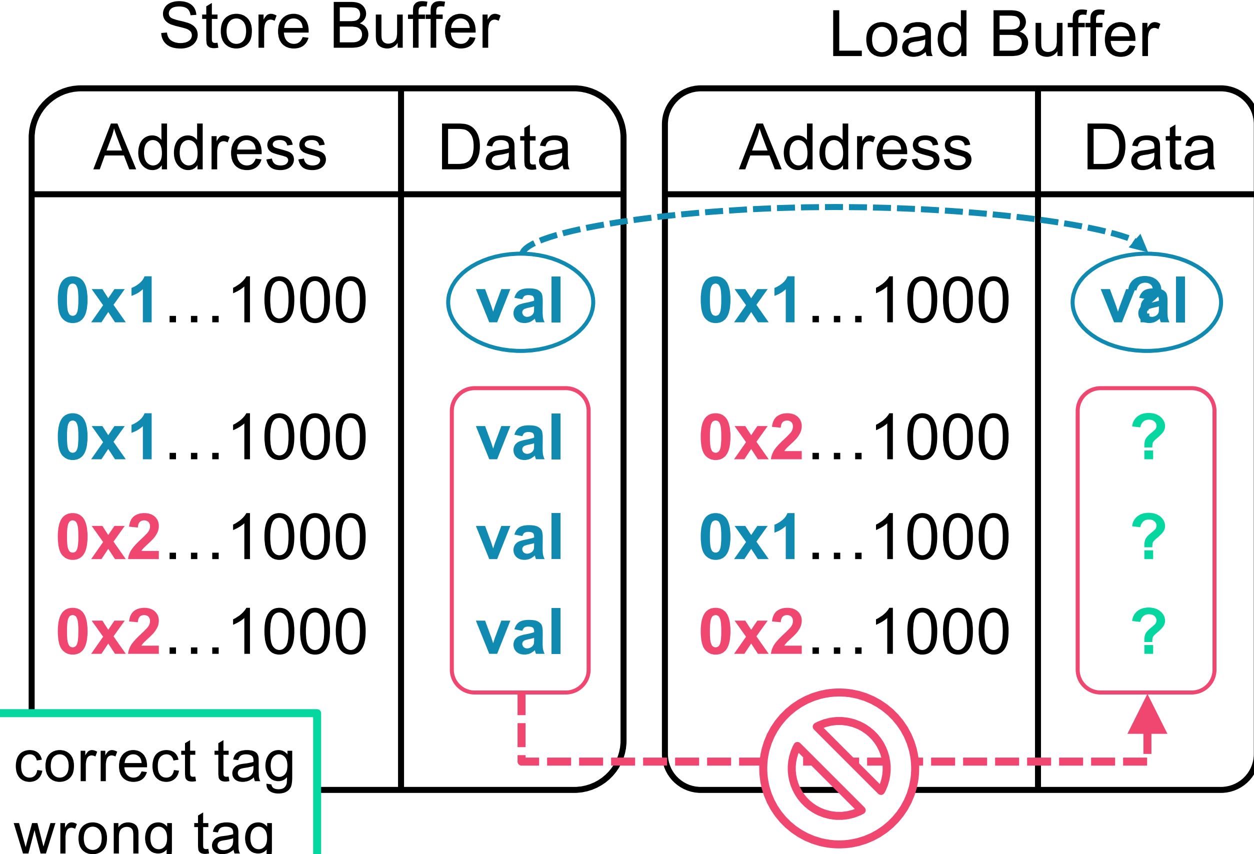
TikTag-v2: Store-to-Load Forwarding

```
if (cond) {
  // CHECK: store-to-load
  *check_ptr = val;        Tag Check Fault
  val = *check_ptr;        Tag Check Fault

  // TEST: dependent load/store
  *(test_ptr+val);         No Access
}
```

within 5 assembly instructions

Suspected root cause

- On **tag check fault**, the CPU blocks ***store-to-load forwarding***

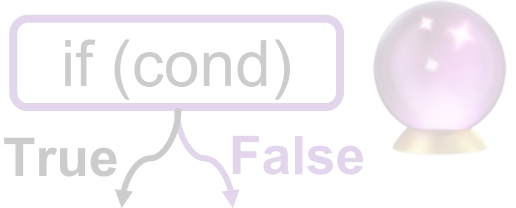


Roadmap

ARM Memory Tagging Extension

Cache Side-Channel

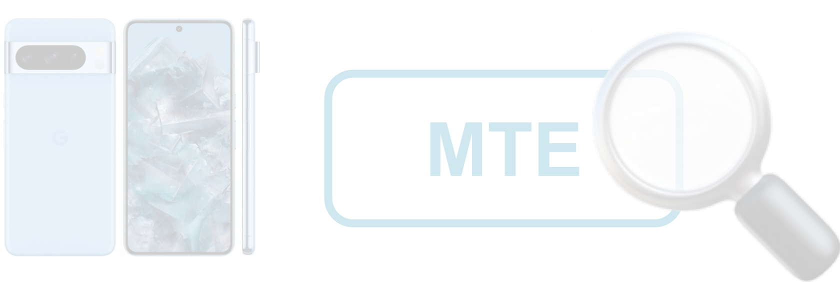
Speculative Execution



Real-world MTE Bypass Attack

JS → MTE

TikTag: MTE Tag Side-Channel



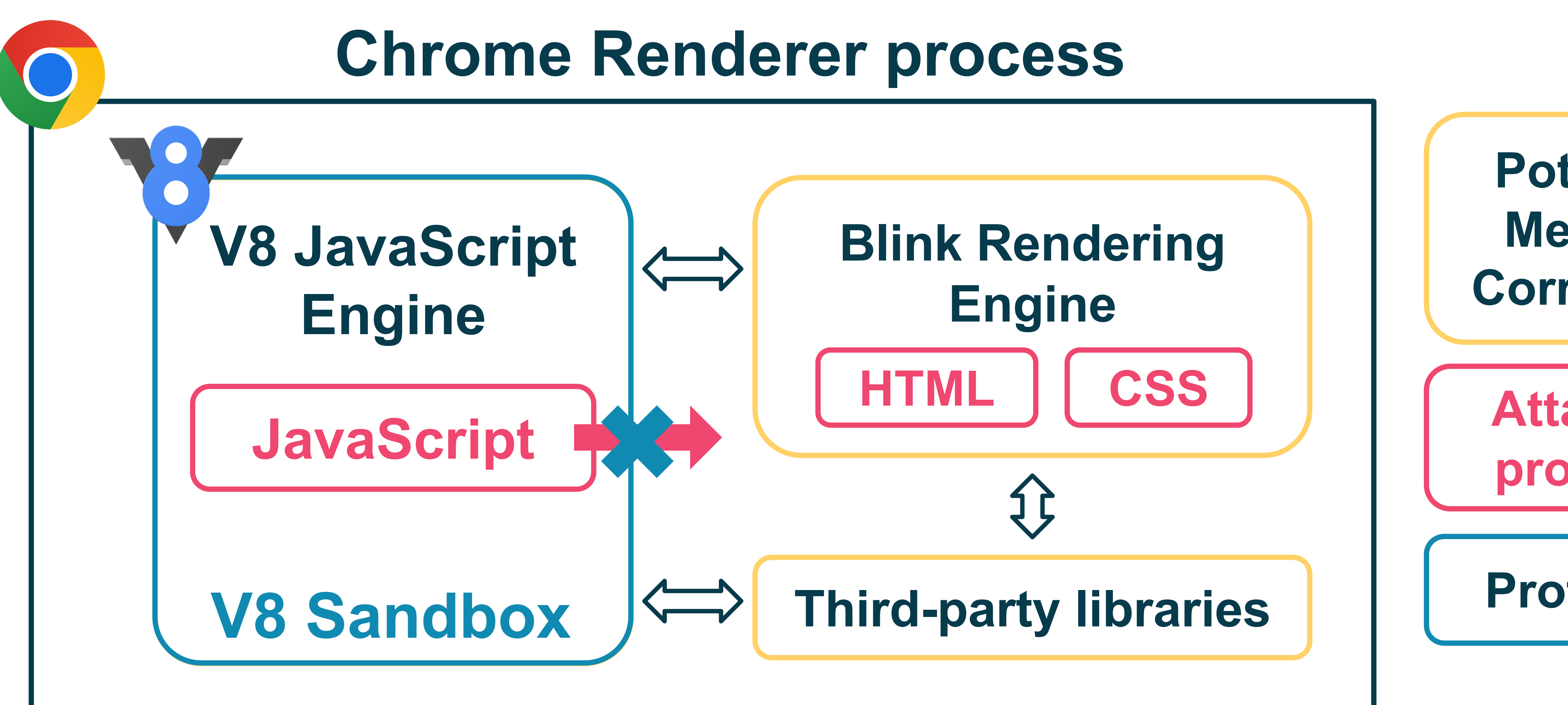
Real-world Gadgets & Attacks
When MTE is enabled

1. Google Chrome V8 Engine
Constructed exploitable Gadget 2 from JavaScript
→ Leak MTE tag of the renderer memory

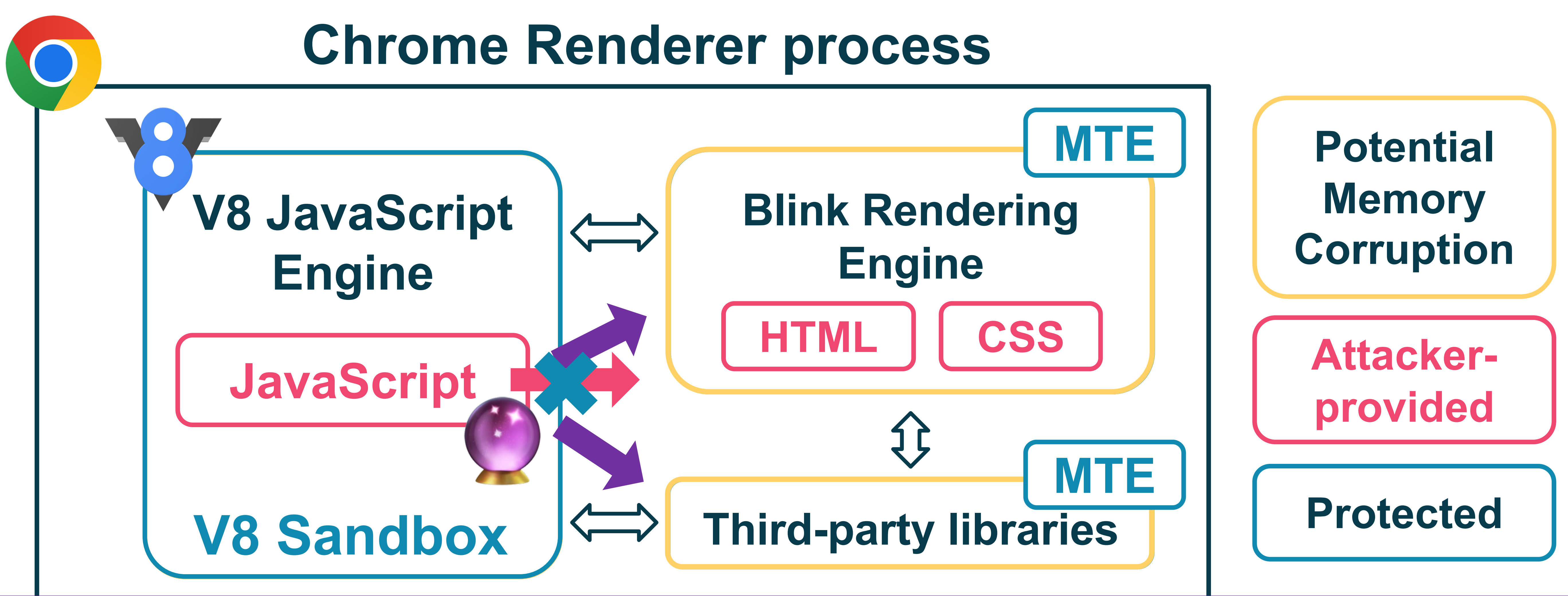
2. Linux kernel
Found potential Gadget 1 in snd_timer()
→ Leak MTE tag of the kernel memory from user space

Refer to our paper for the details: https://arxiv.org/abs/2406.08719

Google Chrome Threat Model



Google Chrome Threat Model



Challenge: Speculatively accessing beyond the V8 Sandbox

Speculatively accessing beyond the V8 Sandbox

JavaScript code

```
arr = new Uint8Array(64);
…
val = arr[idx];
```

V8 compiled code

37-bit (128GB Sandbox)

```
…
elements = arr.Pointer + PtrComprBase;
if (idx < arr.Length)     val = elements[idx];
else                      val = undefined;
```

64-bit out-of-bounds   35-bit   access arbitrary 64-bit addr

→ **V8 does not protect the speculative paths**

→ Allows constructing **TikTag-v2 gadget** (+ Spectre-v1)

TikTag-v2 in JavaScript

```
if (cond) {
  check[idx] = val;
  val = check[idx];
  x = test[val];
}
```

Speculative Execution

idx: out-of-bounds index (64-bit)

check[idx]: check_ptr

test[val]: test_ptr

Exploiting TikTag-v2 in V8

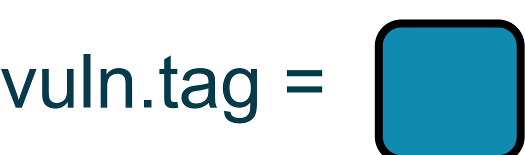
```
TagLeak(target) {
  for (let tag=0; tag < 16; ++tag) {     ← Iterate all tag values
    idx = AddrToIdx(tag, target);        ← out-of-bounds index
    if (cond) {
      check[ idx ] = val;
      val = check[ idx ];
      x = test[val];
    }
    time[tag] = Measure( test[val] );
  }
  return time.indexOf(min(time));
}
```

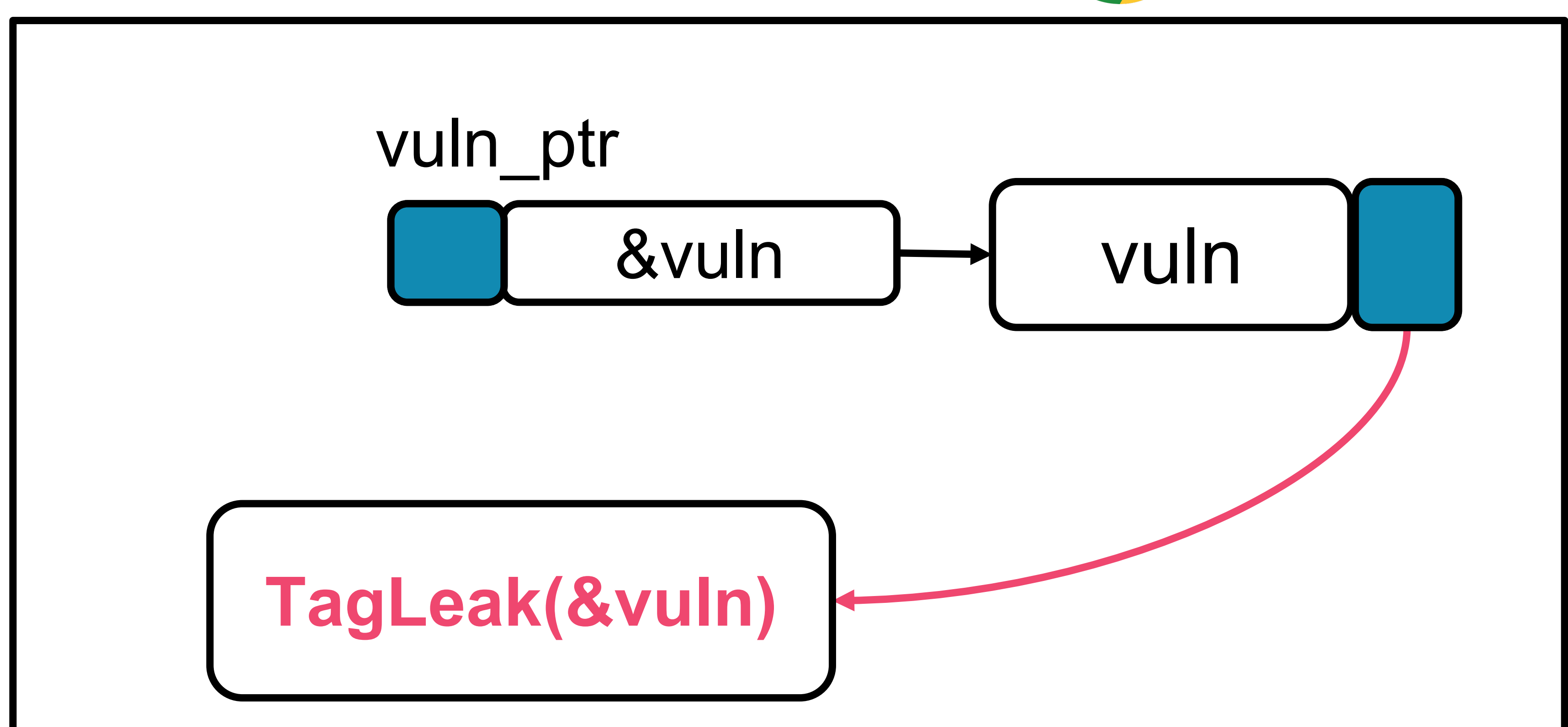
Tag Leakage Gadget

check[idx]	Valid tag	Invalid tag
	No fault	Tag Check Fault
	No fault	Tag Check Fault
	Access	No Access
	Fast	**Slow**

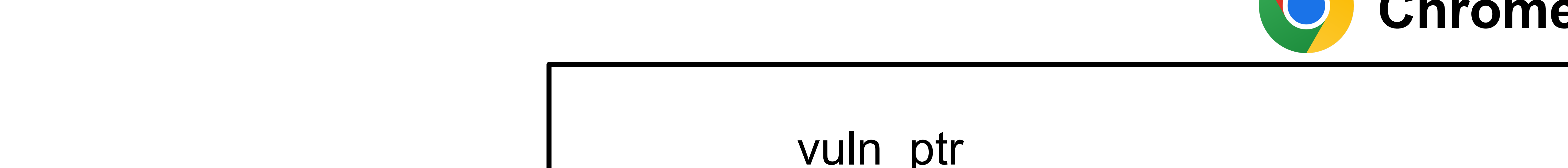
Tag Leaked!

1. Leak MTE Tag of vulnerable object



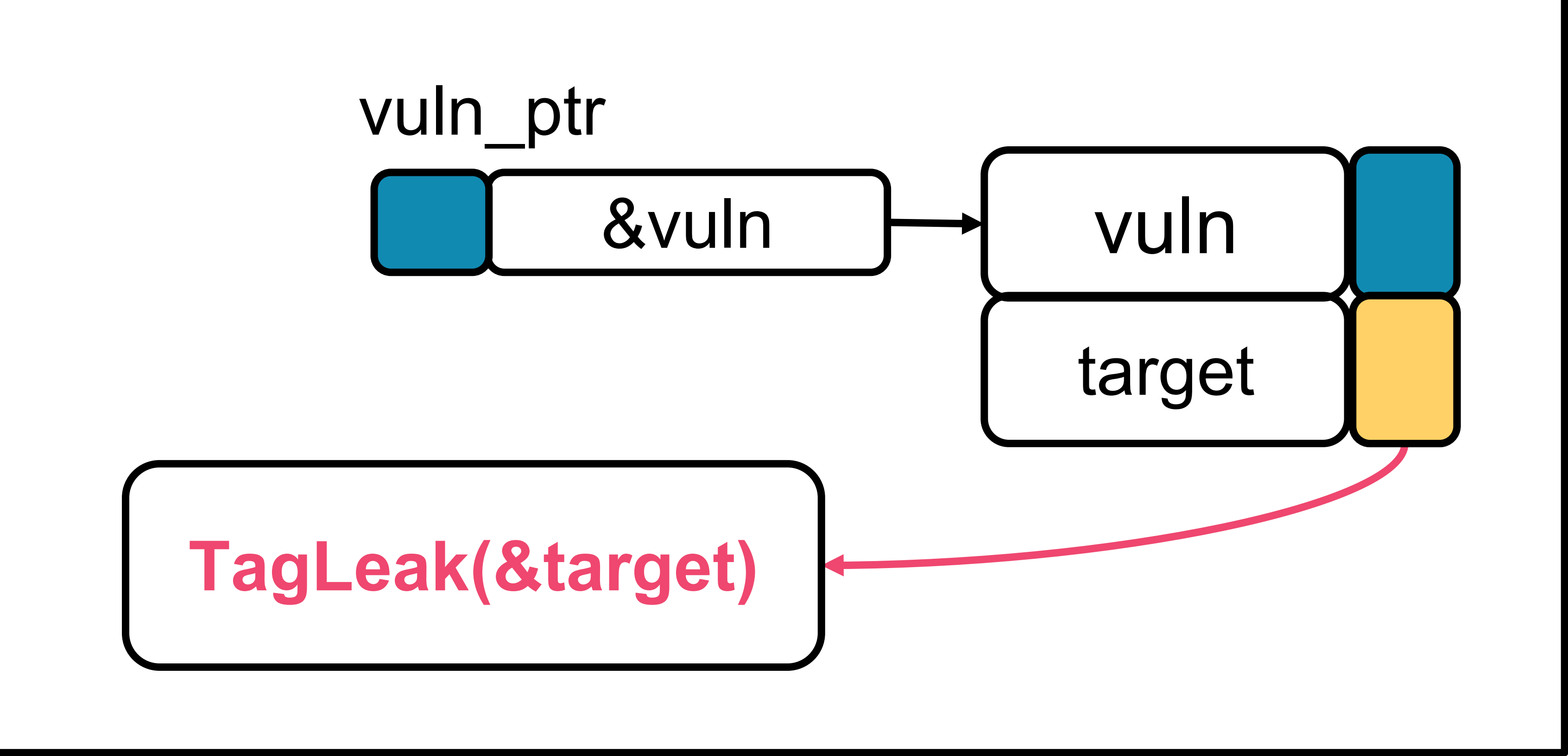


2. Leak MTE Tag of target object

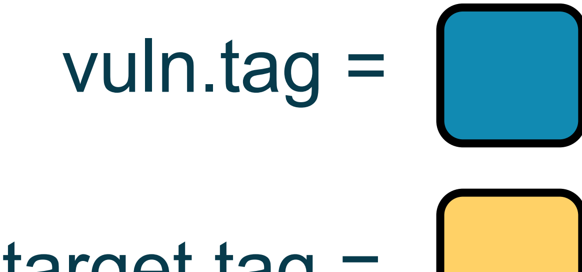


vuln.tag =

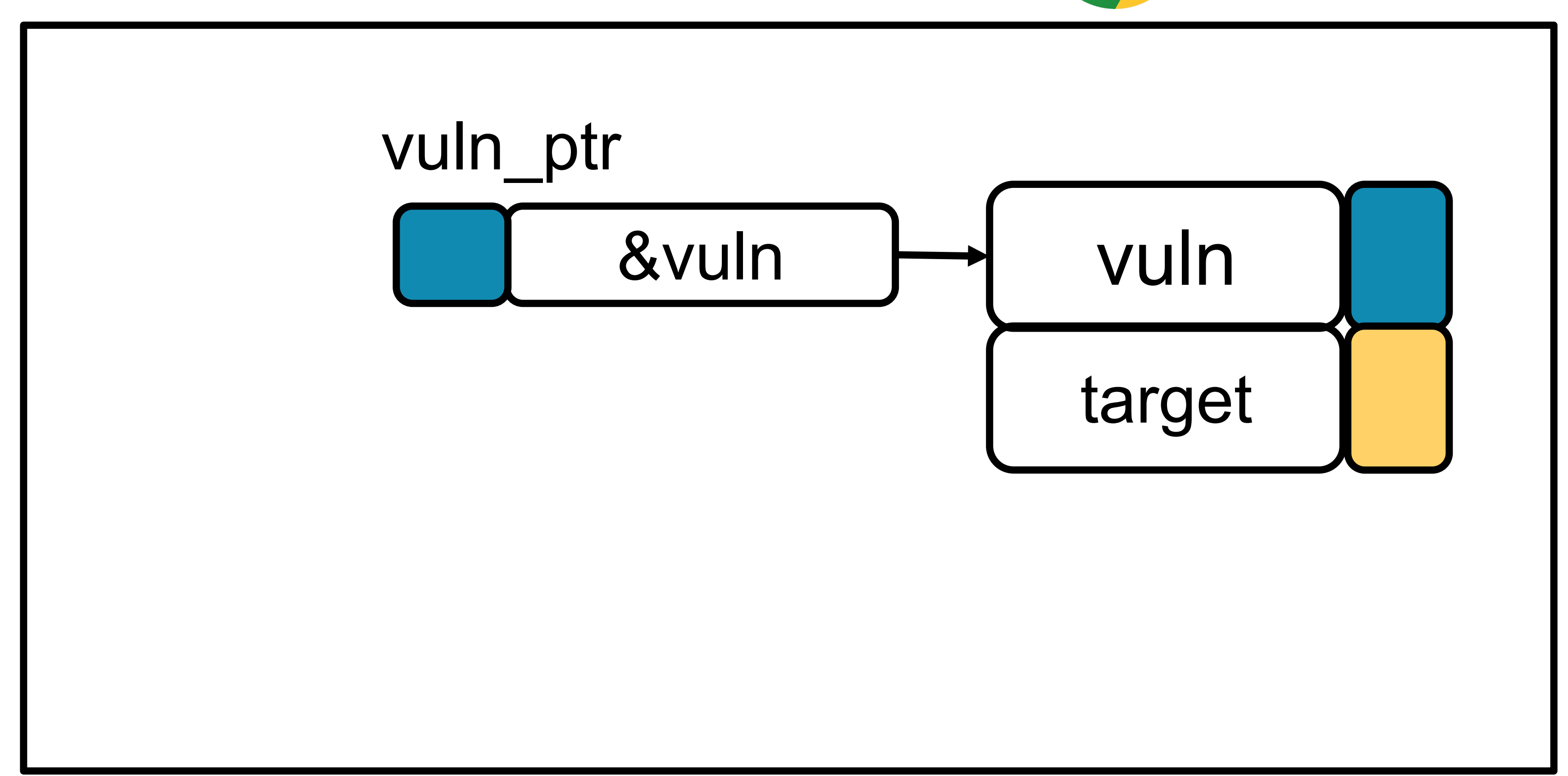
target.tag =



3. Reallocate target on tag mismatch



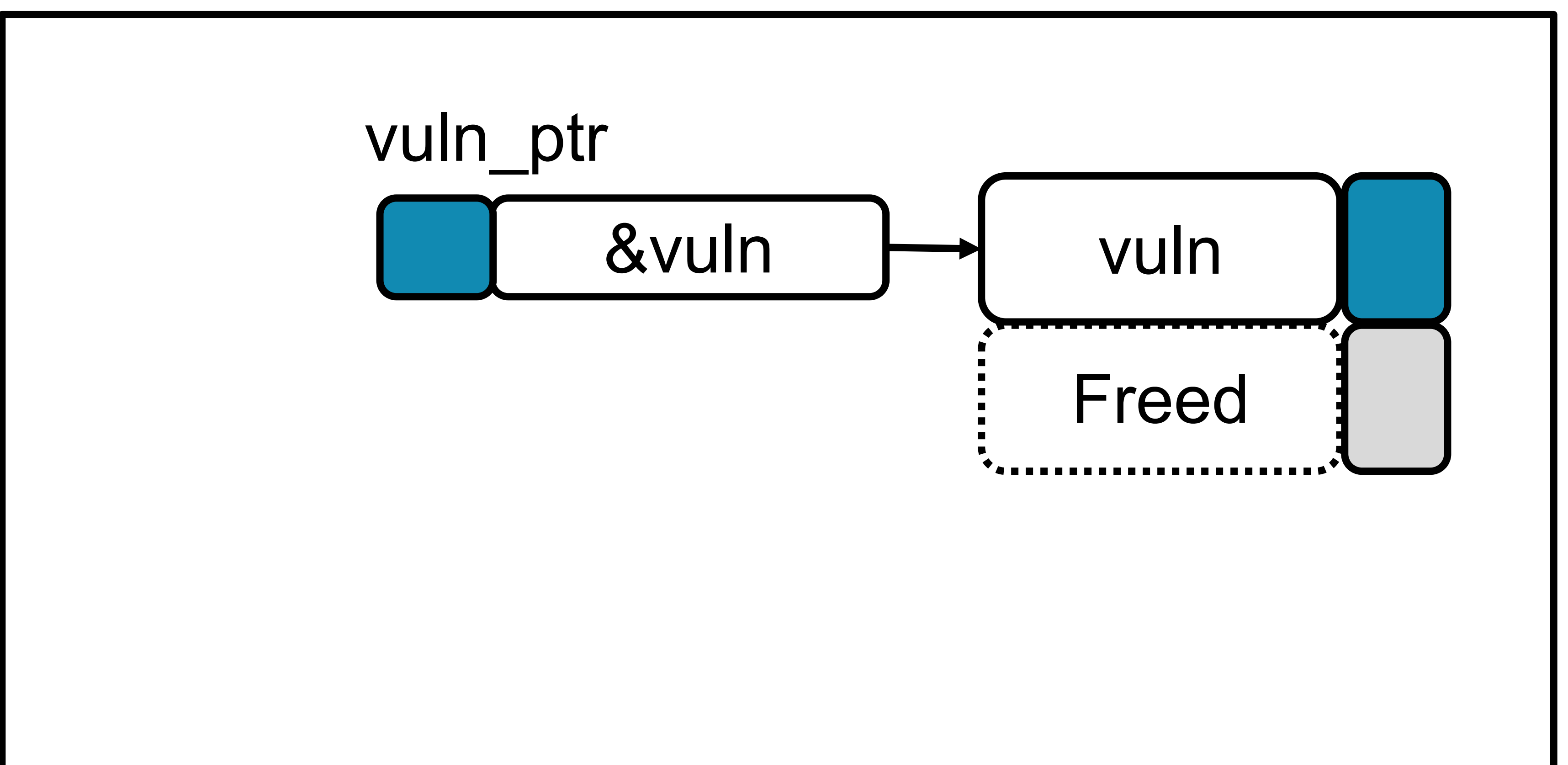
vuln.tag != target.tag



3. Reallocate target on tag mismatch

vuln.tag =

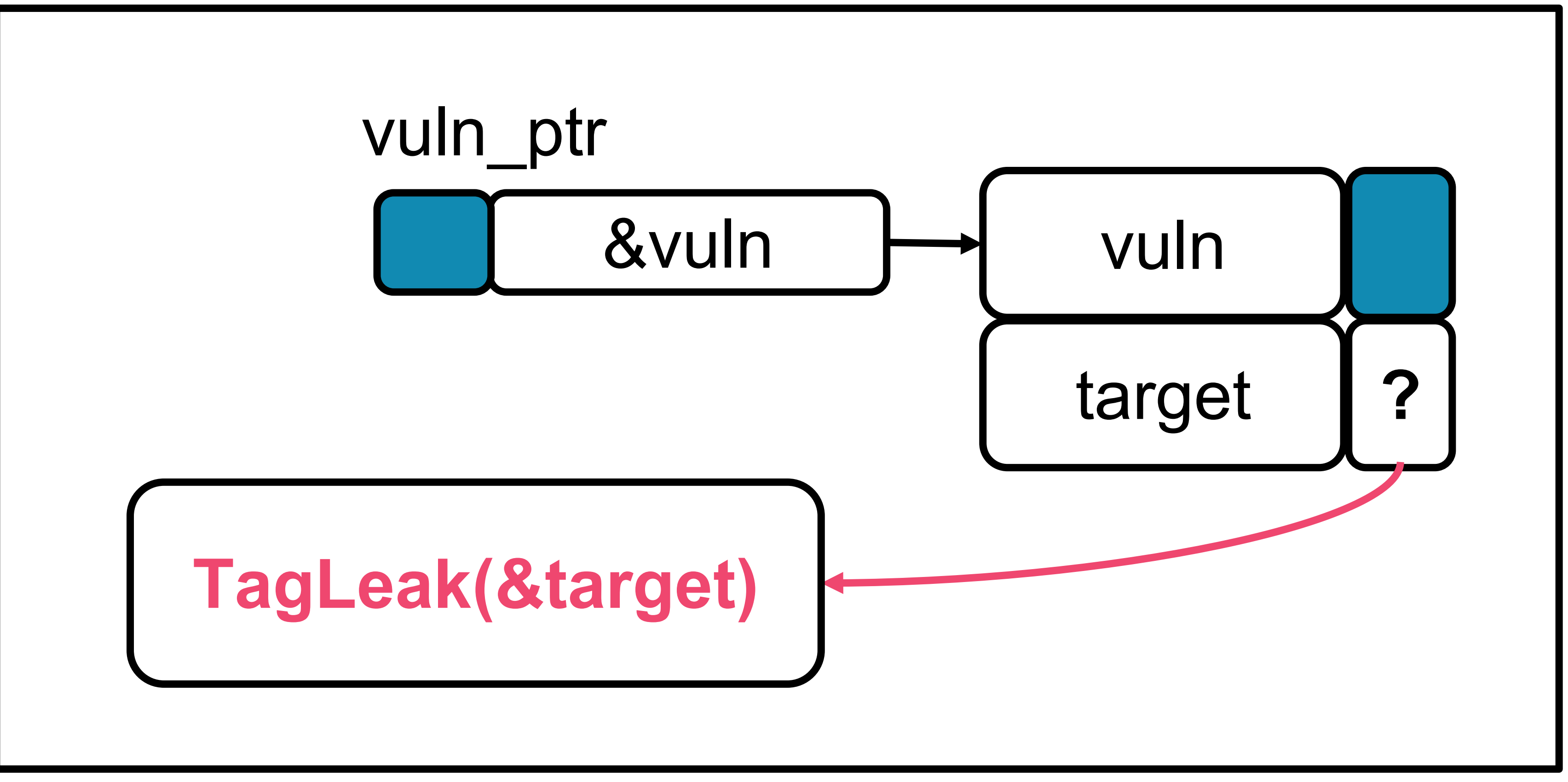
Free(target);



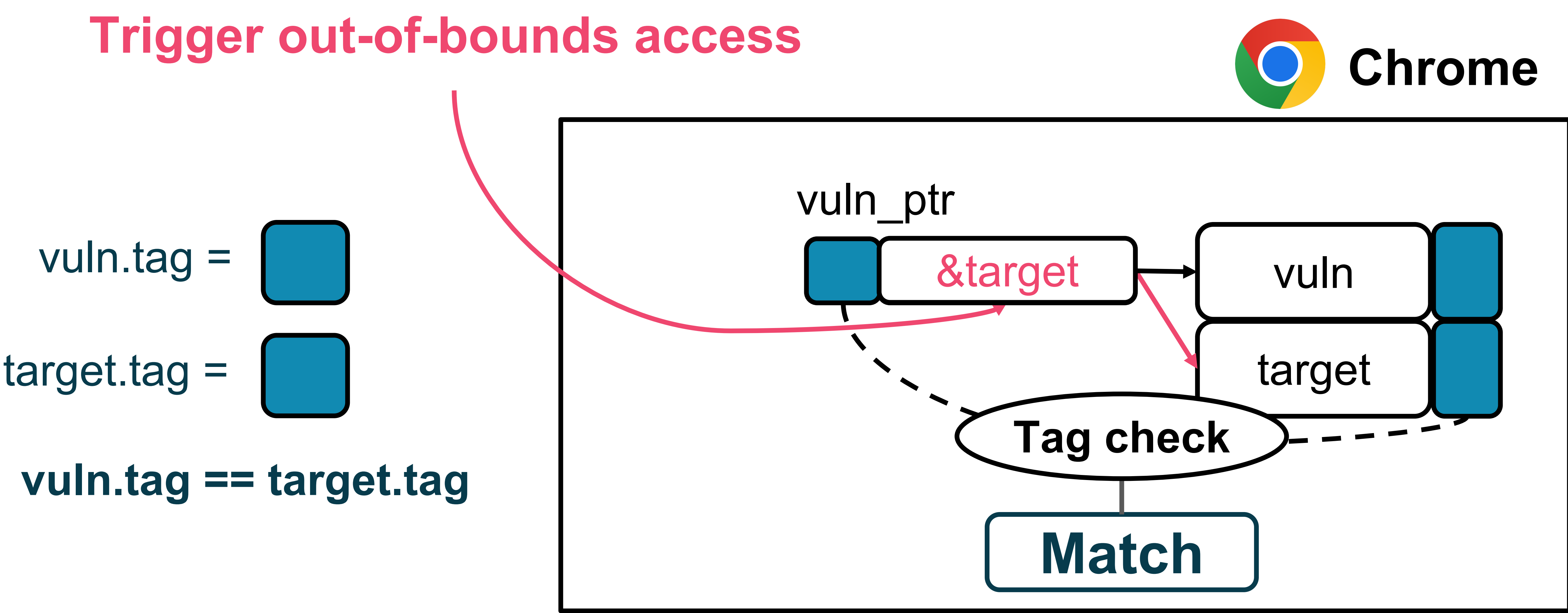
3. Reallocate target on tag mismatch

vuln.tag =

Free(target);
Alloc(target);

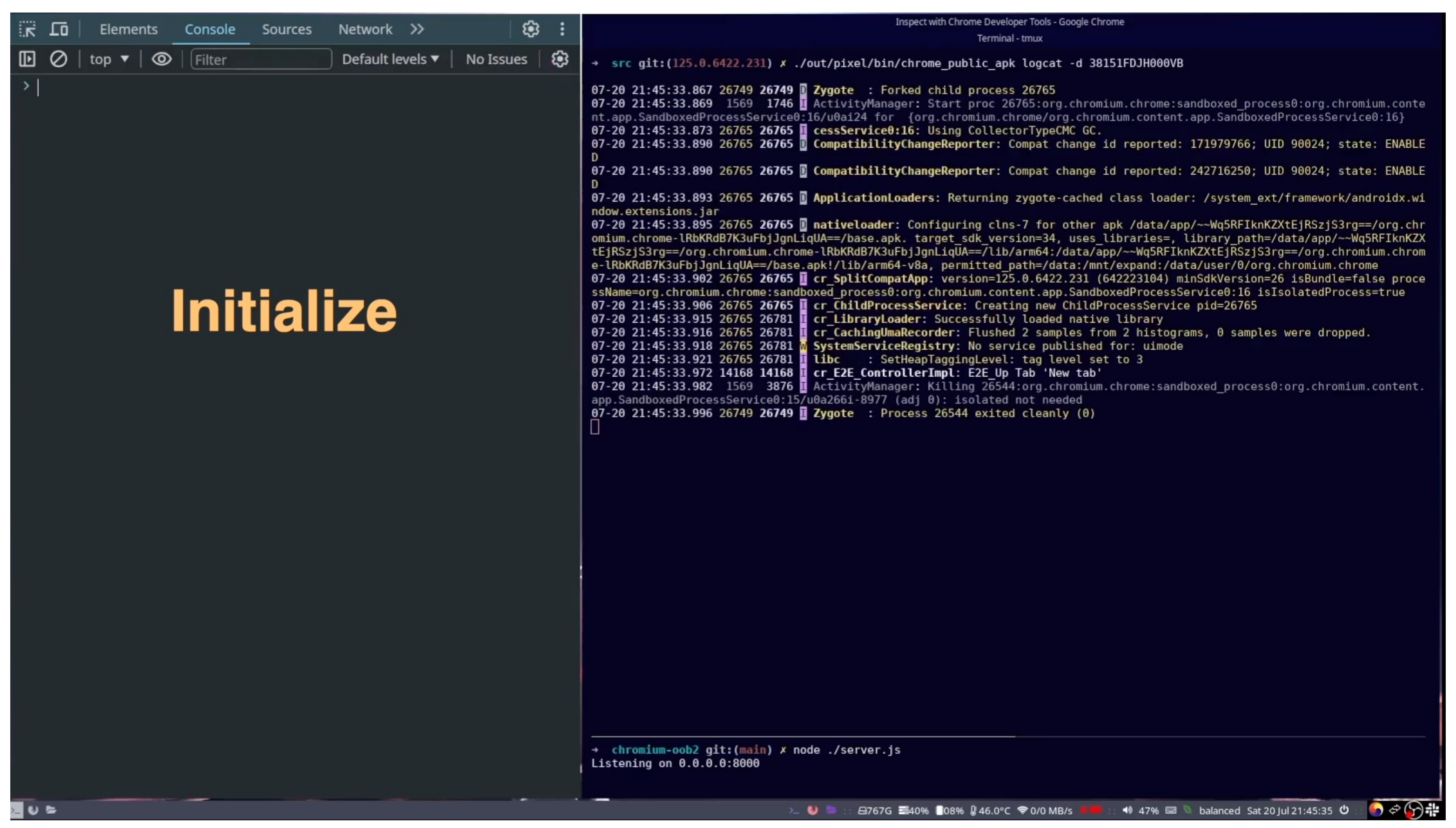


4. Trigger vulnerability on tag match



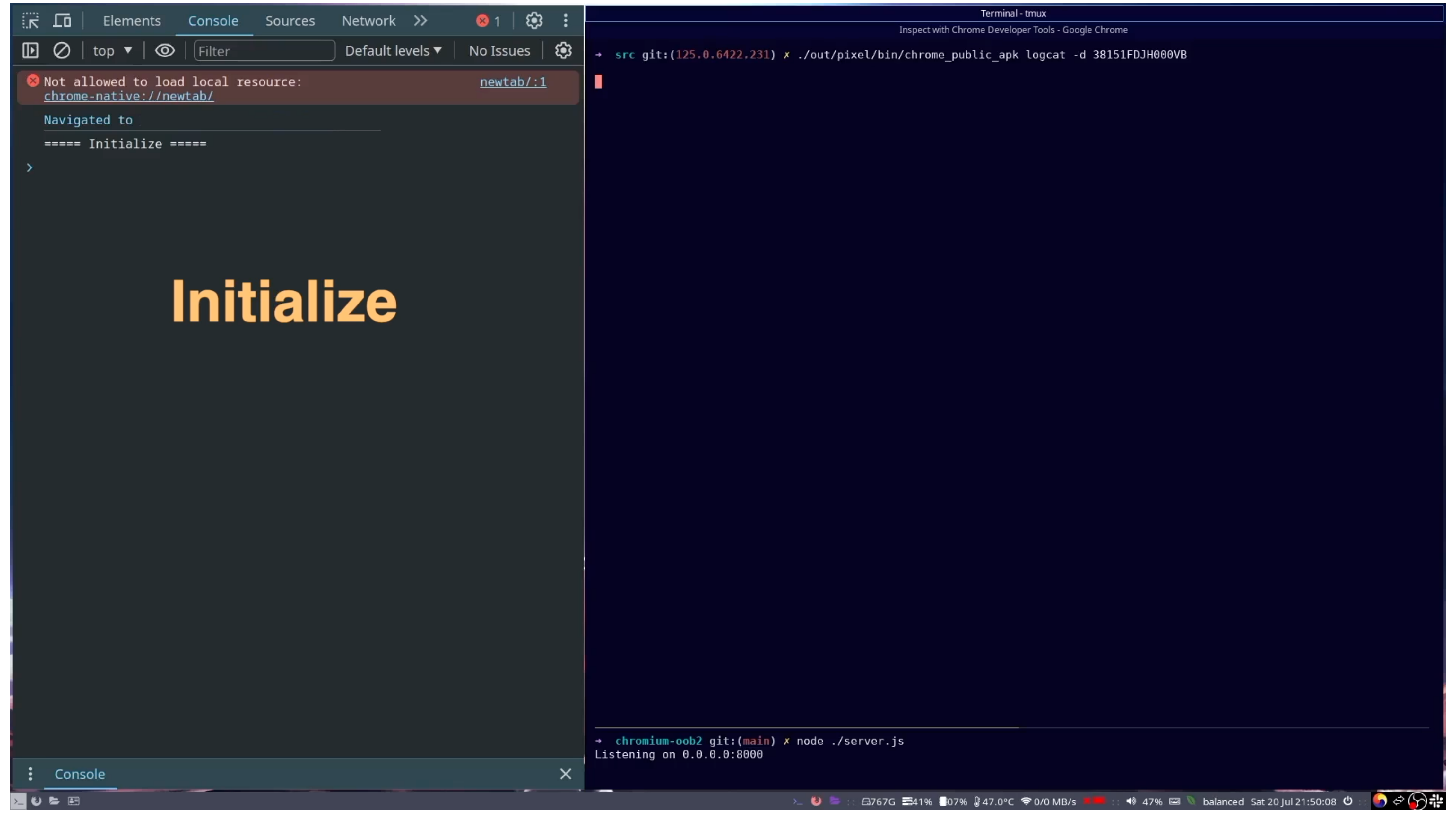
CVE-2023-5217 Chrome libvpx heap overlfow

Original Memory Corruption → Attack Fail



CVE-2023-5217 Chrome libvpx heap overlfow

With MTE Tag Leakage → Attack Success



Vendor Responses

ARM

- Acknowledged the MTE tag side-channel in multiple ARM cores
- Cannot leave comments on the root causes
- *MTE Tags are not a secret*
  → *Tag leakage is not a security vulnerability*
- Expected the cost of the hardware fix to be low and recommended the fix.

ARM MTE Security Updates:
https://developer.arm.com/Arm%20Security%20Center/Arm%20Memory%20Tagging%20Extension

Vendor Responses

Google Android Security Team

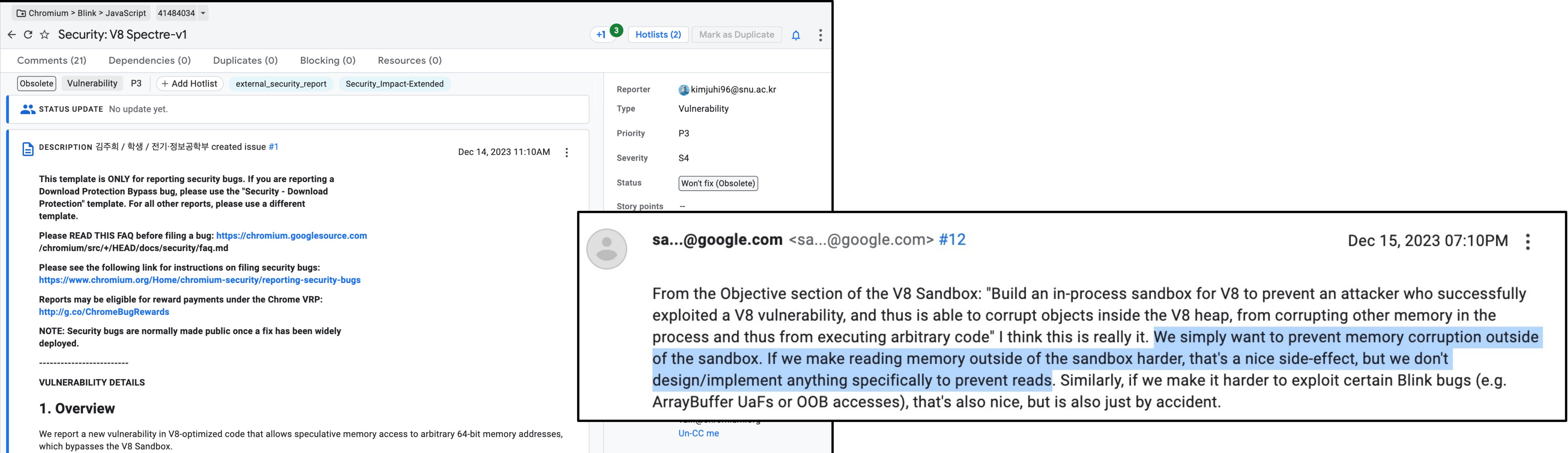
- MTE tag leakage are **hardware flaw** of Pixel 8 & Pixel 8 pro

- **Still, MTE is a strong mitigation against limited-shot exploits:**
  - Minimal attack surface (e.g., Messaging app)
  - Physically remote attack (e.g., Bluetooth, NFC, Wi-Fi, …)
  - Process isolation, IPC attack (e.g., Android, Chrome browser)

Vendor Responses

Google Chrome V8 Team

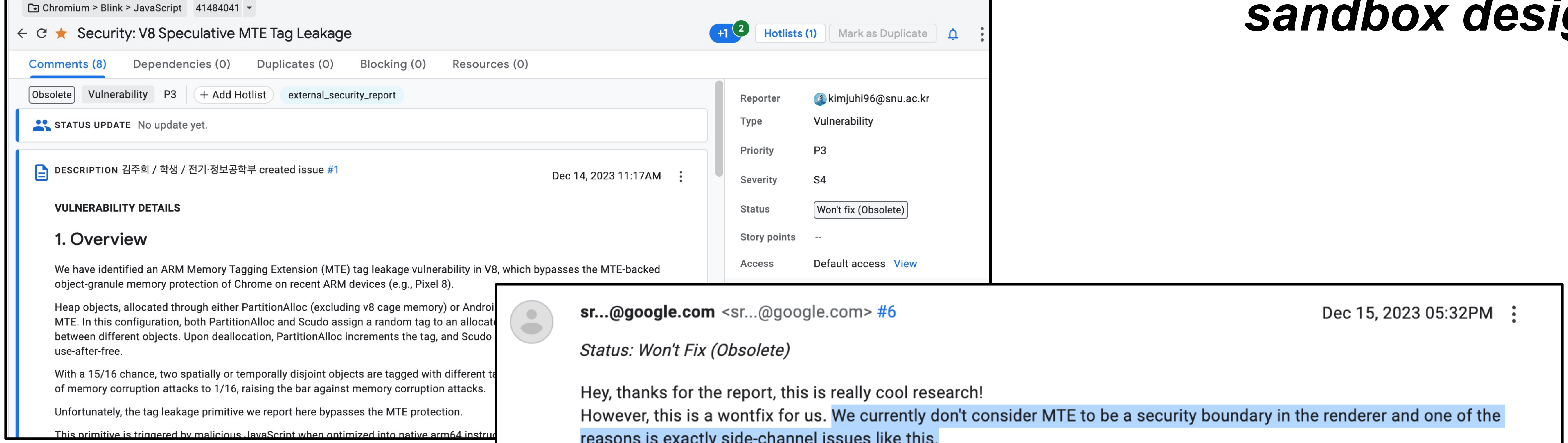
- **Data confidentiality** (including MTE tag's confidentiality) is out of scope of the V8 Sandbox



Vendor Responses

Google Chrome V8 Security Team

- Currently doesn’t plan to adopt MTE on renderer due to **potential side-channel issues** → ***Can be mitigated with a speculation-aware sandbox design***



Hardware Mitigations

```
if (cond) { ⌛
  // Check
  Access(check_ptr);
  // Test
  Access(test_ptr);
}
```

1. Always Block (Bad)

Access(check_ptr);	No Fault	Tag Check Fault
Access(test_ptr);	No Access	No Access

→ **Performance sacrifice**

No Speculative Execution & No Data Prefetching

Hardware Mitigations

```
if (cond) {

  // Check
  Access(check_ptr);

  // Test
  Access(test_ptr);

}
```

2. Always Allos (Better)

No Fault	Tag Check Fault
Access	Access

→ **MTE-Meltdown**

Allows leaking data protected by MTE

Software Mitigations

1. Eliminating Gadgets

- Gadget scanning
- Inserting speculation barrier or dummy instructions to invalidate the gadget

2. Speculative path-aware Sandbox

- V8 sandbox: Complete pointer (and index) compression

The End of Memory Corruption?

- Current MTE **tag leakage side-channels** allow the attacker to bypass MTE random tagging

- **Next steps** to end memory corruption:

- Deterministic tagging (No random tags! - StickyTags),
- Random tagging
  Fix TikTag and prevent brute-force attacks (No second chance!)
- …

Questions?

kimjuhi96@snu.ac.kr