

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



**Georgia Institute
of Technology**

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

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 bounty
- CTF player



Samsung Research



Georgia Institute of Technology

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

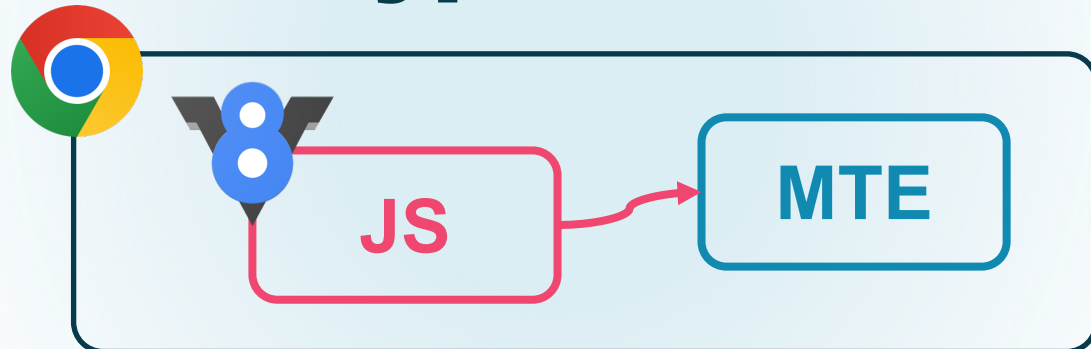
Roadmap

ARM Memory Tagging Extension

arm



Real-world MTE Bypass Attack



Cache Side-Channel

Cache



Speculative Execution

if (cond)

True False



TikTag: MTE Tag Side-Channel



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)

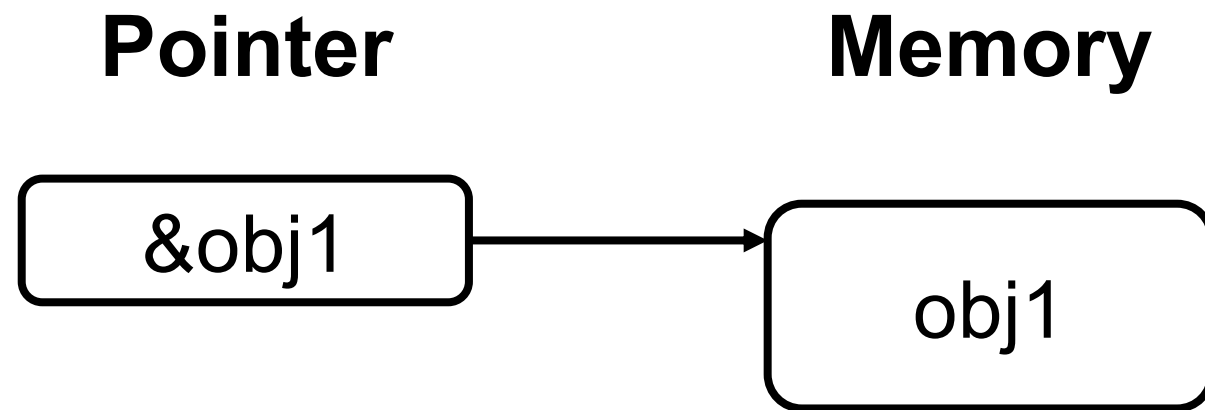
regreSSHion: Remote Unauthenticated Code Execution Vulnerability in OpenSSH server

BLASTPASS (2023)

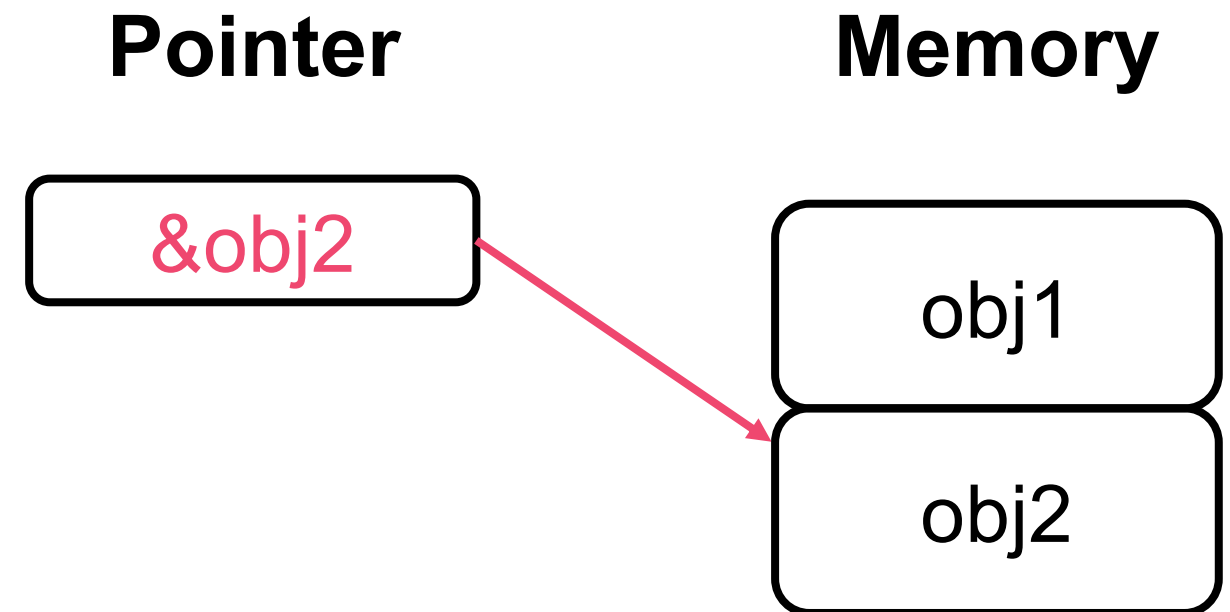
NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild

What is Memory Corruption?

Valid Access



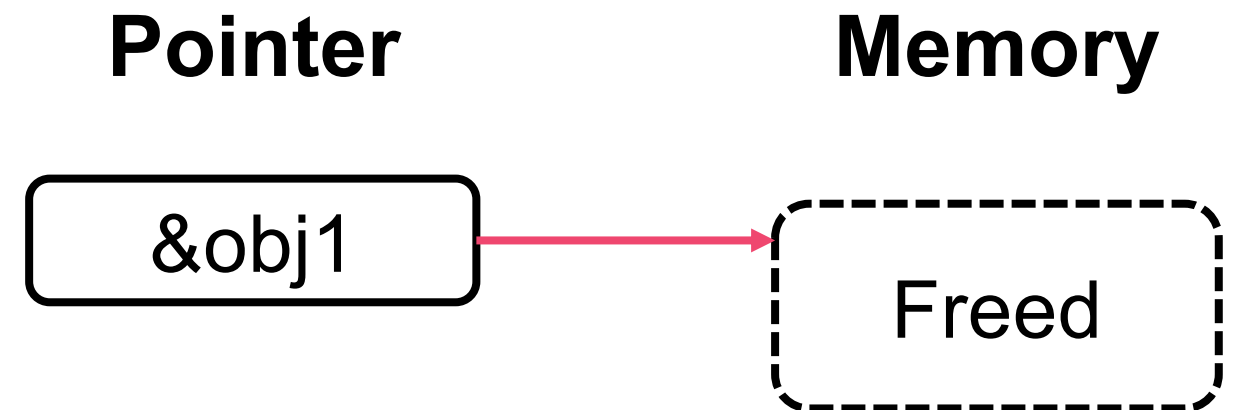
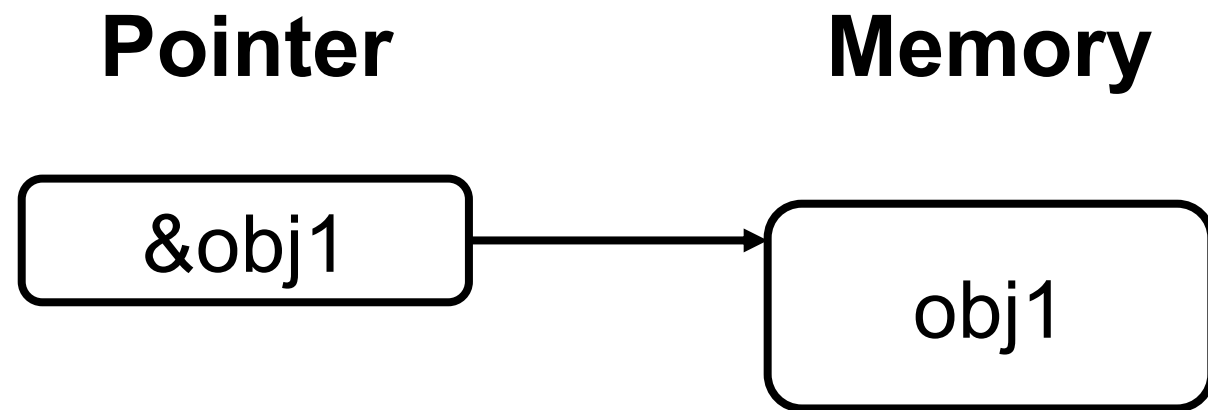
Invalid Access
(Out-of-bounds)



What is Memory Corruption?

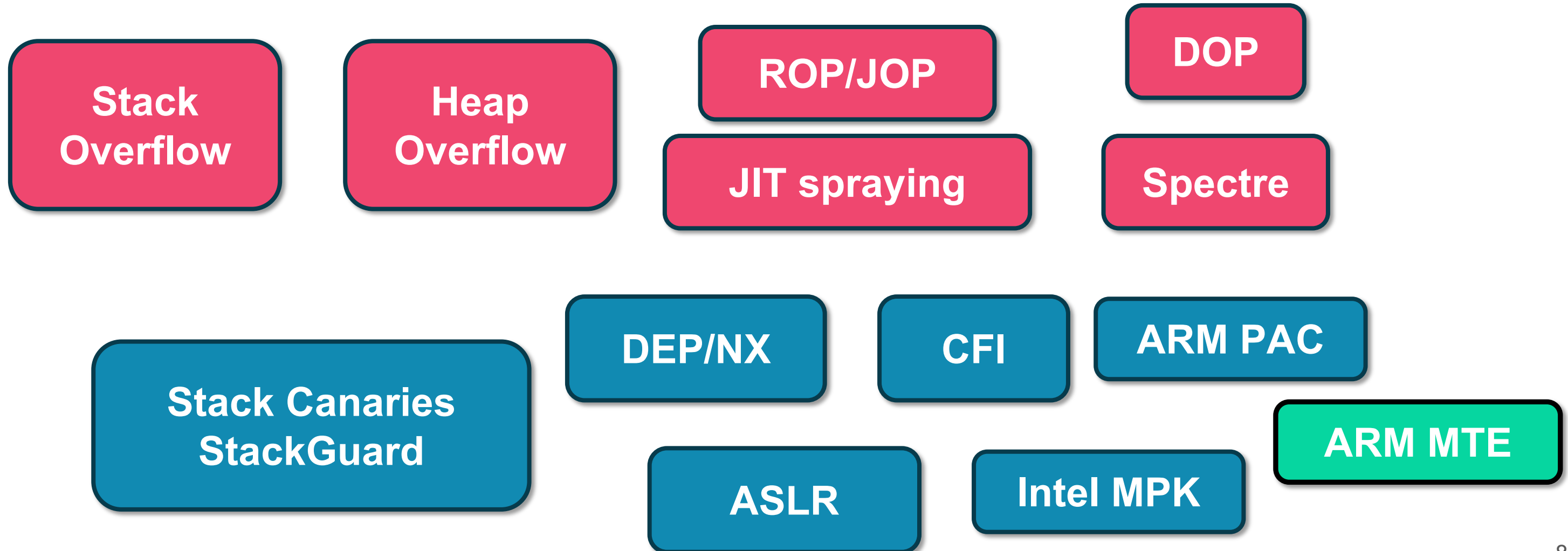
Valid Access

**Invalid Access
(Use-after-free)**



Attack and Defense Techniques

70s-80s — 90s — 2000s — 2010s — 2020s →



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)

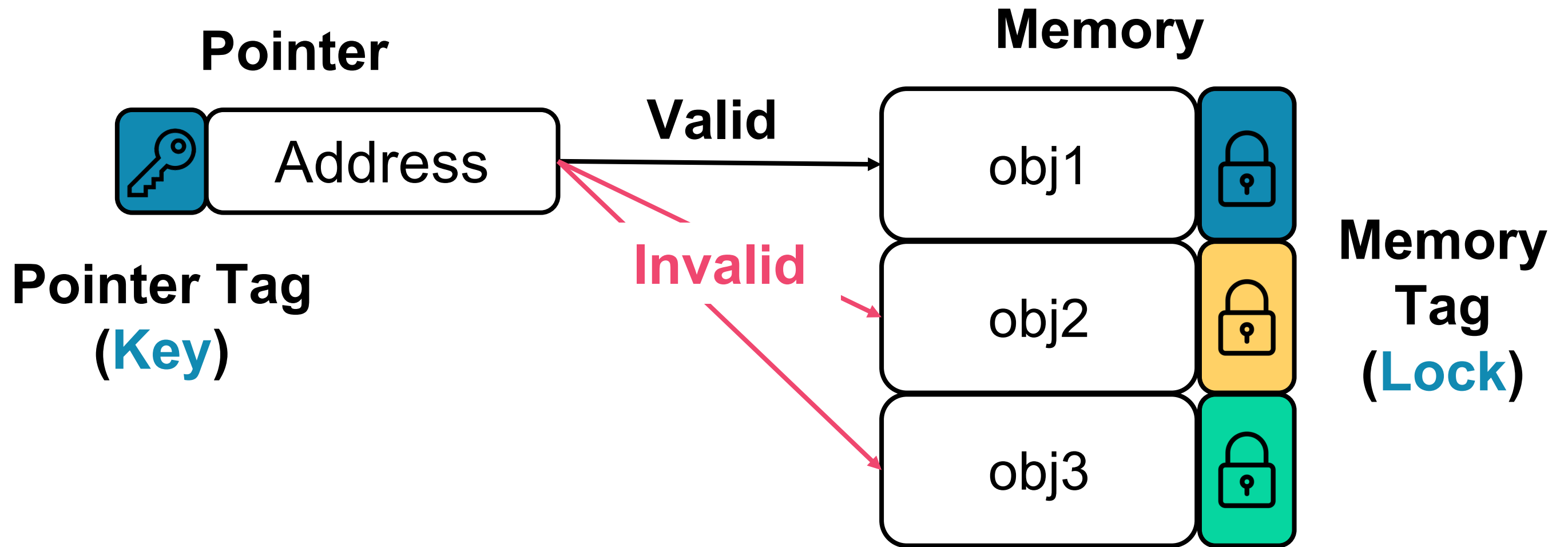
Why is MTE so Special?

Hardware-based

Memory Corruption Detection

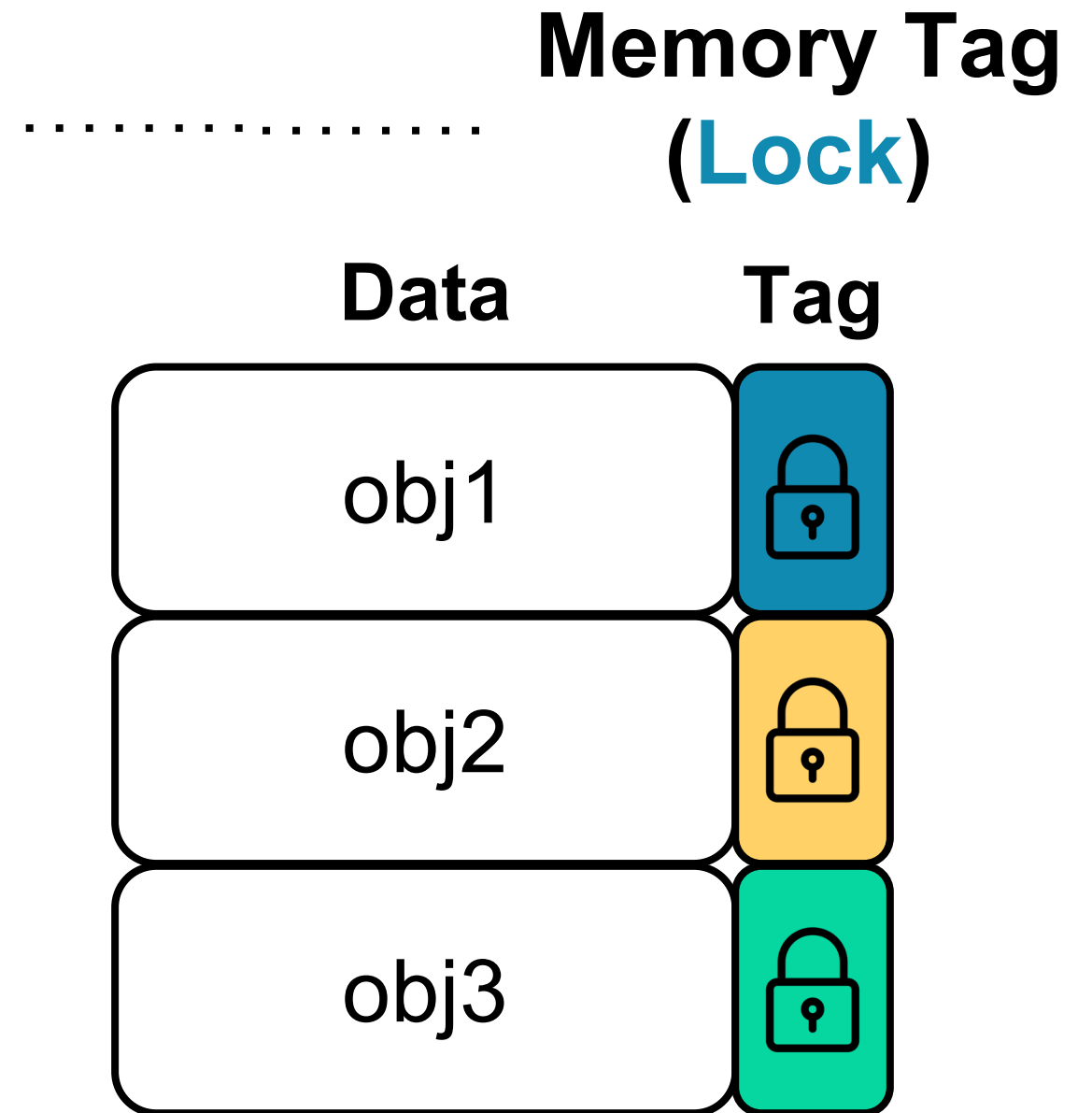
Fast and Compatible

ARM Memory Tagging Extensions



(1) Memory Tag

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



(2) Pointer Tag

Pointer



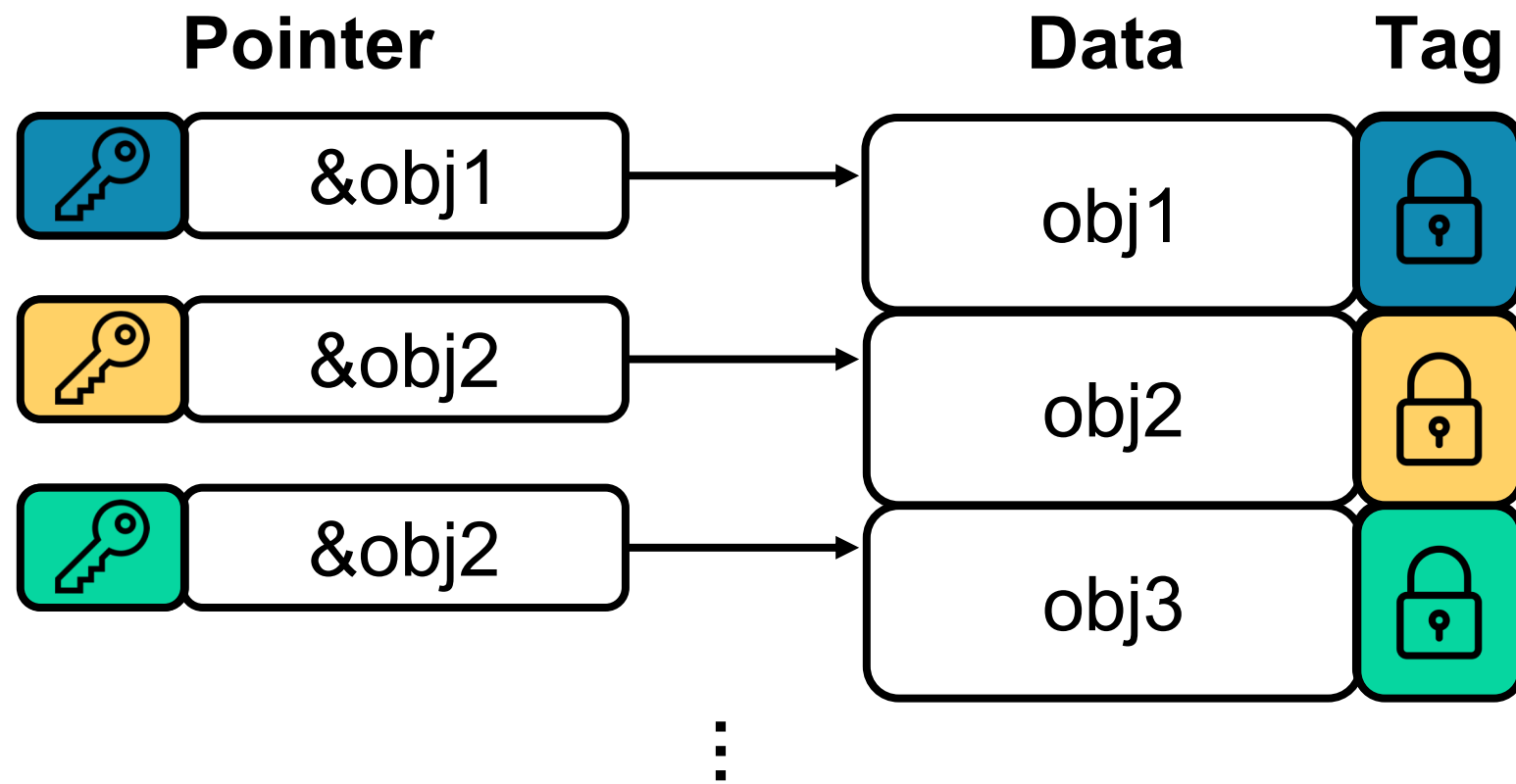
Pointer Tag
(Key)

.....

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

(3) Tag Allocation

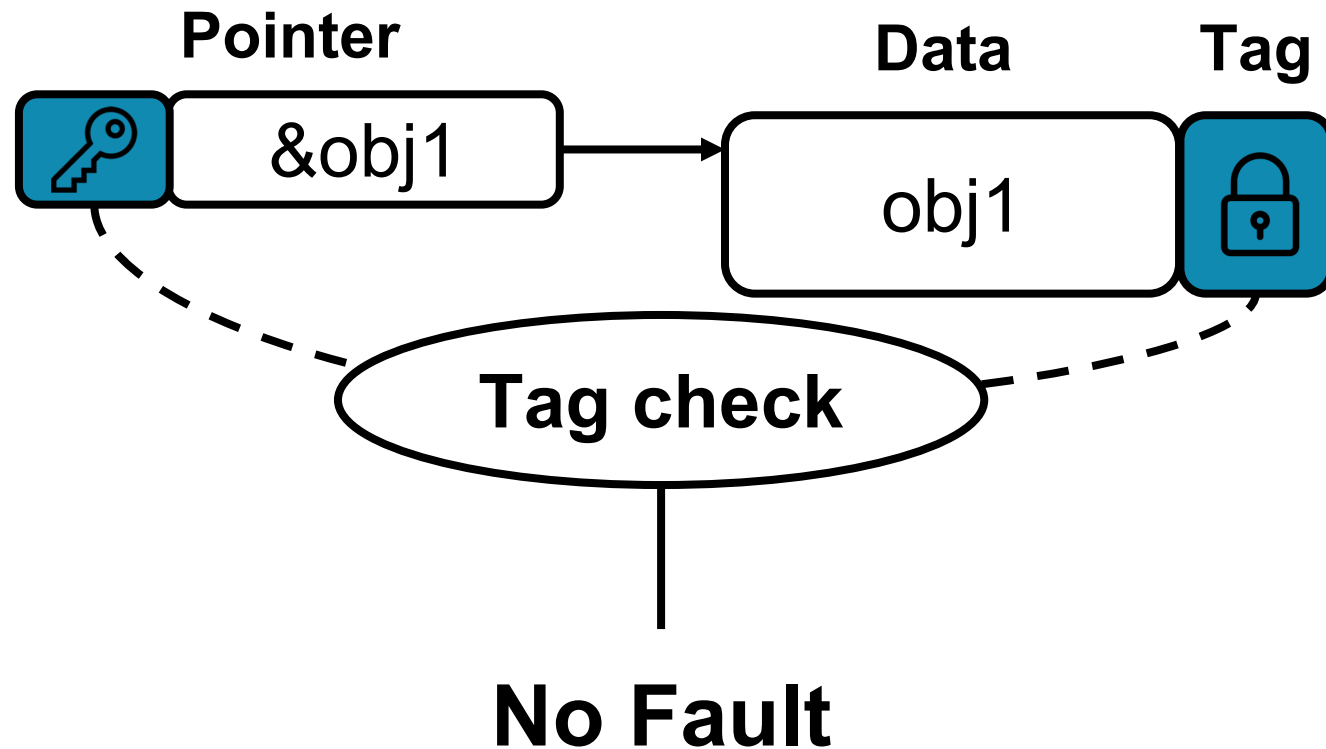
New instructions to **create a random tag** and **load/store memory tags**



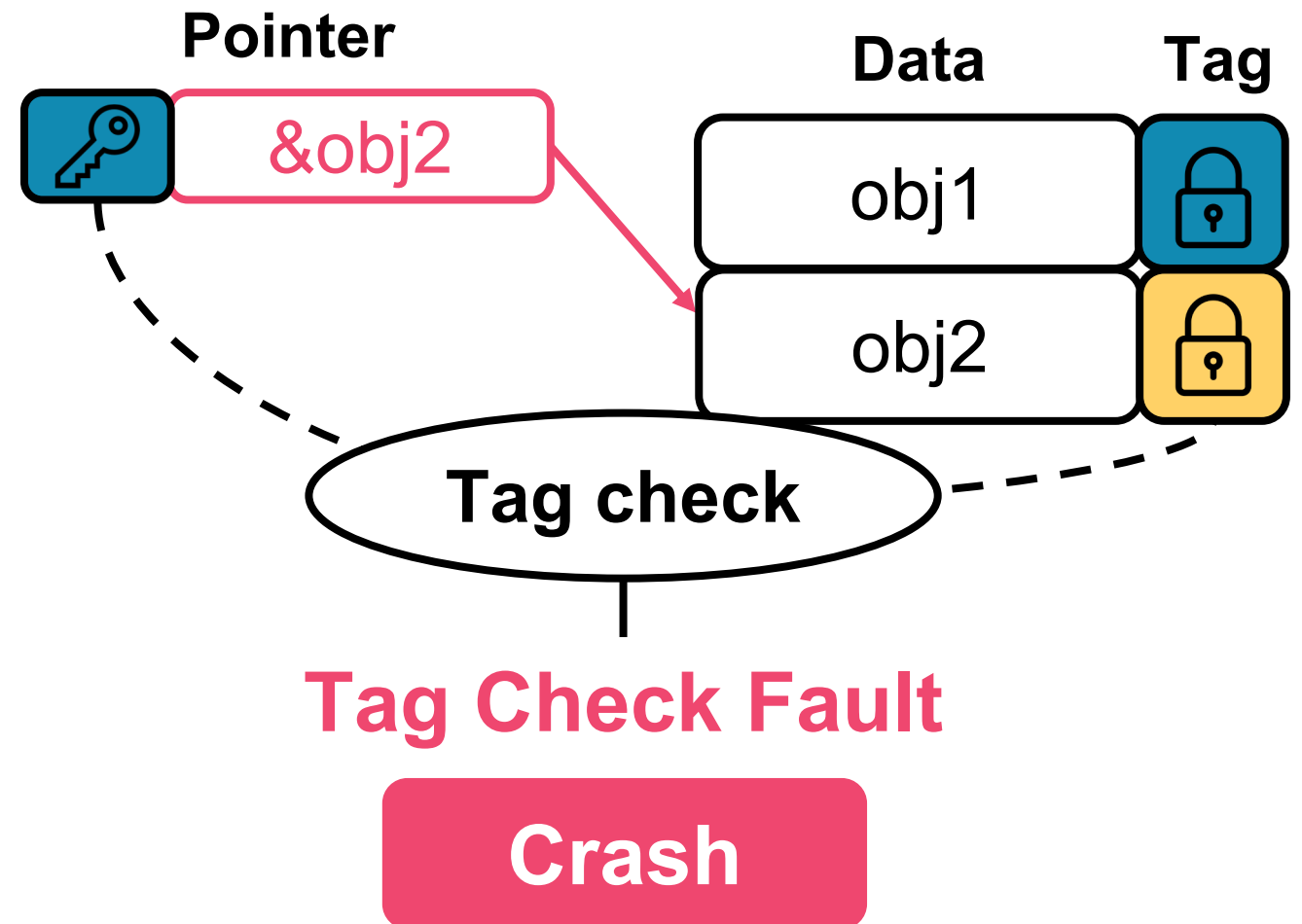
(4) Tag Check

Transparently done by hardware

Valid memory access



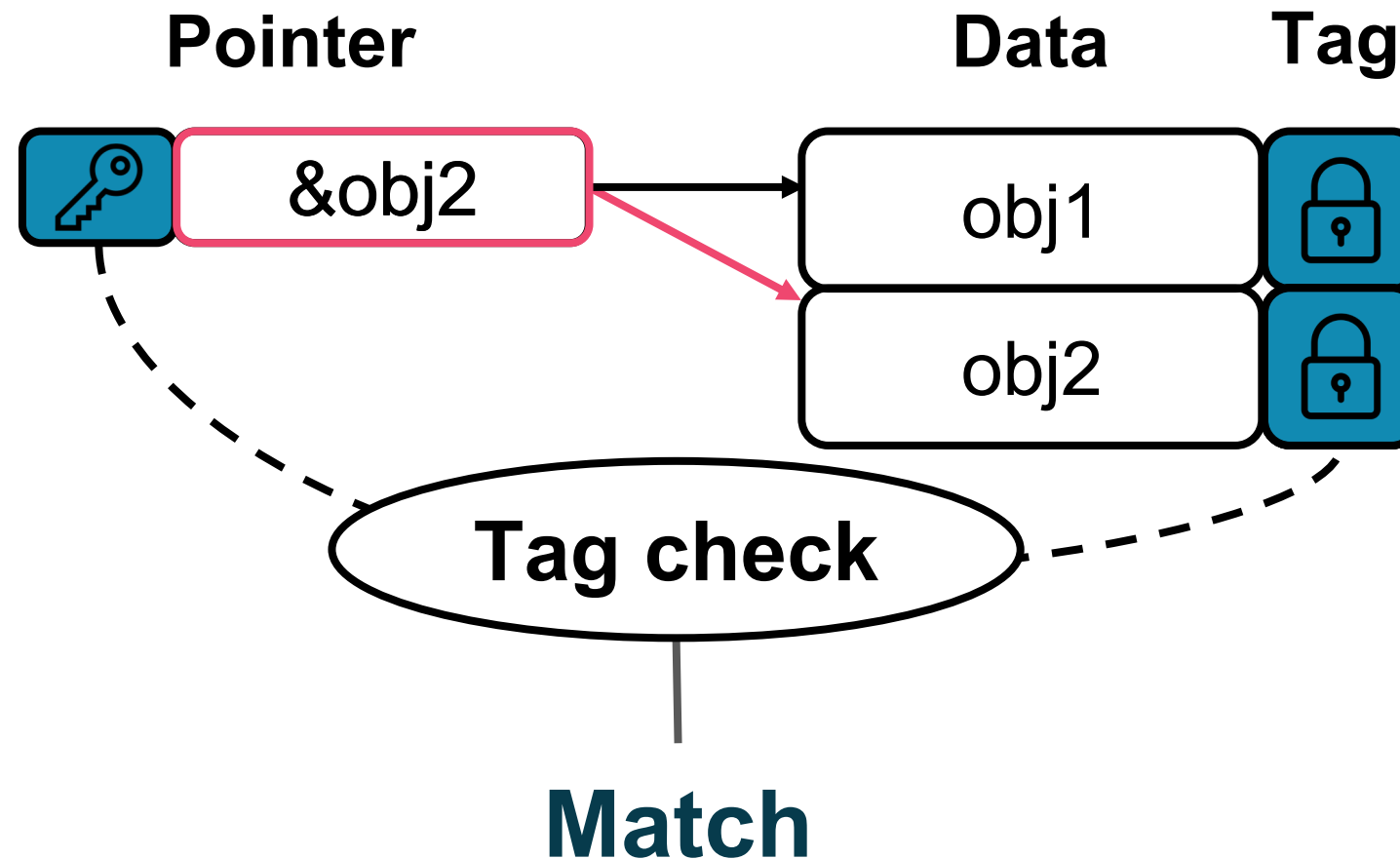
Invalid memory access



How to Bypass MTE?

(1) Tag Collision (16 possible tags)

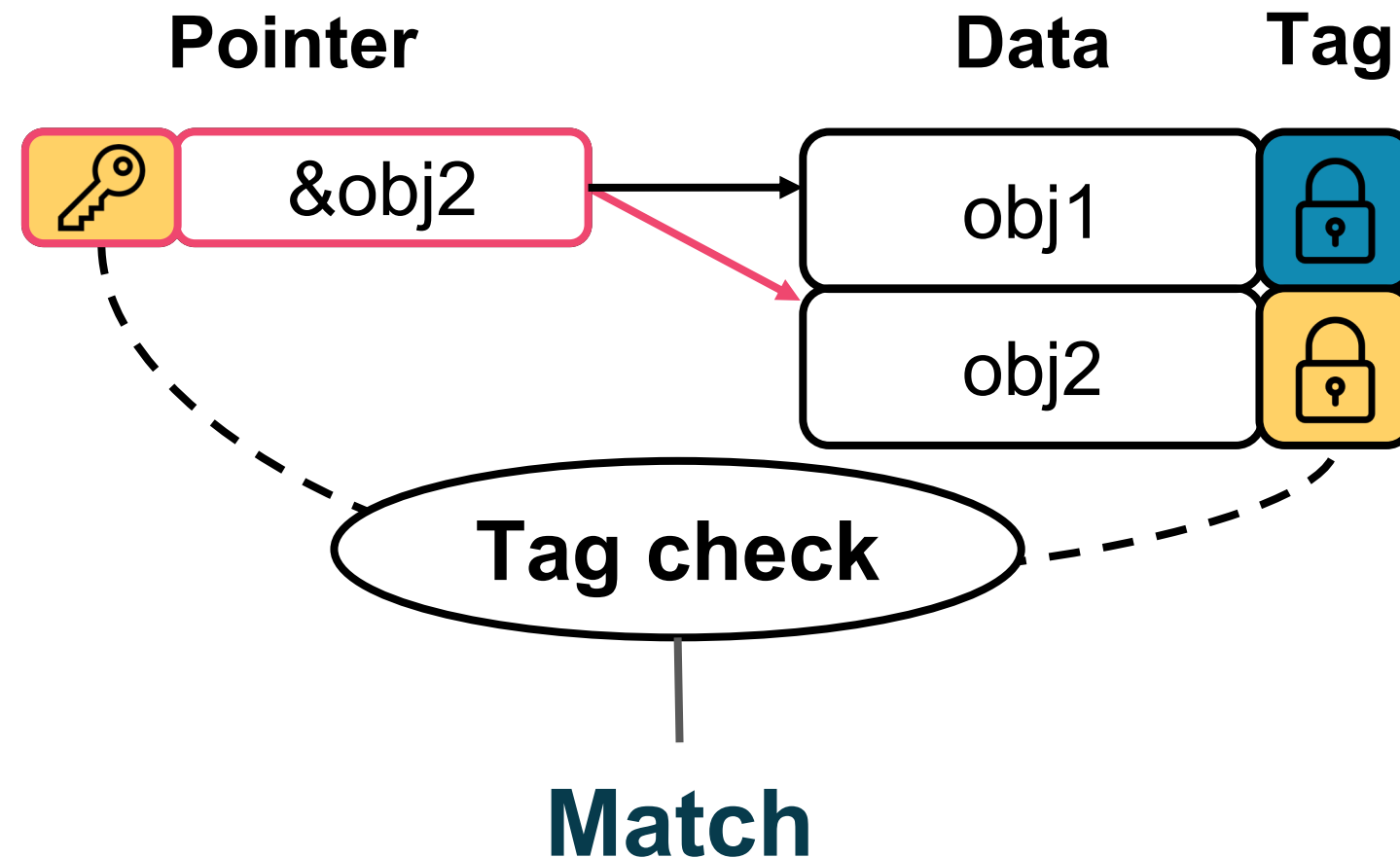
Wait until the **pointer tag** matches the **target memory tag**



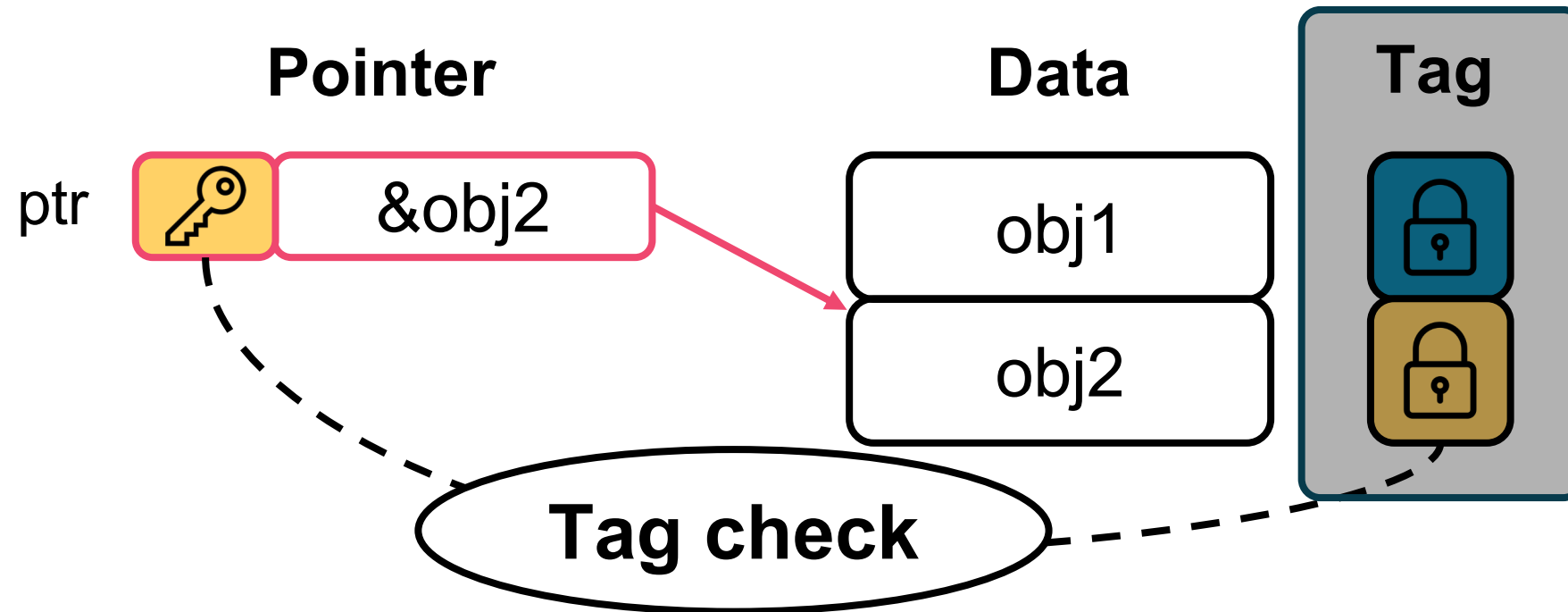
How to Bypass MTE?

(2) Pointer Tag Corruption

Corrupt the **pointer tag** to the **target memory tag**



Challenge: Random Tags



Match \Rightarrow *Attack Succeeds 1/16 (6%)*

Mismatch
Crash

\Rightarrow *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

ARM Memory Tagging Extension

arm



Cache Side-Channel

Cache



Speculative Execution

if (cond)

True

False



Real-world MTE Bypass Attack



JS

MTE

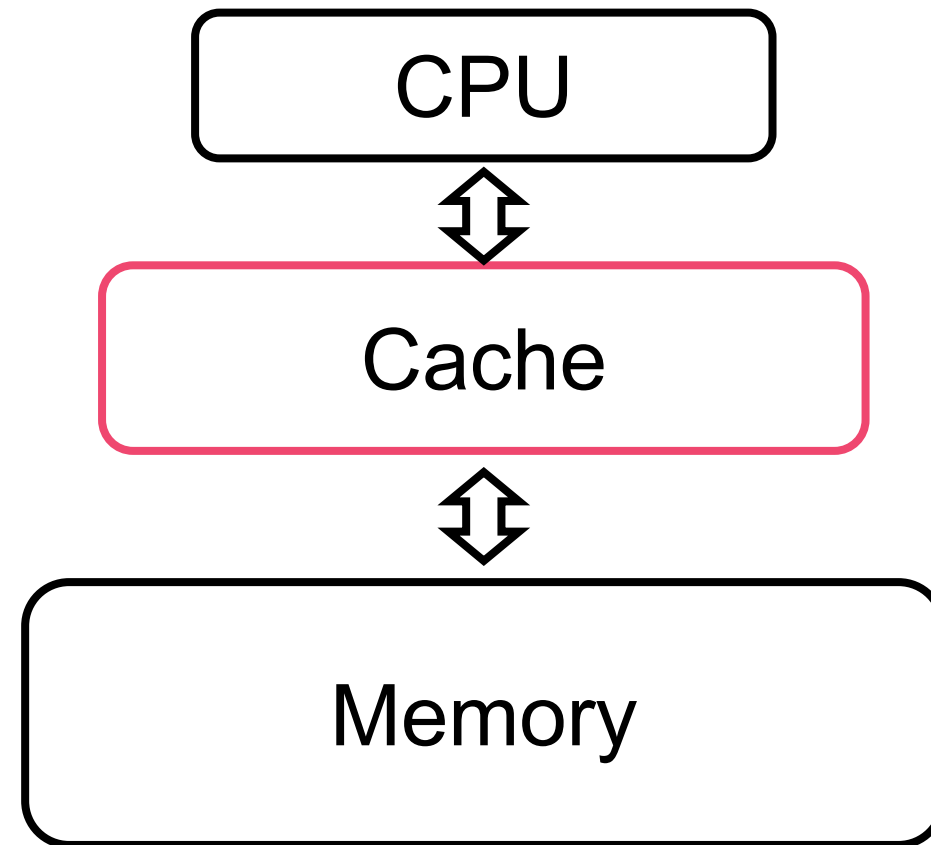
TikTag: MTE Tag Side-Channel



MTE



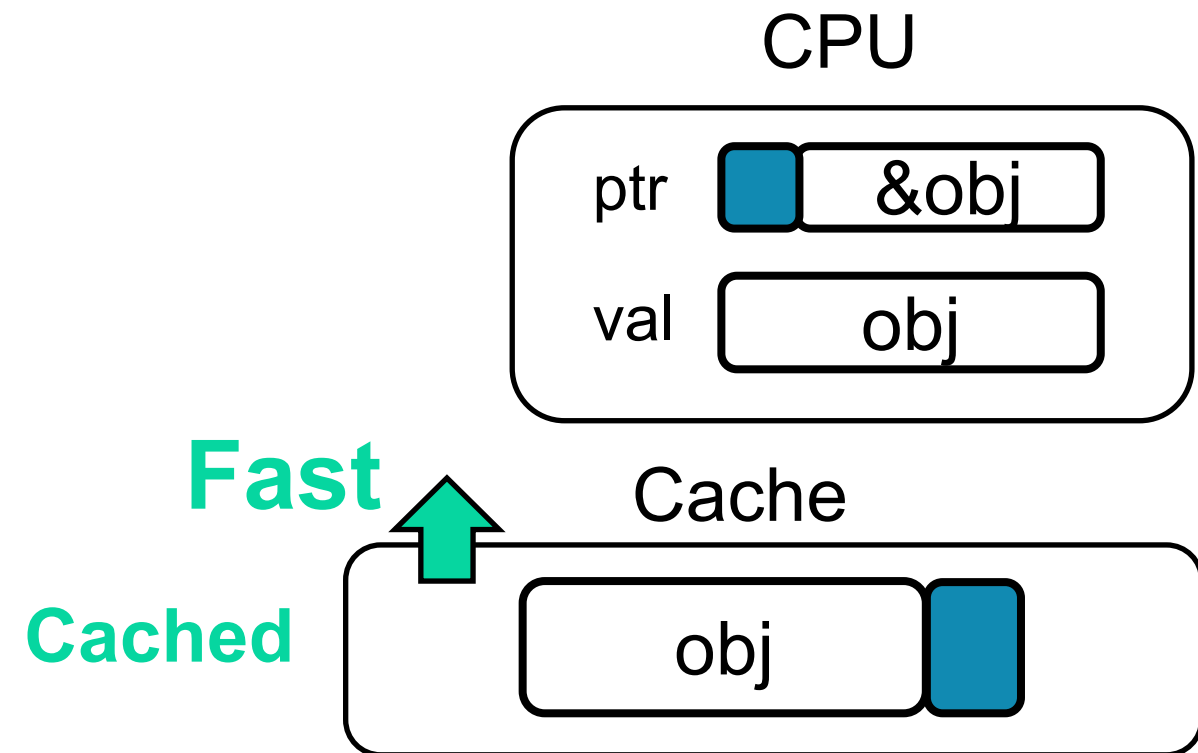
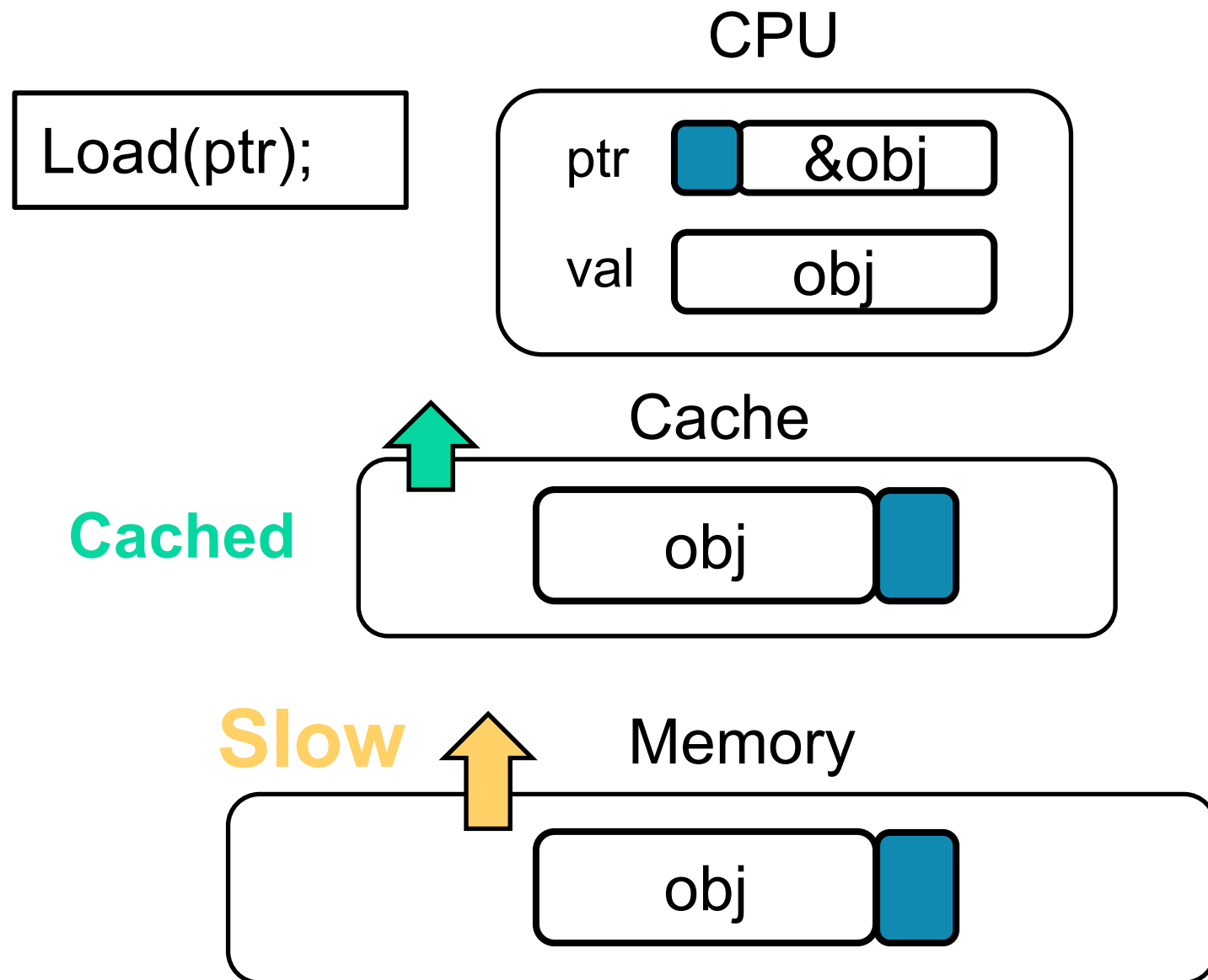
What is Cache?



What is Cache?

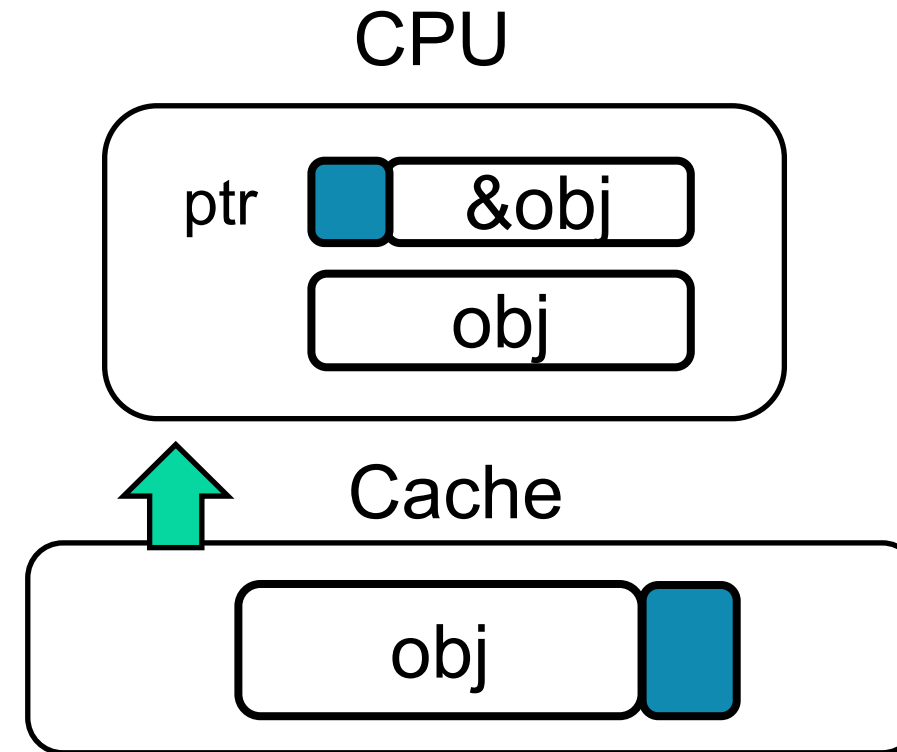
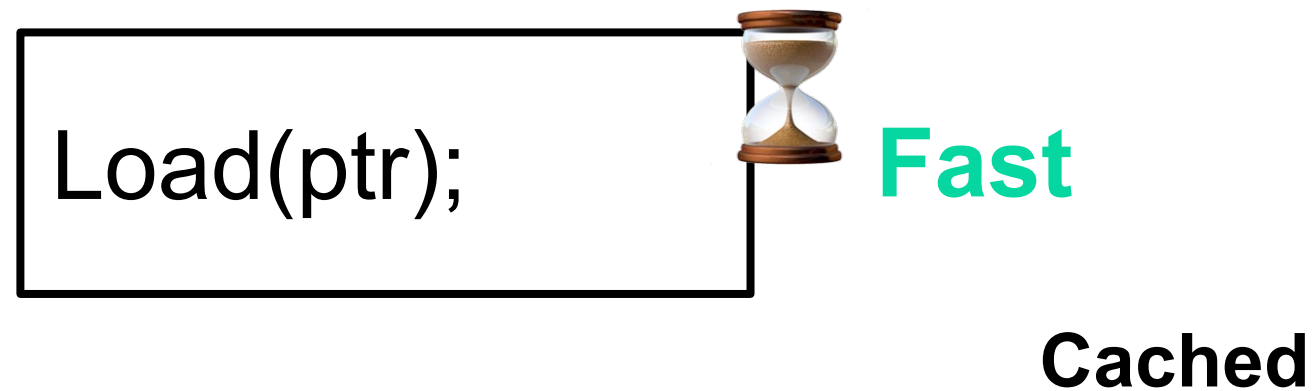
First Access : Slow

Second Access : Fast



Cache Side-Channel

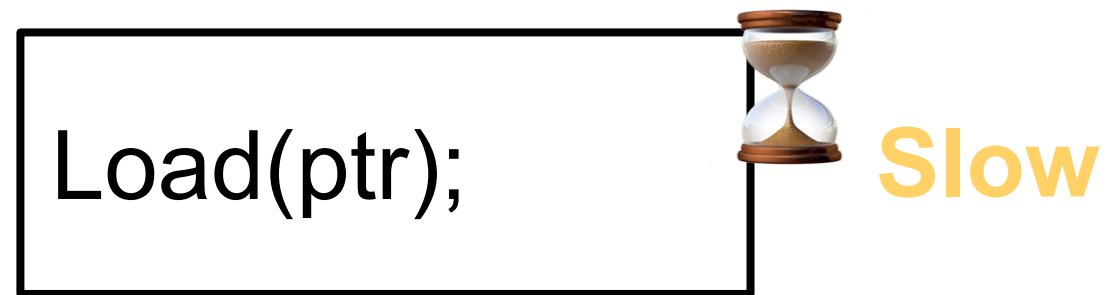
Q. Has obj been accessed?



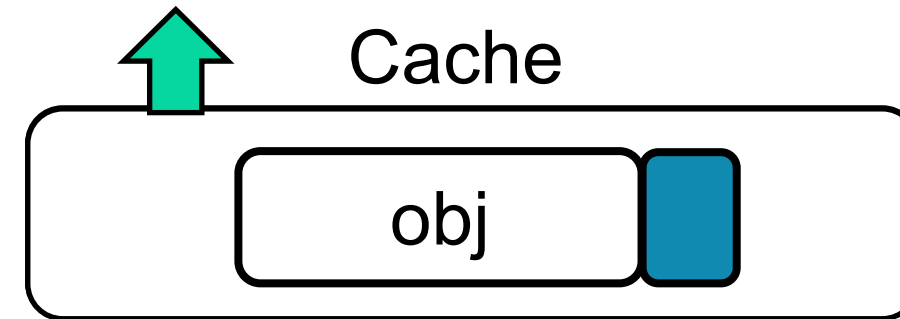
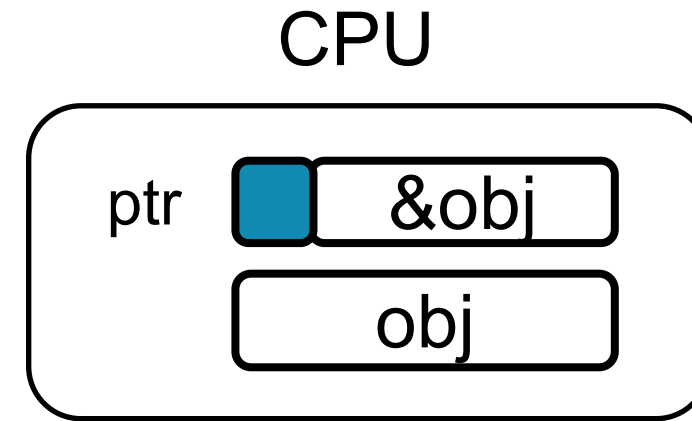
A. `ptr` has been accessed!

What is Cache Side-Channel?

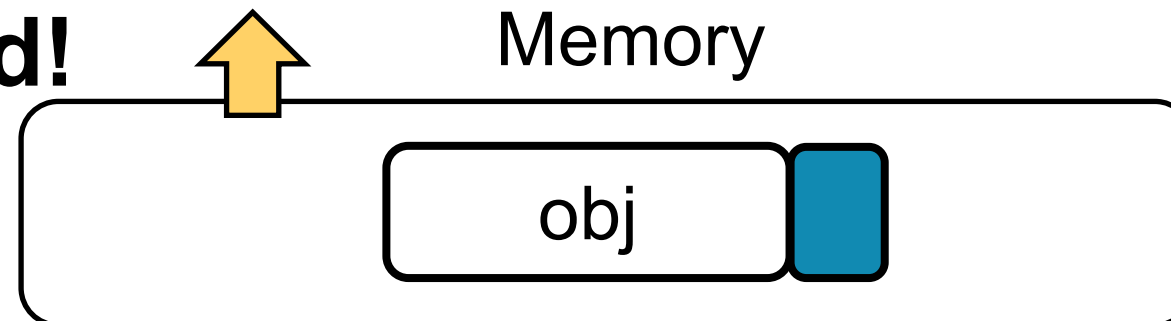
Q. Has obj been accessed?



Not
Cached



A. ptr has NOT been accessed!



Exploit cache side-channel

→ **Leak whether an address is accessed**

Roadmap

ARM Memory Tagging Extension

arm



Cache Side-Channel

Cache



Speculative Execution

if (cond)

False



Real-world MTE Bypass Attack



JS

MTE

TikTag: MTE Tag Side-Channel

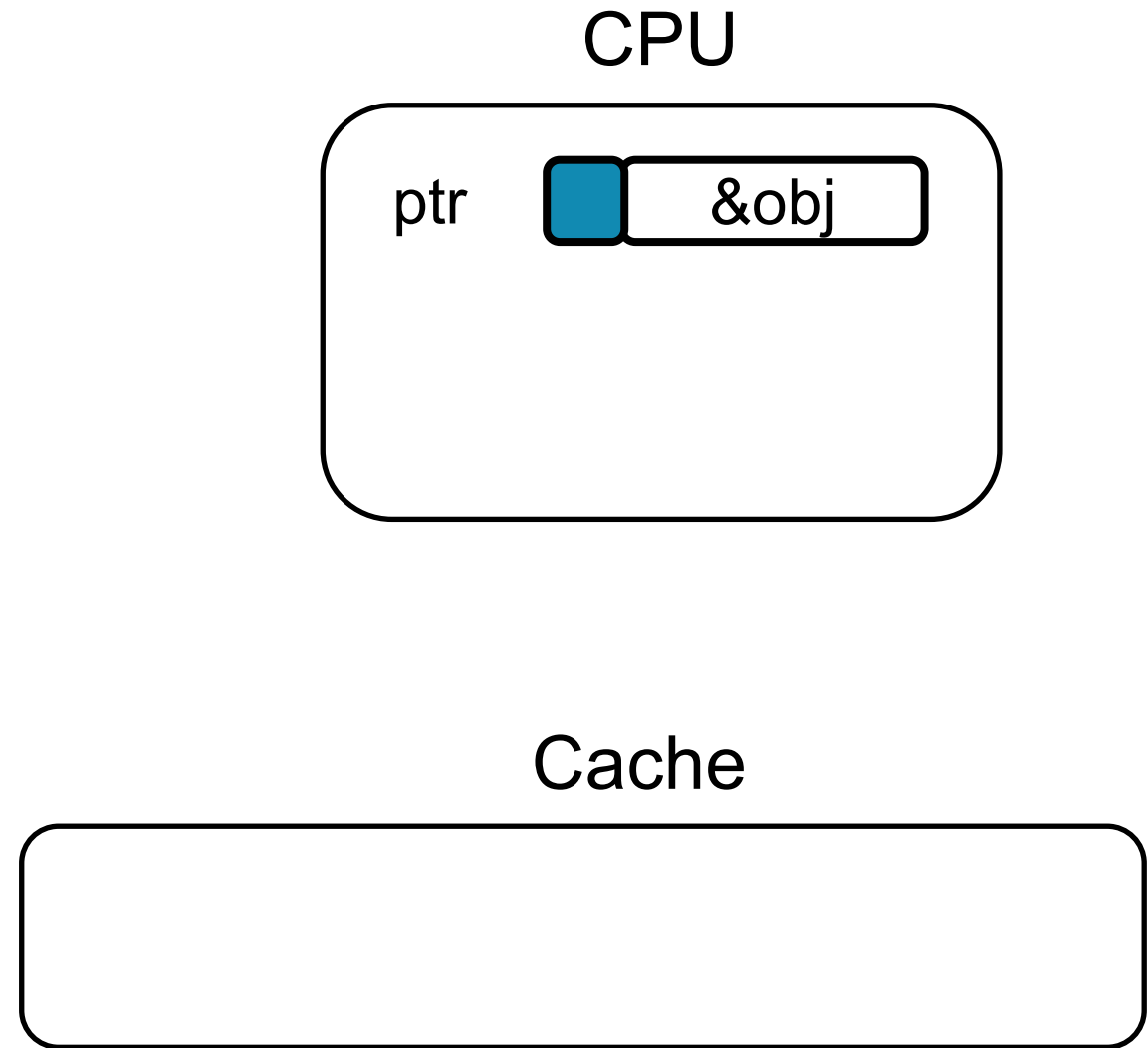


MTE

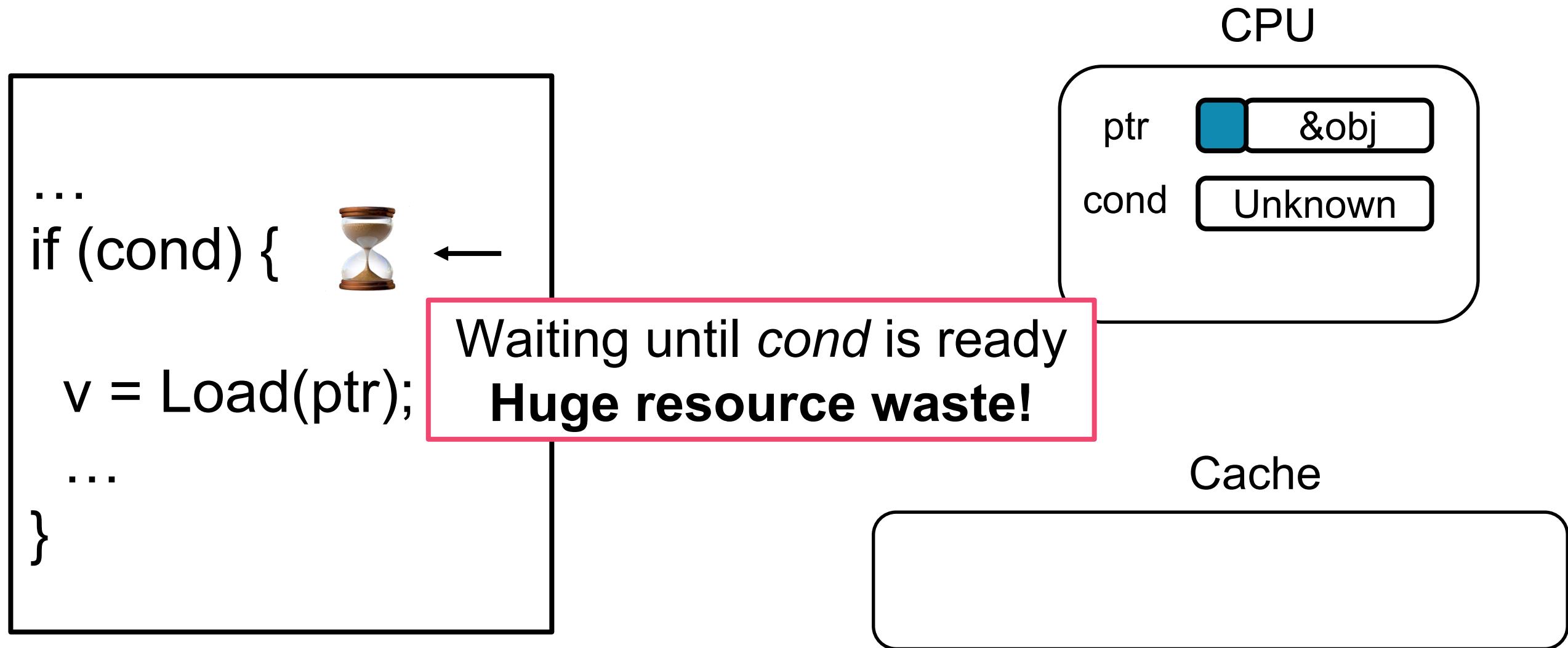


What is Speculative Execution?

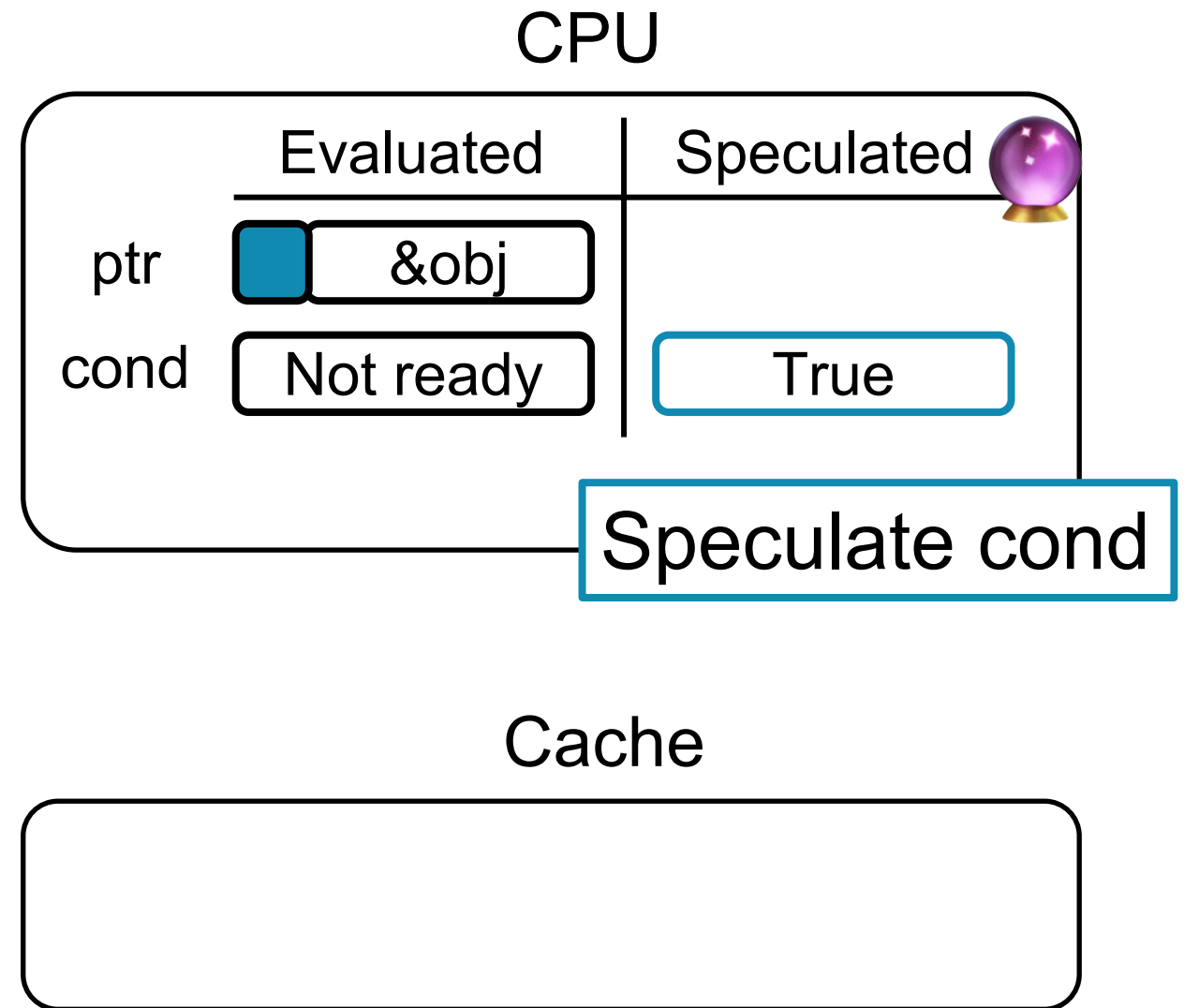
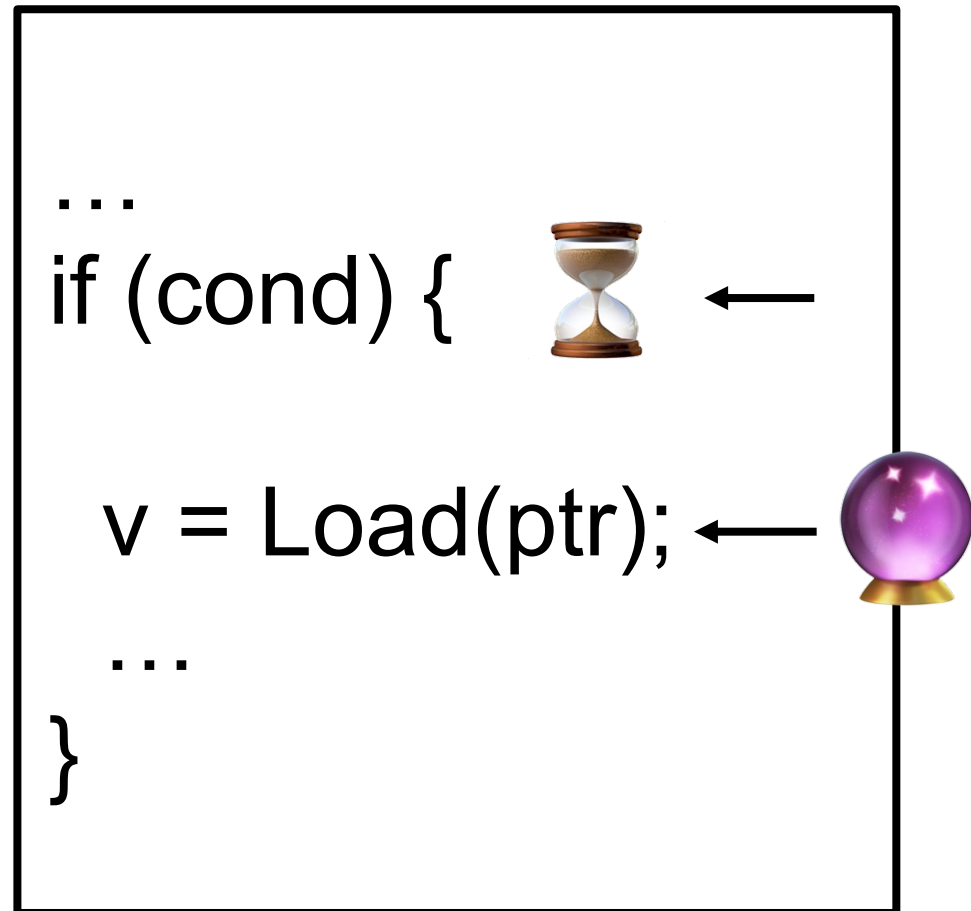
```
...  
if (cond) { ←  
  
    v = Load(ptr);  
    ...  
}
```



What is Speculative Execution?

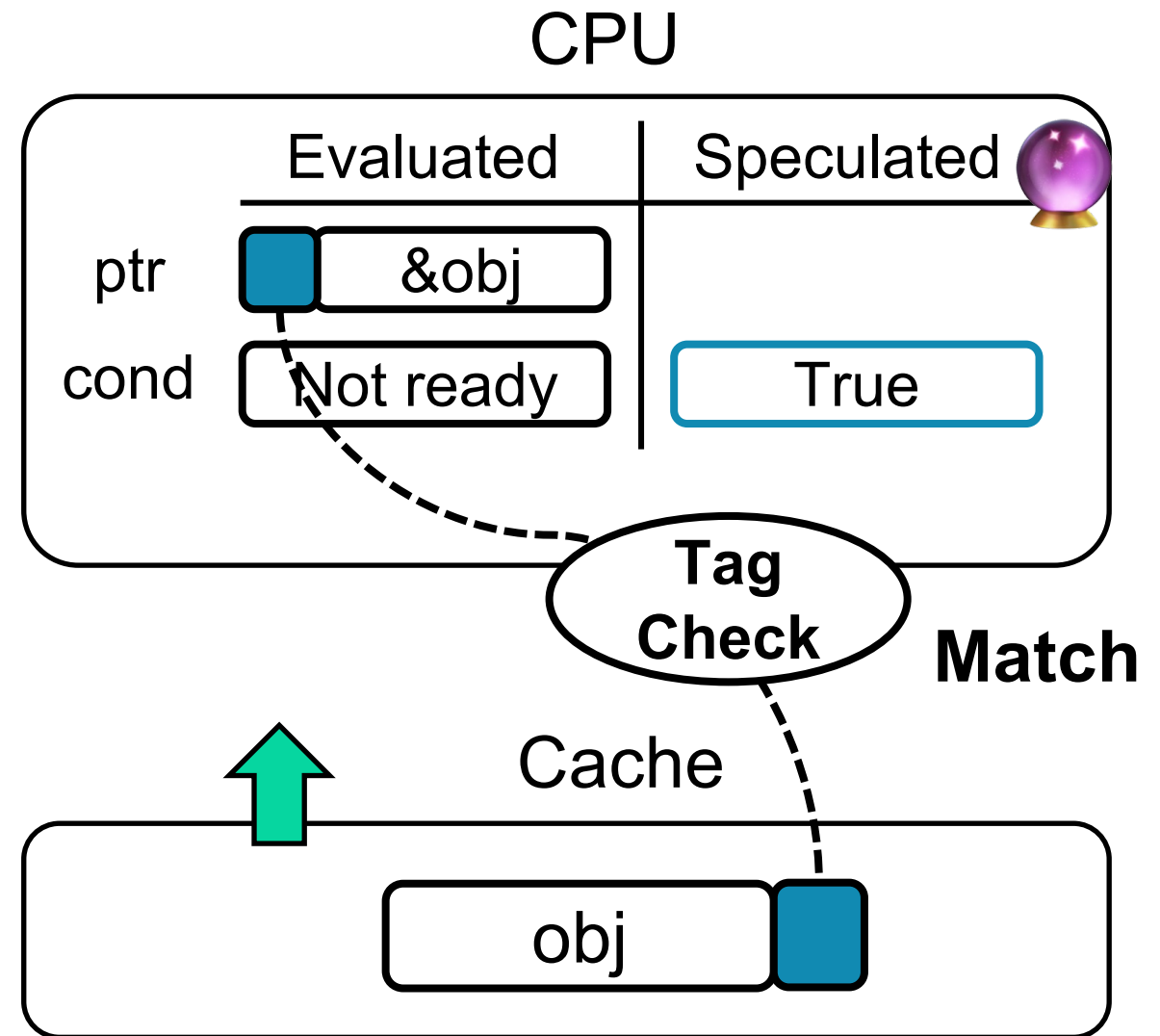


What is Speculative Execution?

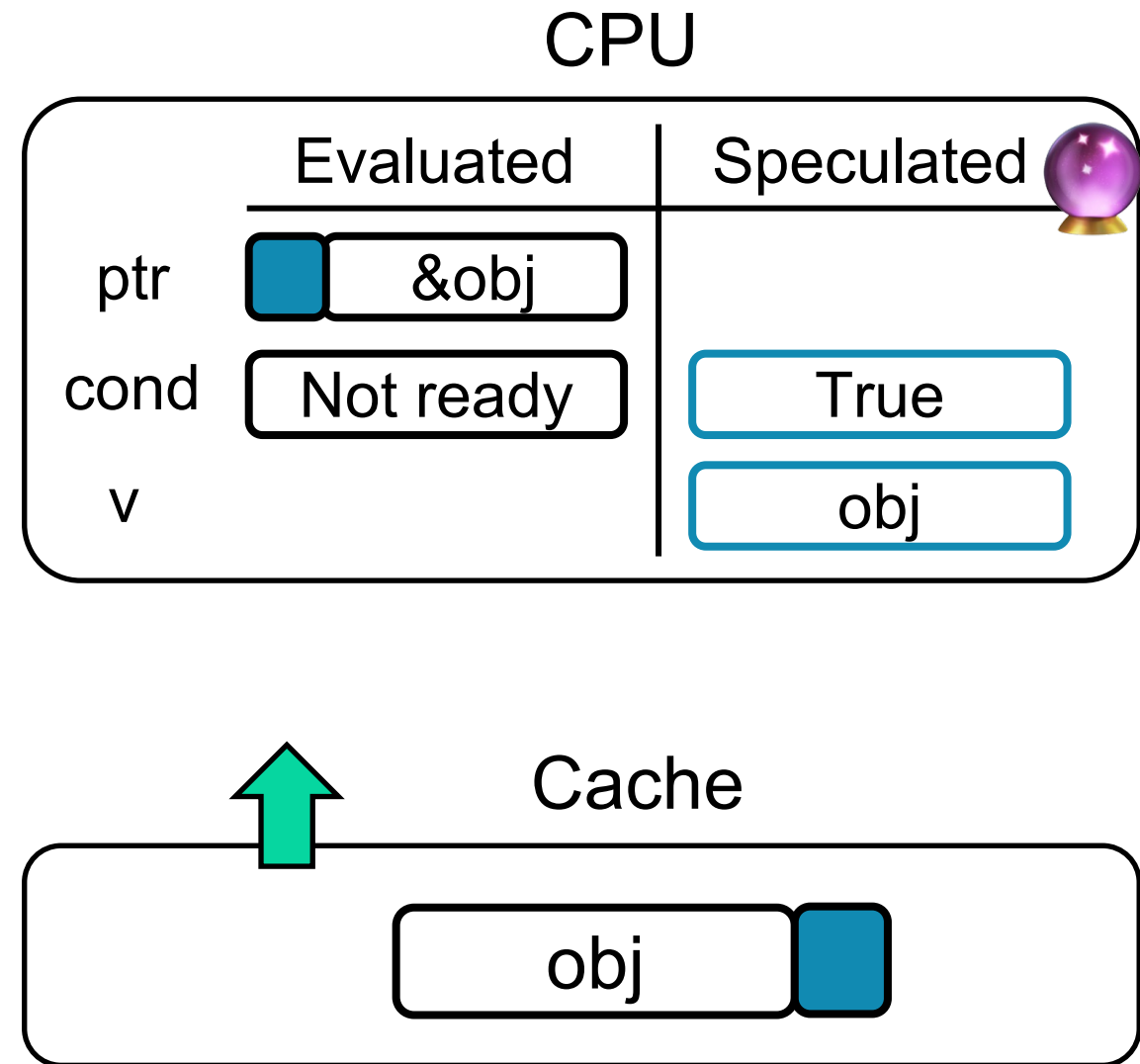
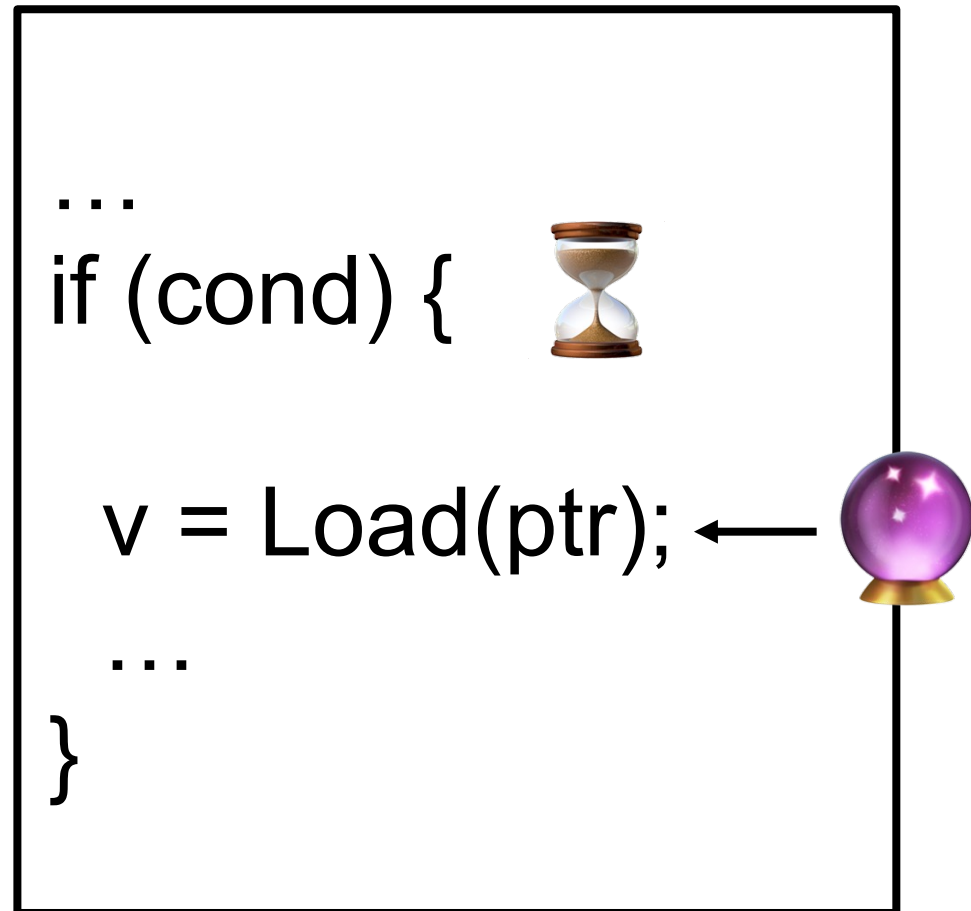


What is Speculative Execution?

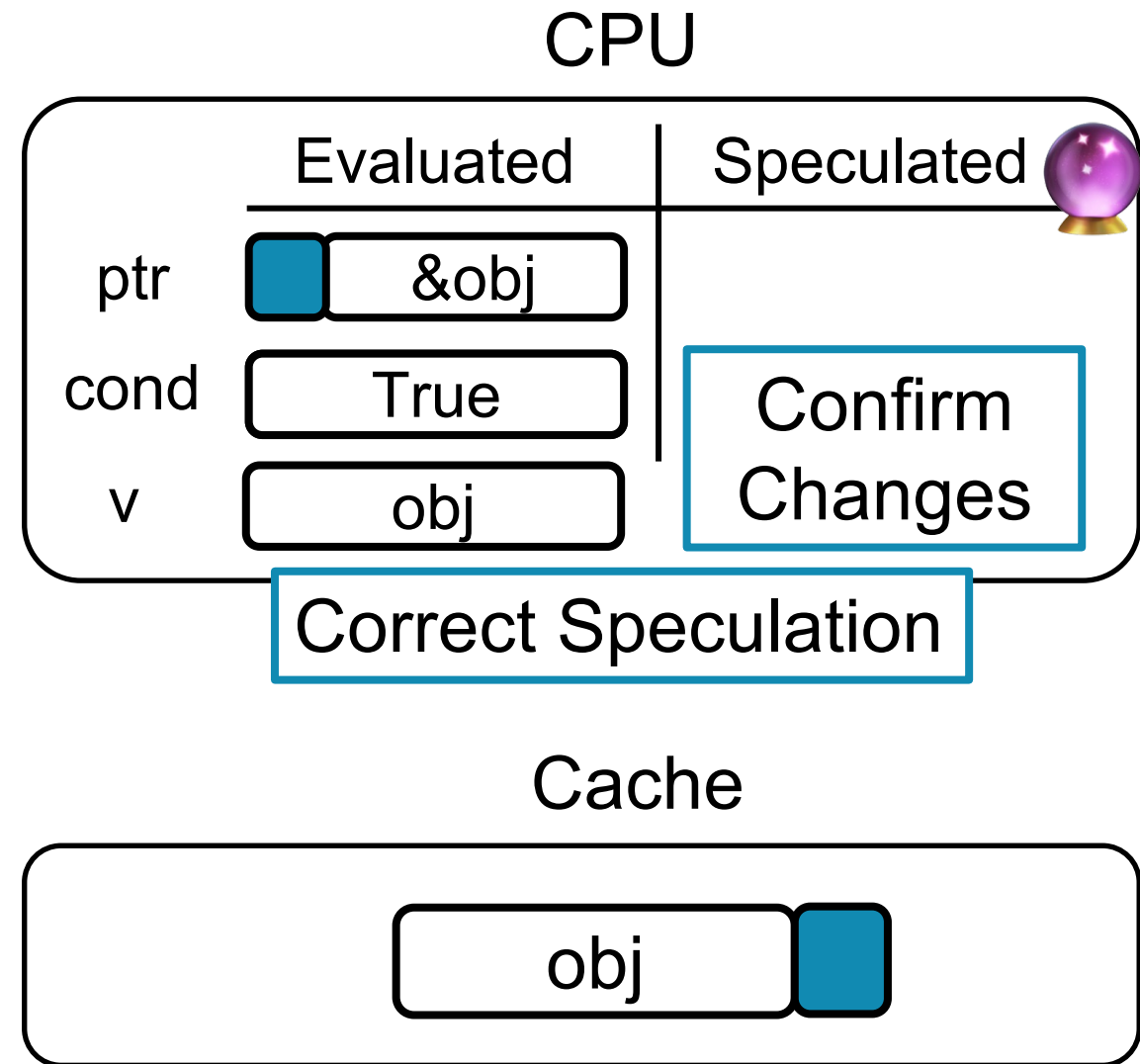
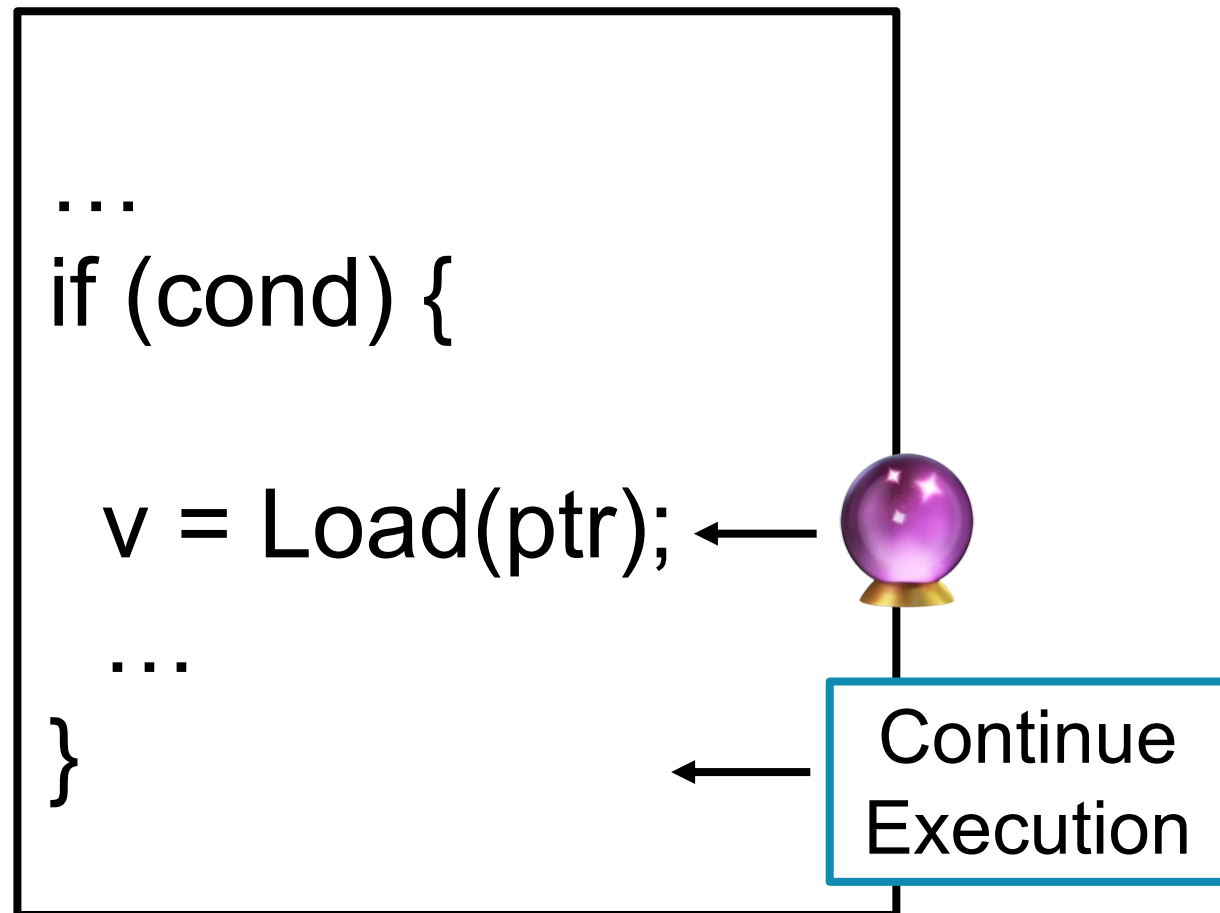
```
...  
if (cond) {  
    v = Load(ptr);  
    ...  
}
```



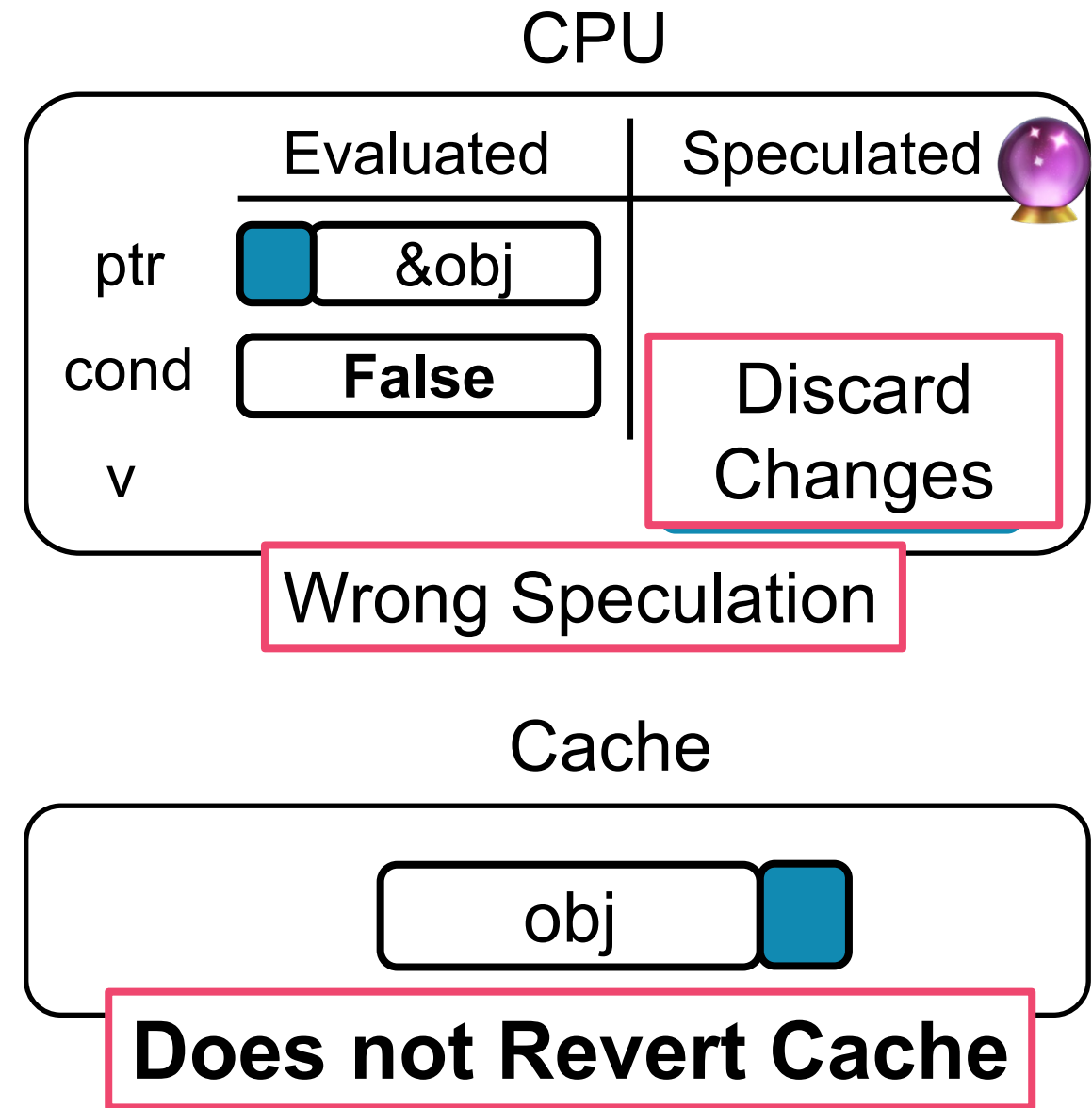
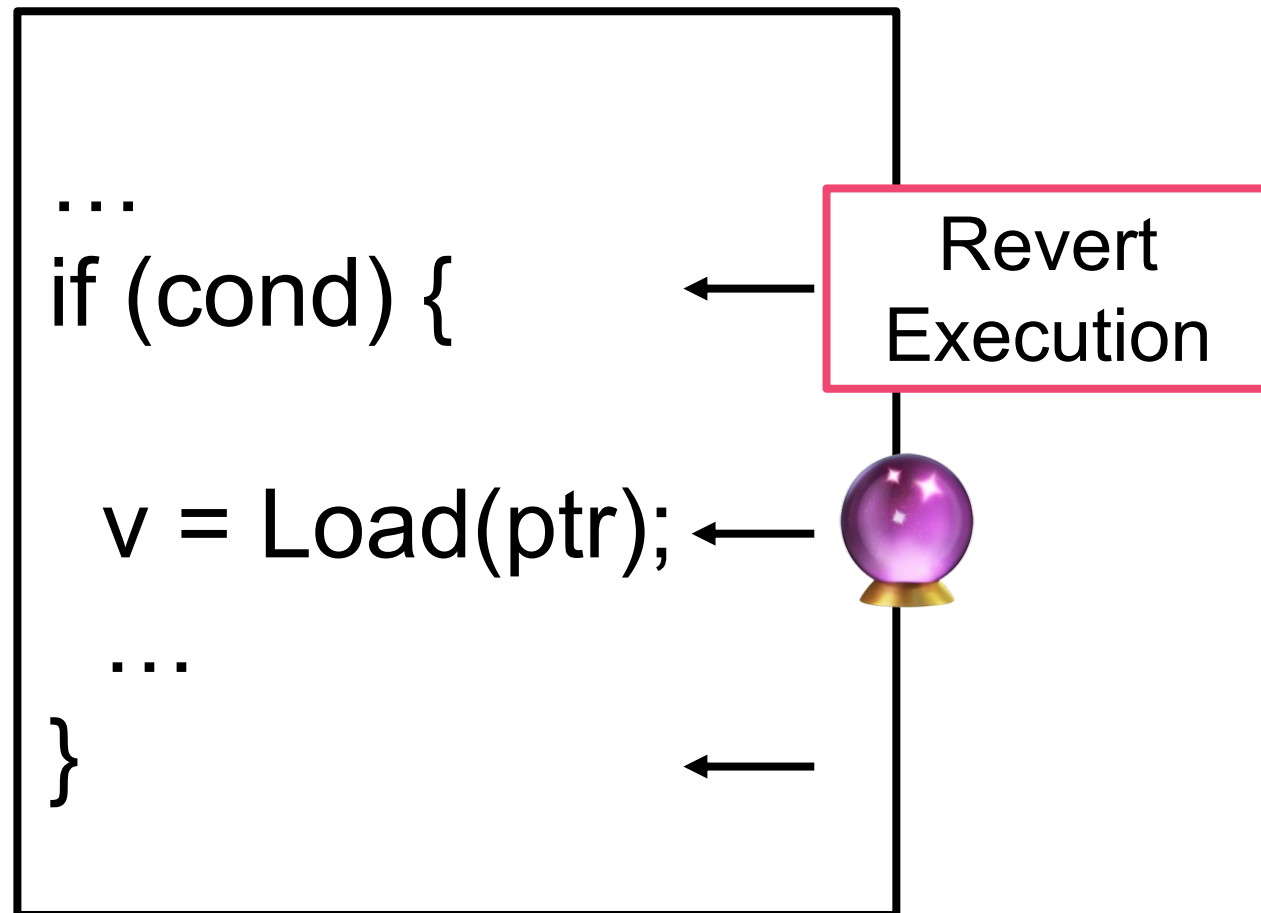
What is Speculative Execution?



What is Speculative Execution?

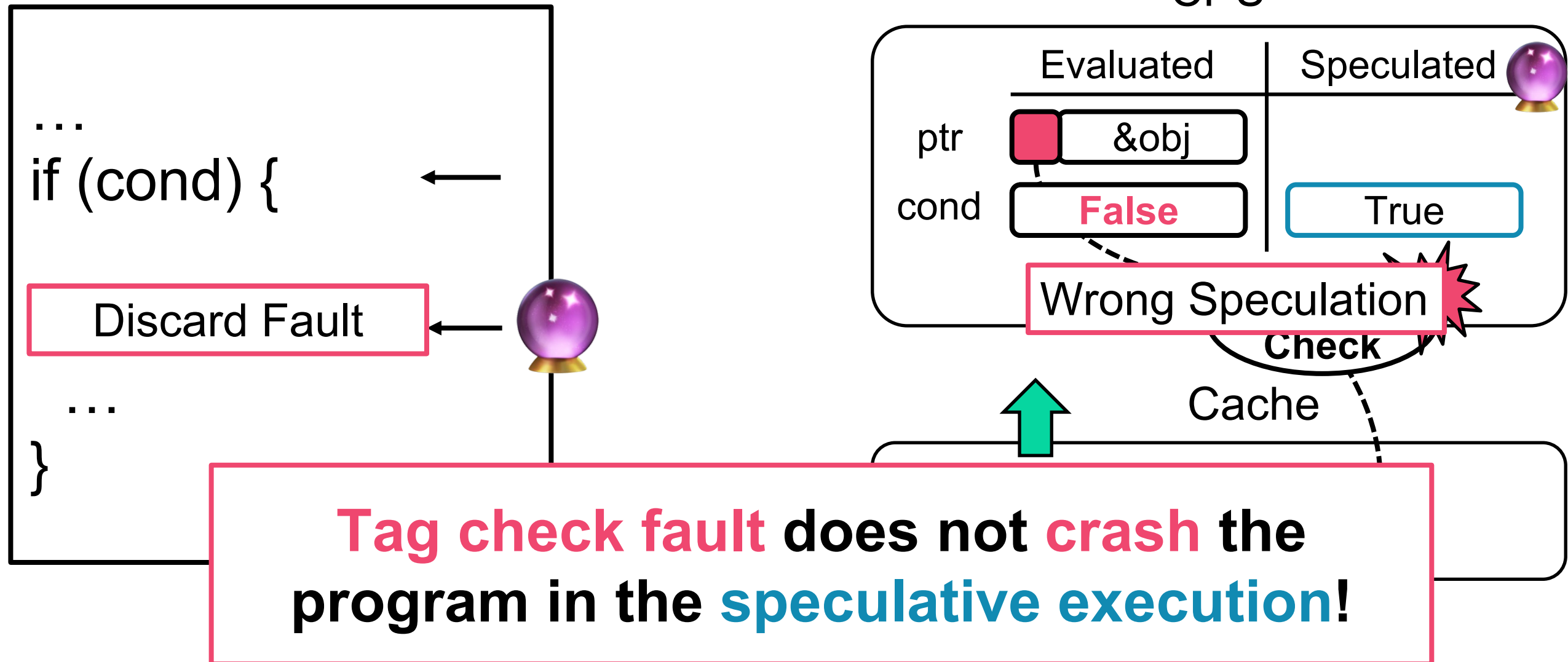


What is Speculative Execution?



Tag check fault on Speculative Execution?

CPU



Exploit cache side-channel

→ **Leak whether an address is accessed**

Exploit speculative execution

→ **Avoid crash on tag check fault**

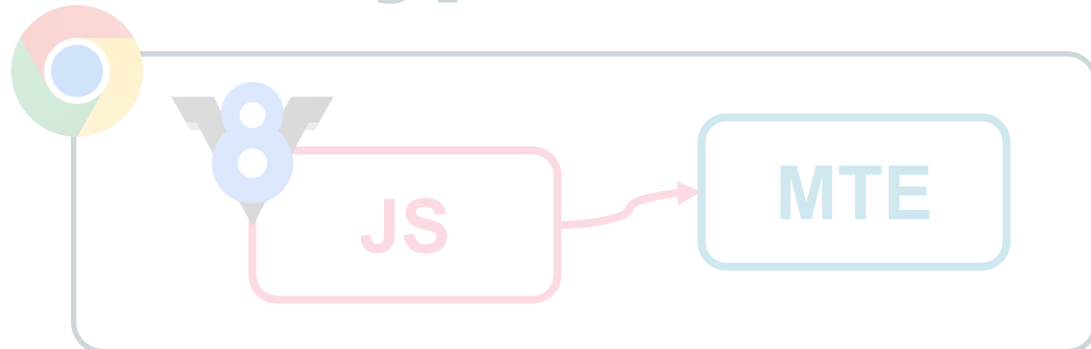
Roadmap

ARM Memory Tagging Extension

arm



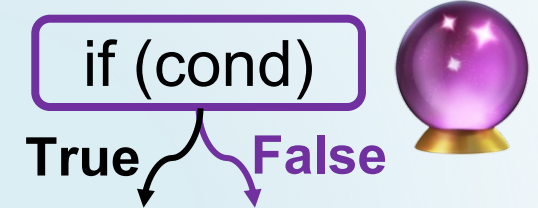
Real-world MTE Bypass Attack



Cache Side-Channel



Speculative Execution

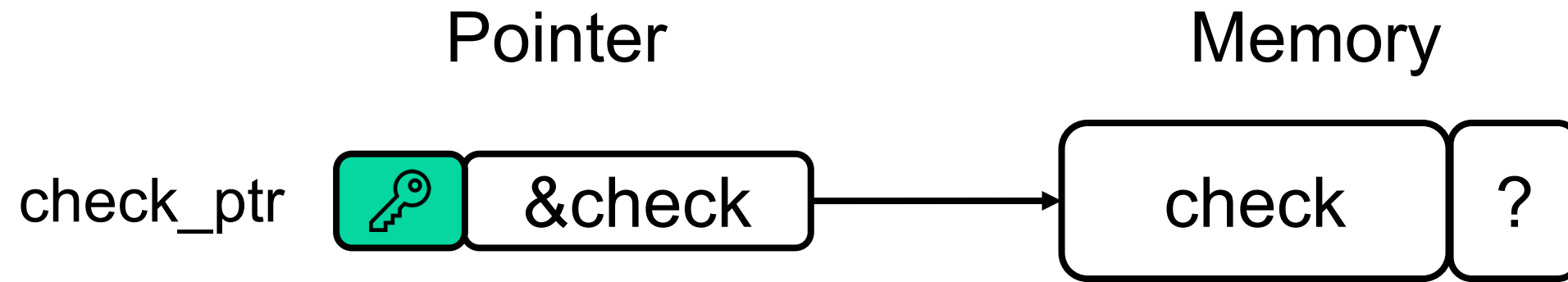


TikTag: MTE Tag Side-Channel



MTE Side-channel attack

Goal: Leak the memory tag given a pointer



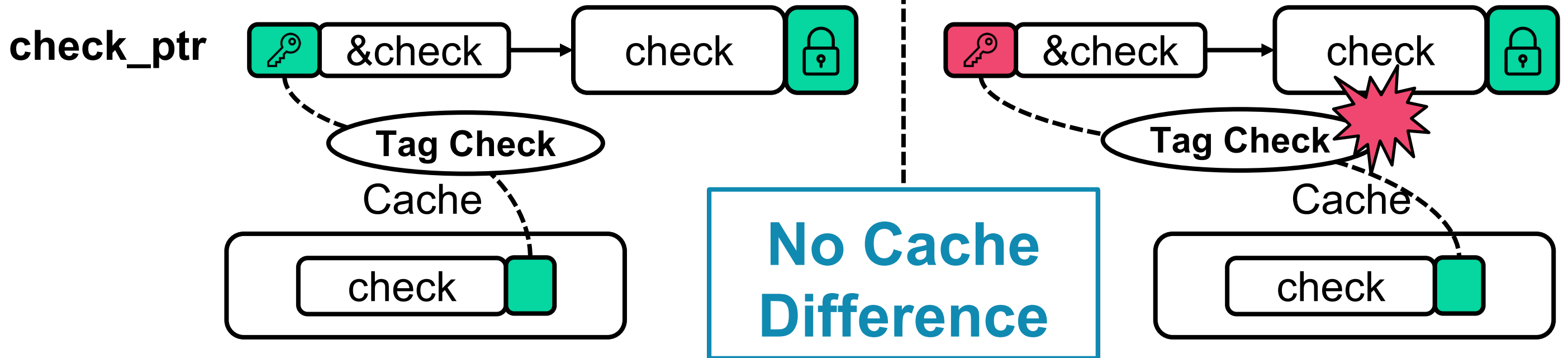
MTE Side-channel attack

Two test cases:

```
Access(check_ptr);
```

A. **Valid** tag in check_ptr

B. **Invalid** tag in check_ptr



MTE Side-channel attack

Two test cases:

```
Access(check_ptr); Access(test_ptr);
```

A. **Valid** tag in check_ptr

B. **Invalid** tag in check_ptr



Cache
Difference?

A. Valid tag in check_ptr

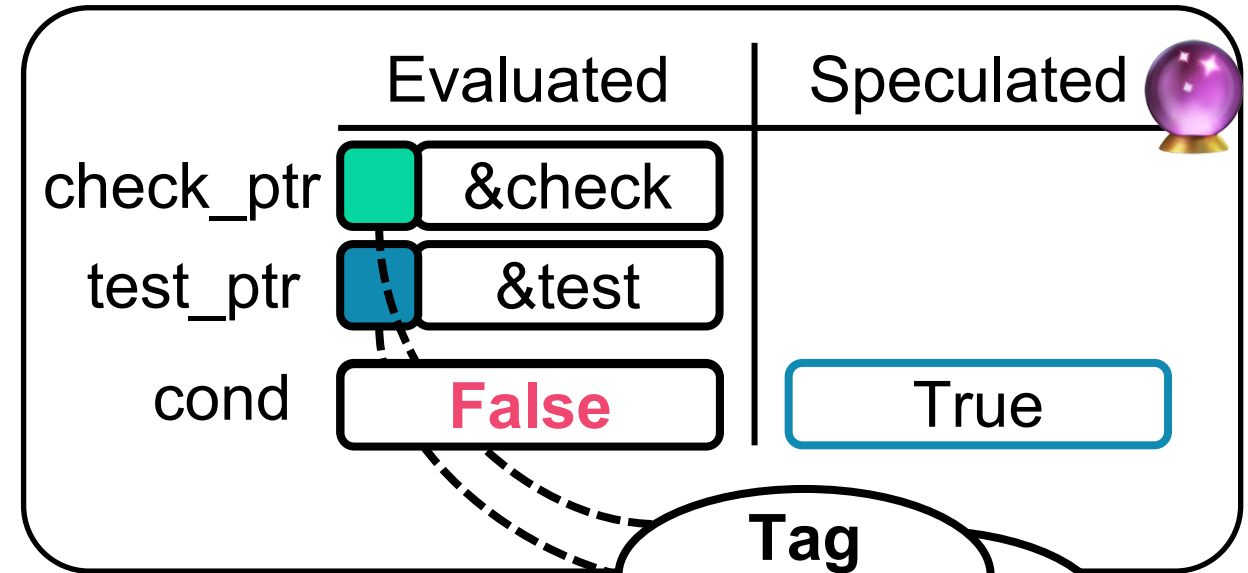
Tag Leakage Gadget

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



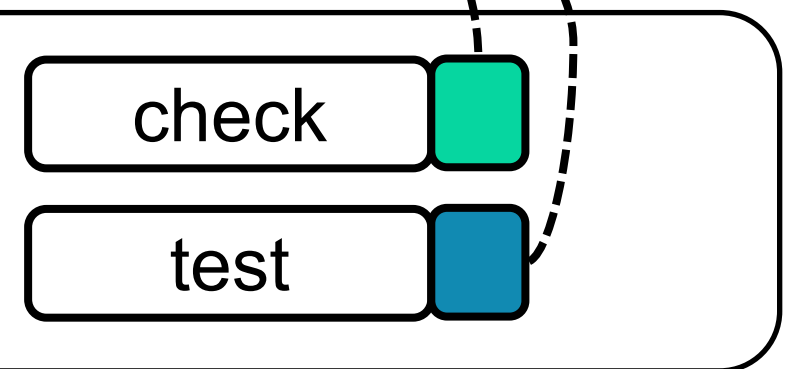
Cache contains both check and test

CPU



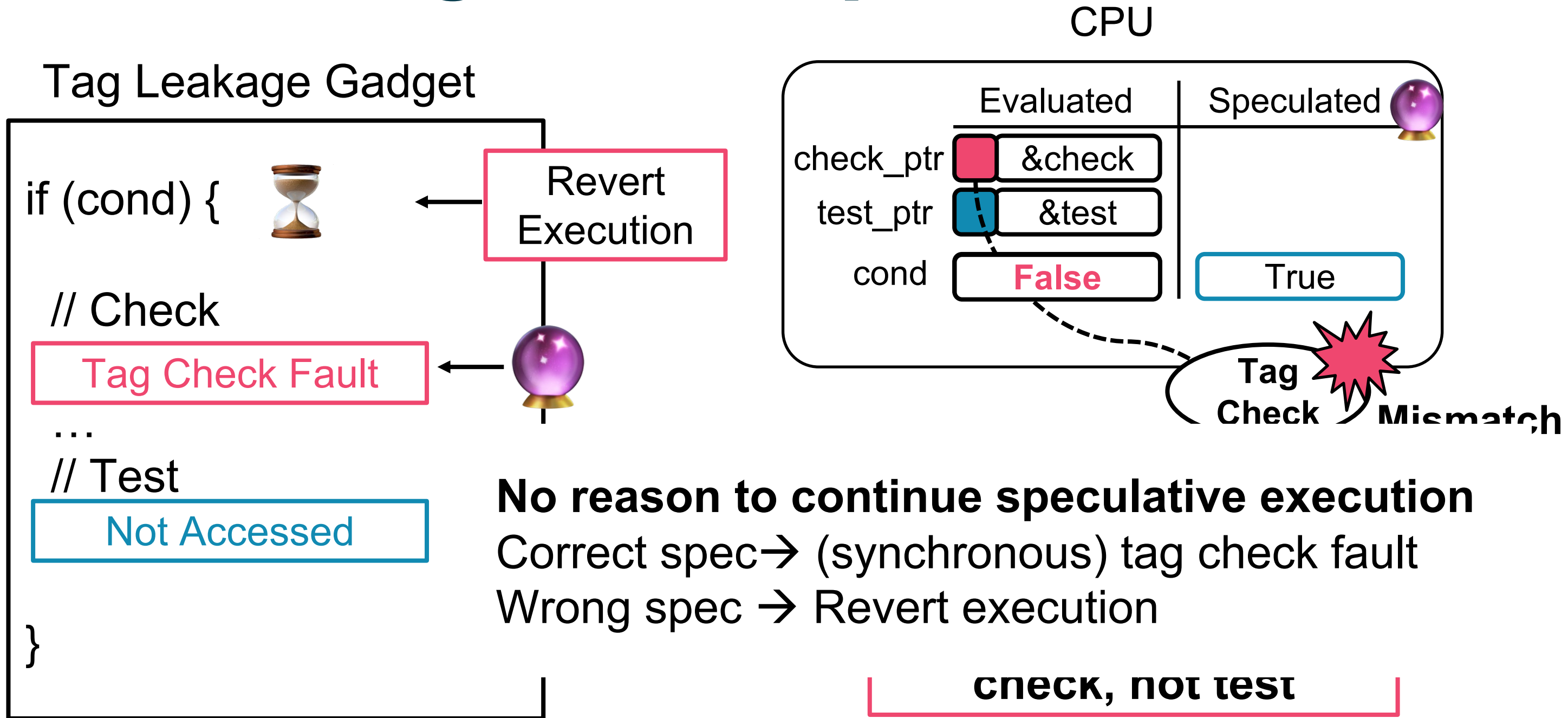
Tag Check

Cache



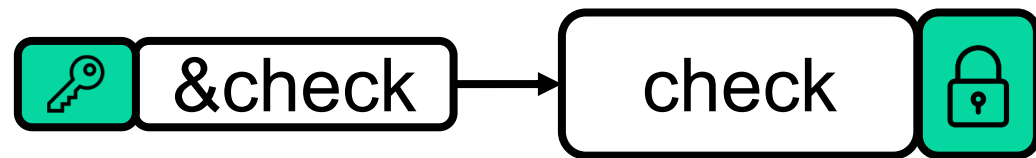
Cache Match

B. Invalid tag in check_ptr

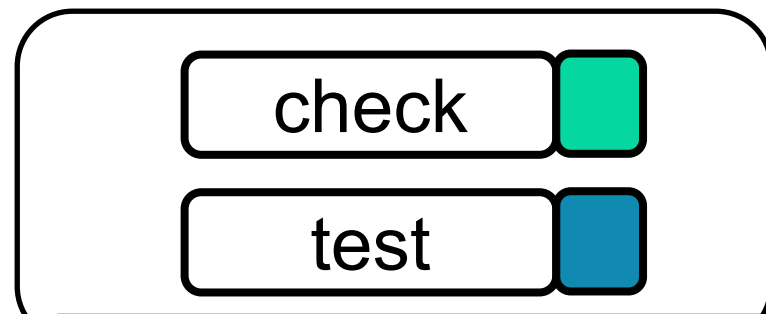


Leak by Cache Side-Channel

A. **Valid** tag in check_ptr



Cache

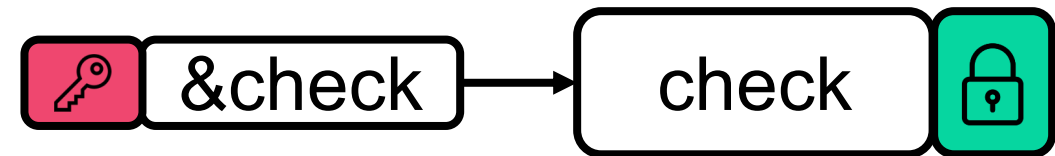


```
Load(test_ptr);
```

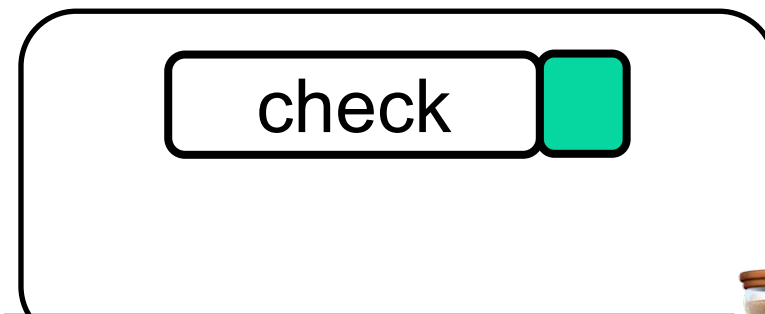


Fast

B. **Invalid** tag in check_ptr



Cache



```
Load(test_ptr);
```



Slow

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**

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

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>

```
    :k_ptr;
```

StickyTags – VUSec, IEEE S&P 2024

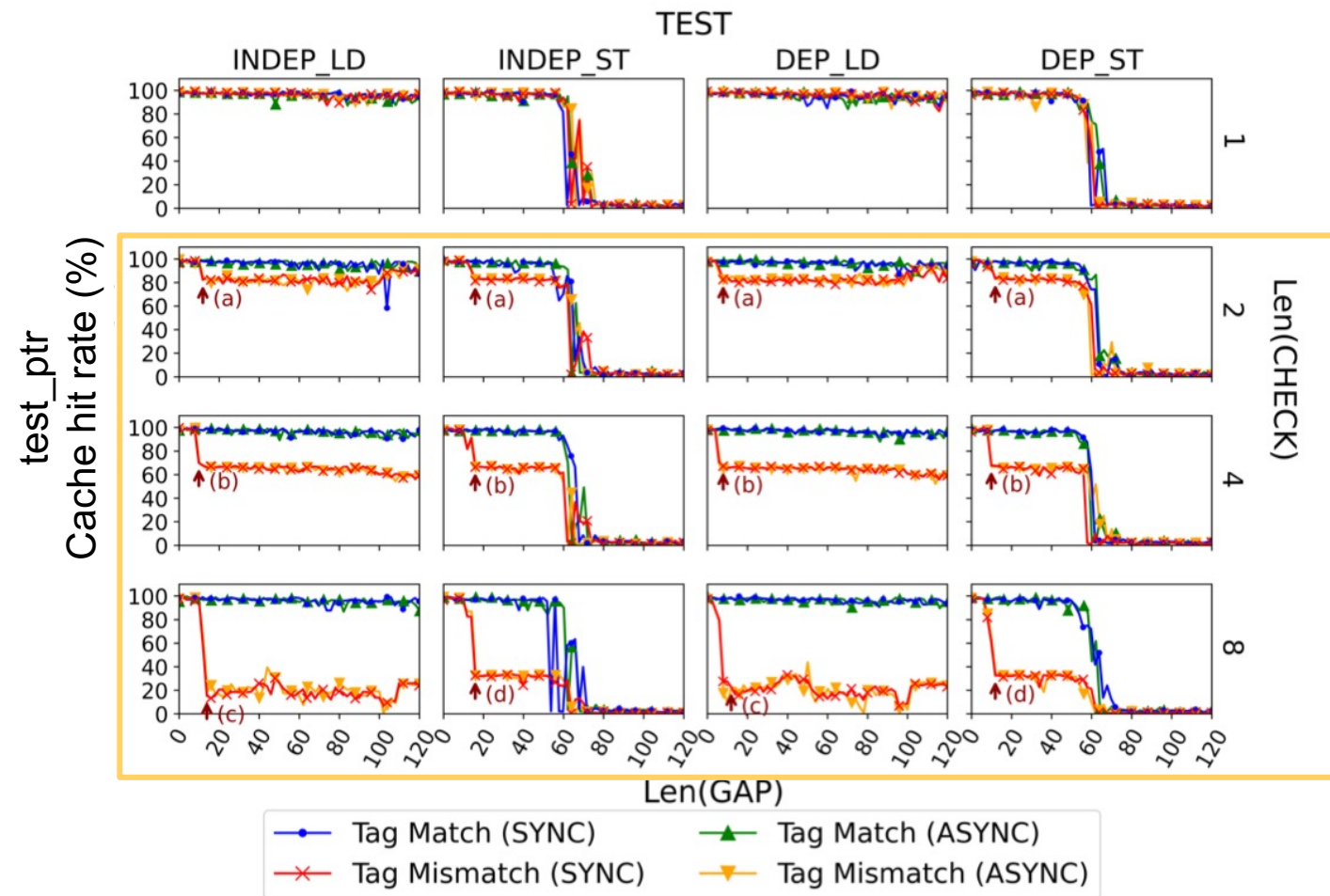
- Orthogonally found one of our tag leakage gadgets

```
    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 (If - 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 (lf - TEST)

→ test_ptr is *periodically not cached*.

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

TikTag-v1: Multiple Loads + Gap

```

if (cond) {
  // CHECK: 2+ loads
  *check_ptr;
  *check_ptr;
  ...
  // GAP: 10+ cycles
  test_ptr = test_ptr | 0;
  ...
  // TEST: dep/indep
  *test_ptr;
}
    
```

Q1. Why 2+ loads?

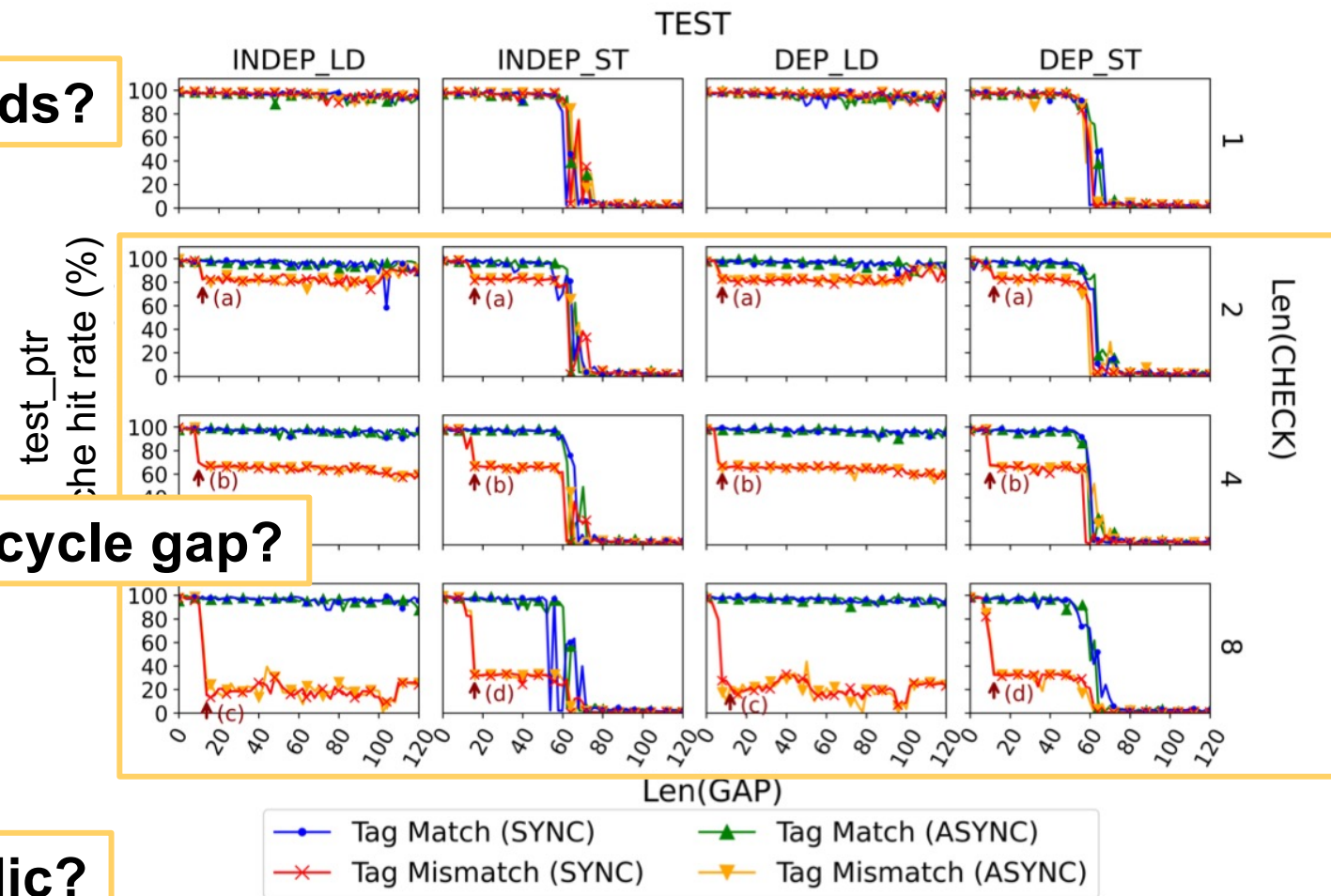
Tag Check Fault

Tag Check Fault

Q2. Why a 10+ cycle gap?

Q3. Why periodic?

Not Accessed



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

→ test_ptr is *periodically not cached*.

→ more tag check faults resulted in the shorter period

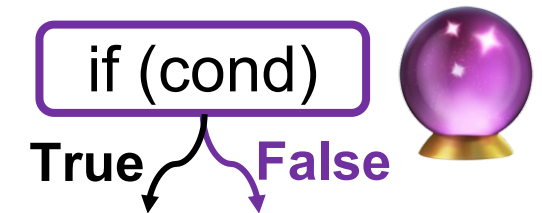
TikTag-v1: Root Cause Analysis

The CPU speculatively accesses memory by:

1) Speculative Execution (SE)

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

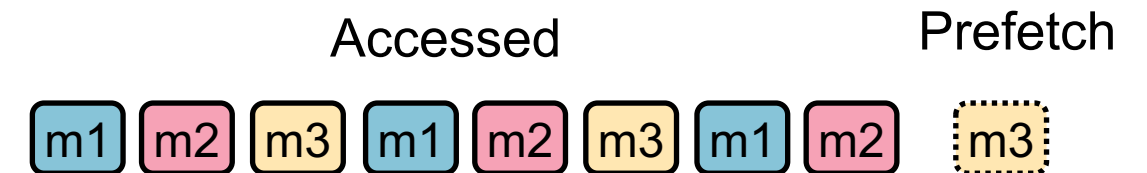
→ Hypothesis 1: **SE** stops on **tag check faults**



2) Hardware Data Prefetching (DP)

: Prefetch based on **memory access history**

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



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

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**

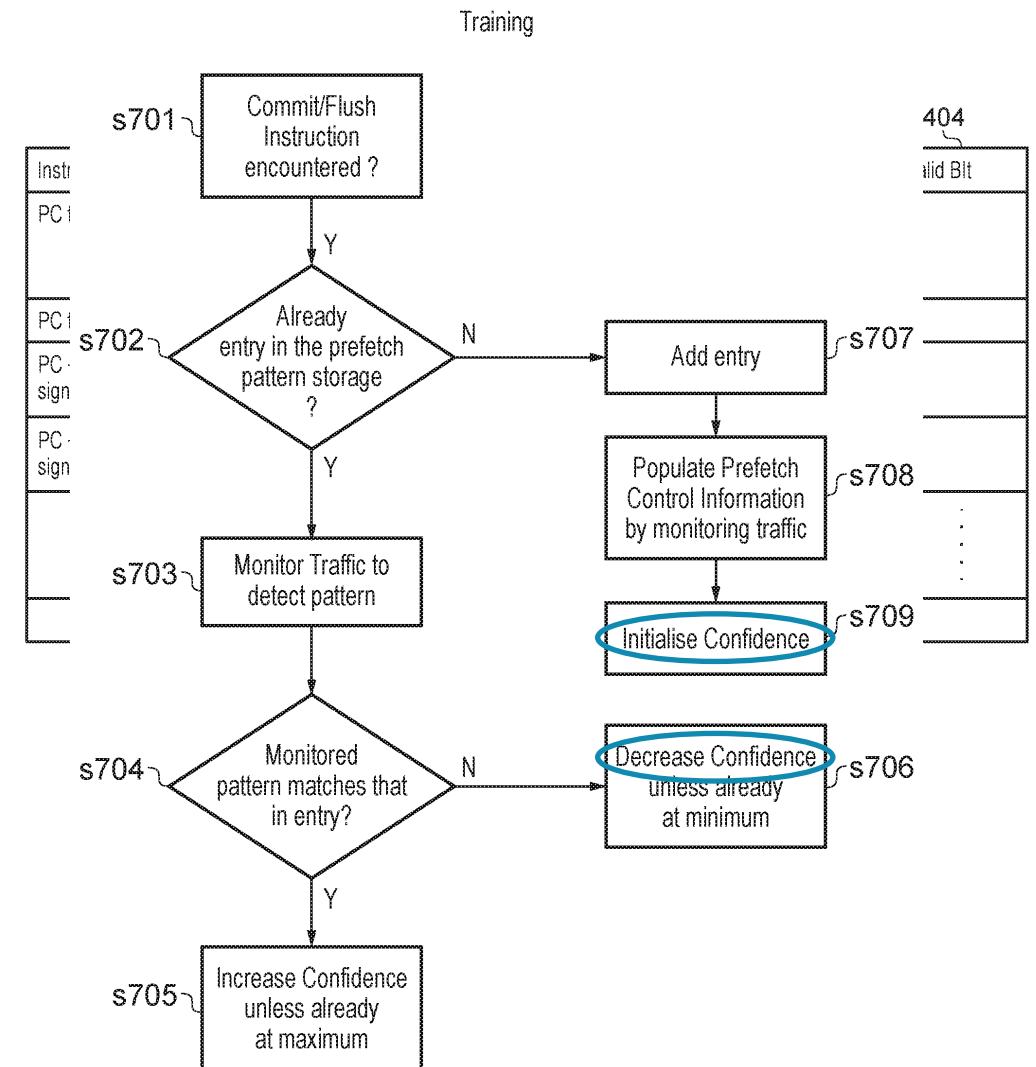
(56) **References Cited**
U.S. PATENT DOCUMENTS

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;
  val = *check_ptr;
  // TEST: dependent load/store
  *(test_ptr+val);
}

```

Tag Check Fault

Tag Check Fault

No Access

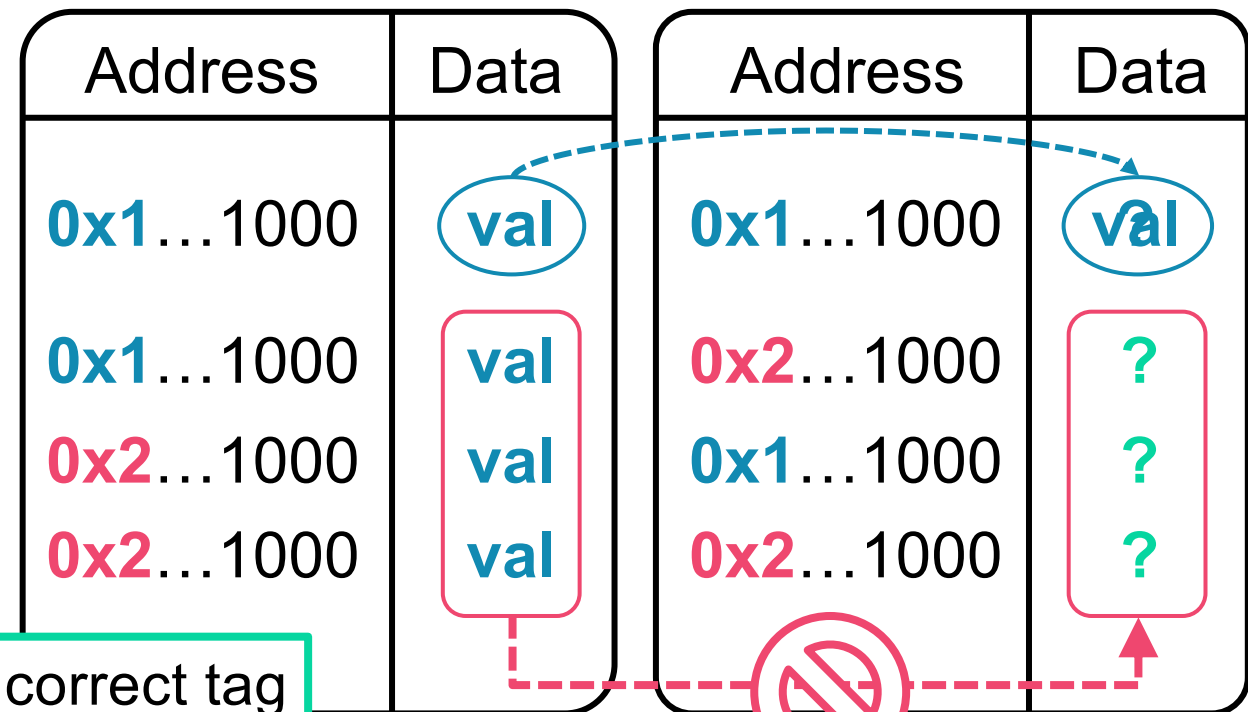
within 5 assembly instructions

Suspected root cause

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

Store Buffer

Load Buffer



0x1: correct tag
0x2: wrong tag

Roadmap

ARM Memory Tagging Extension

arm



Cache Side-Channel

Cache



Speculative Execution

if (cond)
True False



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



Chrome Renderer process



V8 JavaScript Engine

JavaScript

V8 Sandbox



Blink Rendering Engine

HTML

CSS



Third-party libraries

Potential
Memory
Corruption

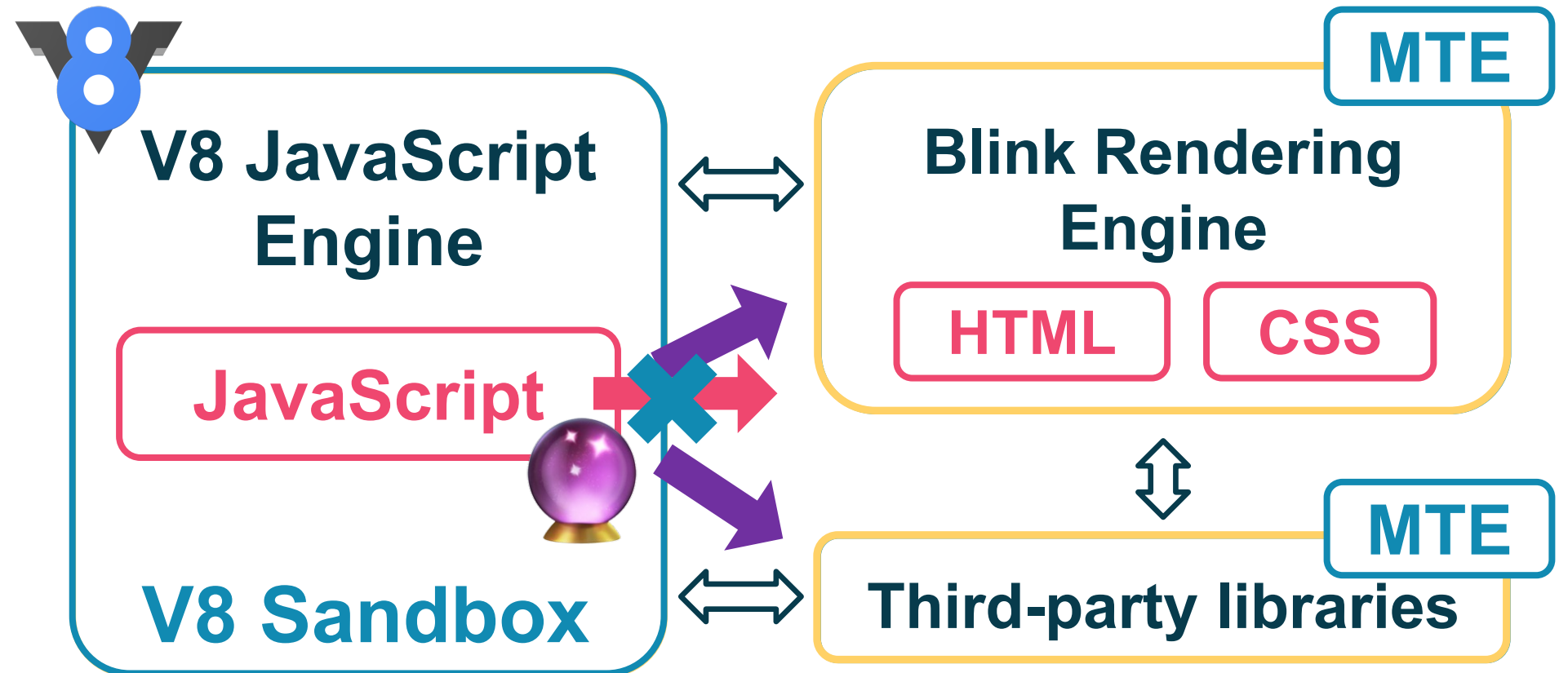
Attacker-
provided

Protected

Google Chrome Threat Model



Chrome Renderer process



Potential
Memory
Corruption

Attacker-
provided

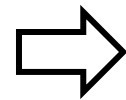
Protected

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 (64-bit out-of-bounds idx < arr.Length) 35-bit val = elements[access arbitrary 64-bit addridx];  
else val = undefined;
```

- 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) {  
    idx = AddrToldx(tag, target);  
    if (cond) {  
      check[idx] = val;  
      val = check[idx];  
      x = test[val];  
    }  
    time[tag] = Measure(test[val]);  
  }  
  return time.indexOf(min(time));  
}
```

← Iterate all tag values
← out-of-bounds index

check[idx]
Valid tag Invalid tag

Tag Leakage Gadget




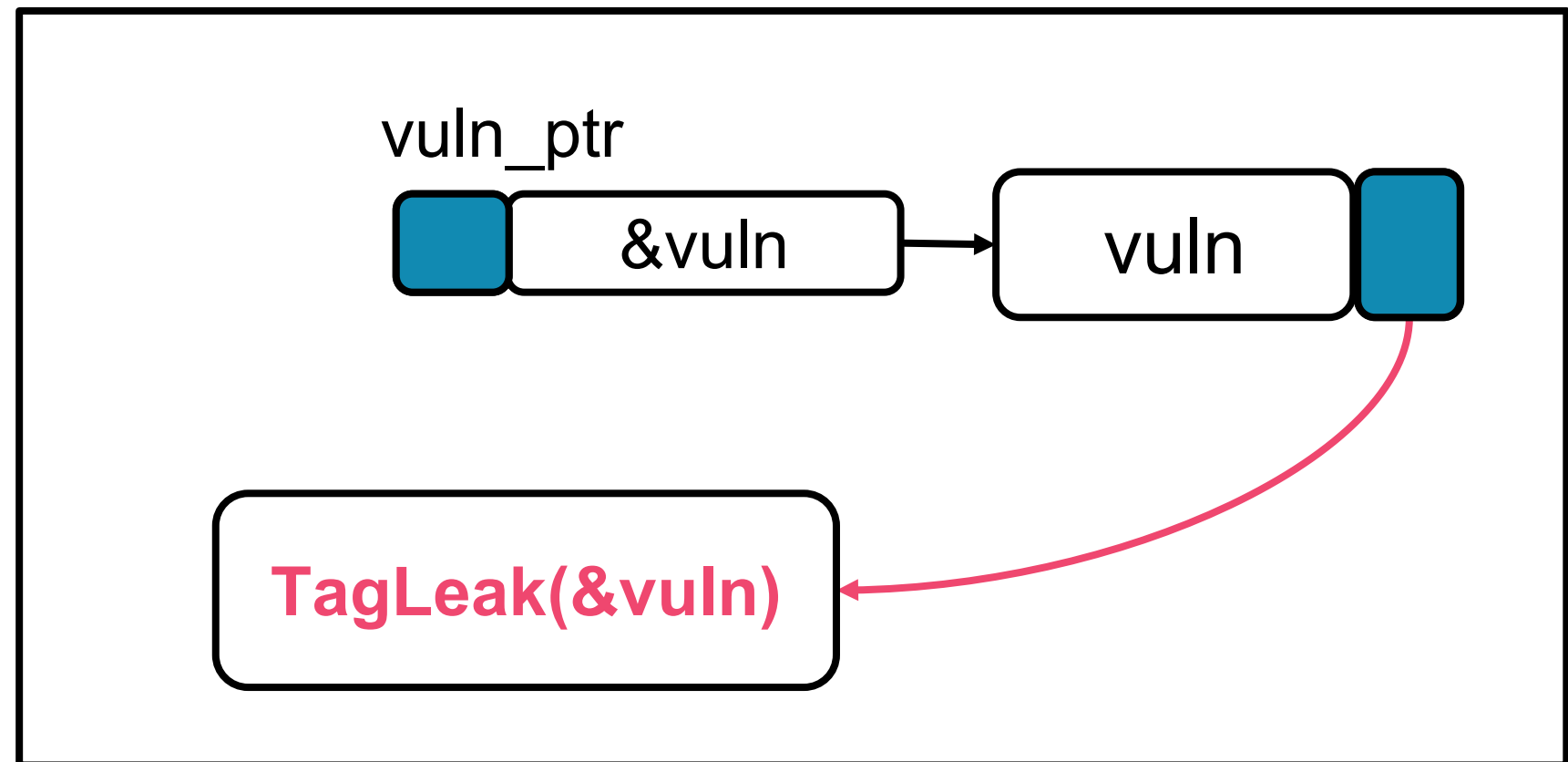
Valid tag	Invalid tag
No fault	Tag Check Fault
No fault	Tag Check Fault
Access	No Access
Fast	Slow

Tag Leaked!

1. Leak MTE Tag of vulnerable object



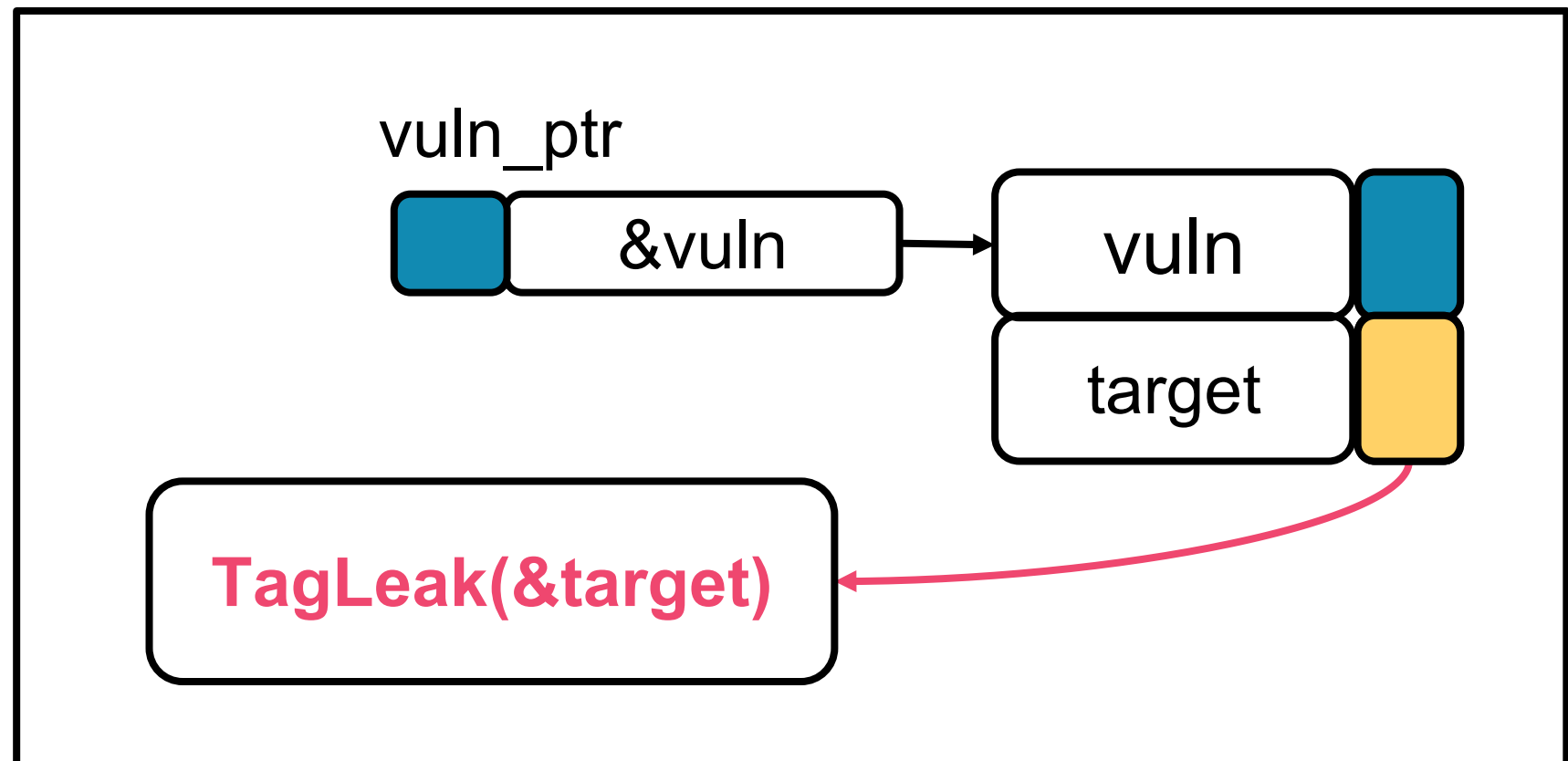
vuln.tag = 



2. Leak MTE Tag of target object




vuln.tag = 
target.tag = 



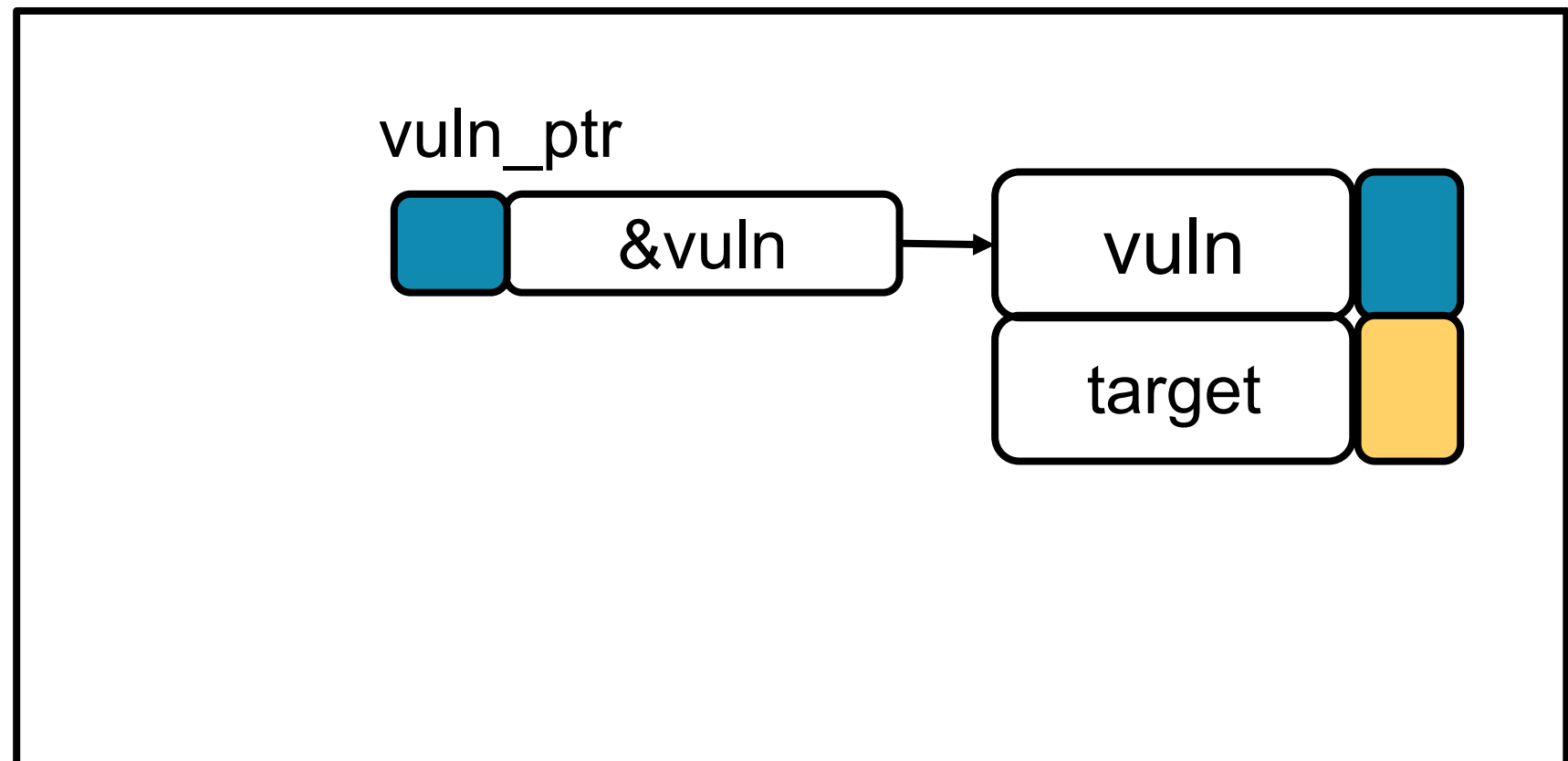
3. Reallocate target on tag mismatch



vuln.tag = 


target.tag = 

vuln.tag != target.tag

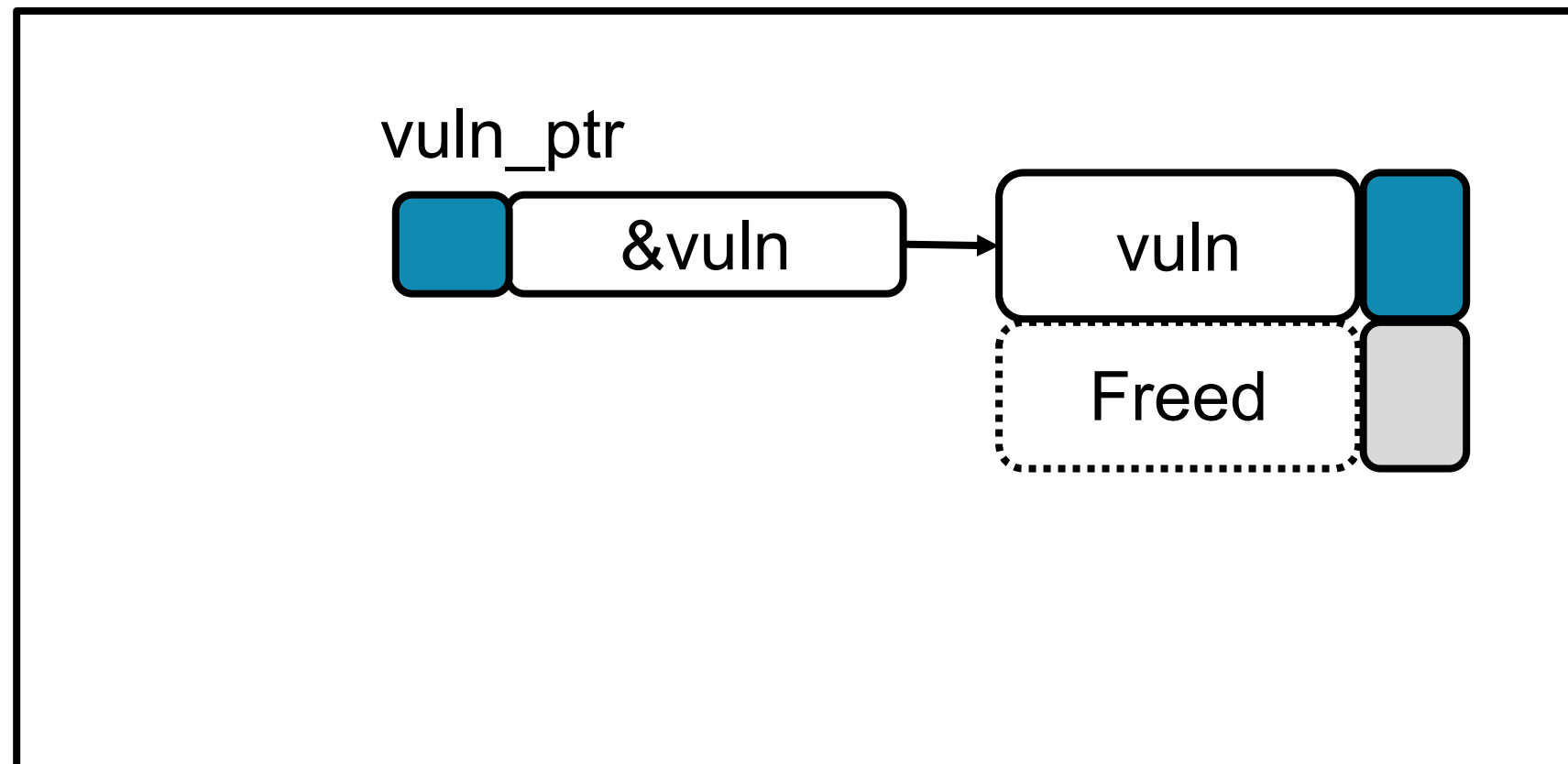


3. Reallocate target on tag mismatch




vuln.tag = 

Free(target);

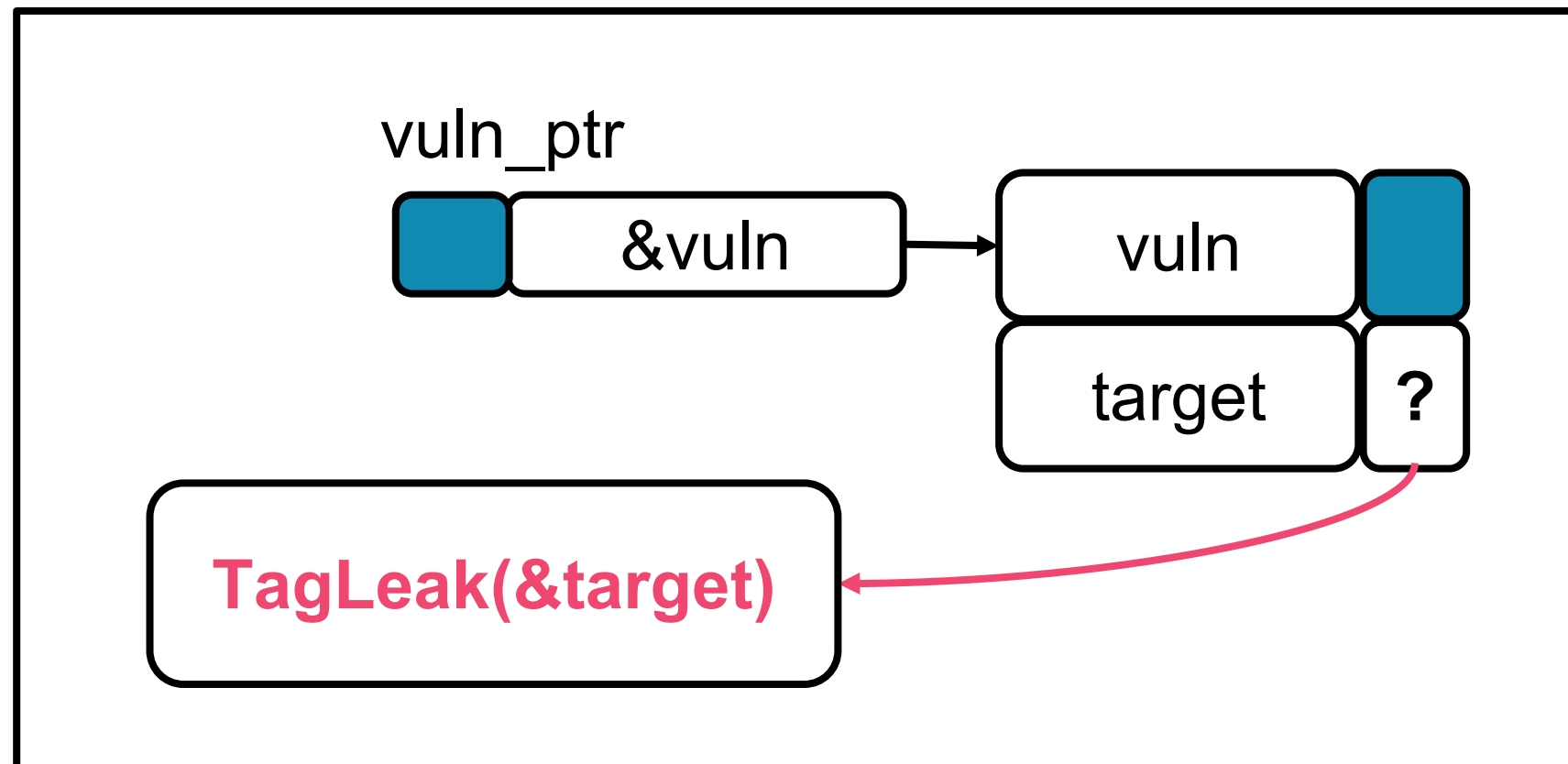


3. Reallocate target on tag mismatch



vuln.tag = 



Free(target);
Alloc(target);

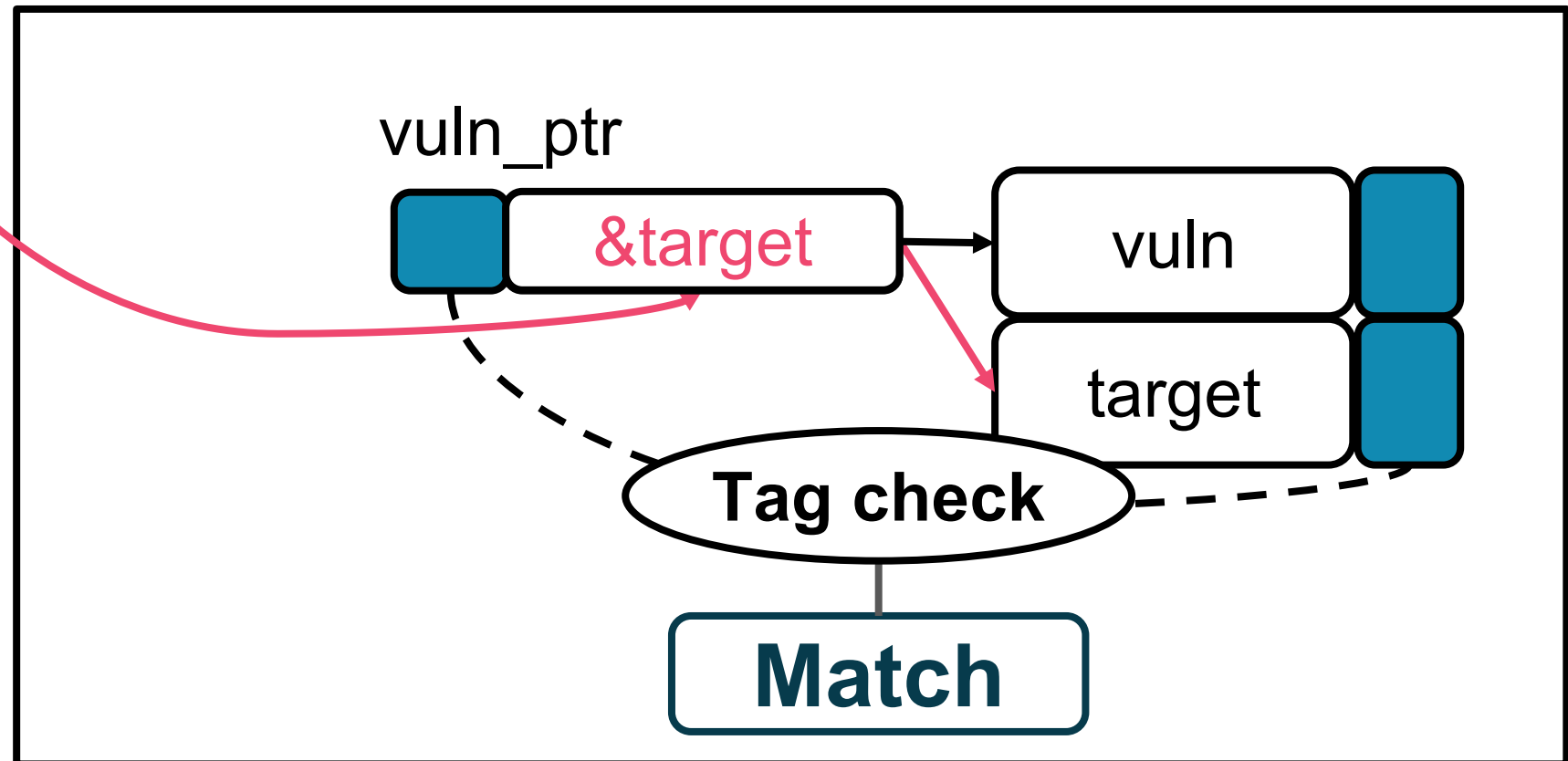


4. Trigger vulnerability on tag match

Trigger out-of-bounds access

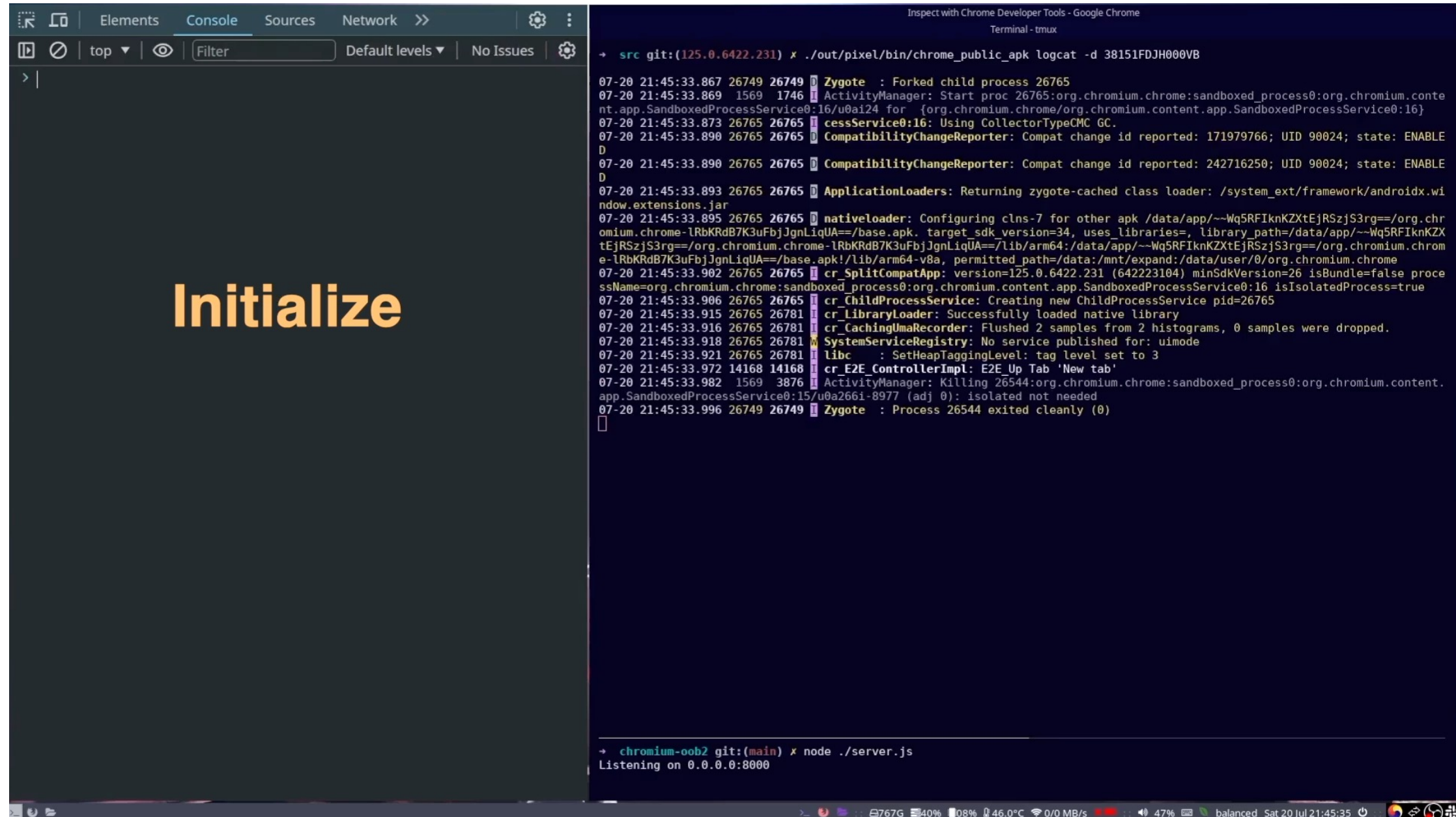


vuln.tag = 
target.tag = 
vuln.tag == target.tag



CVE-2023-5217 Chrome libvpx heap overflow

Original Memory Corruption → Attack Fail



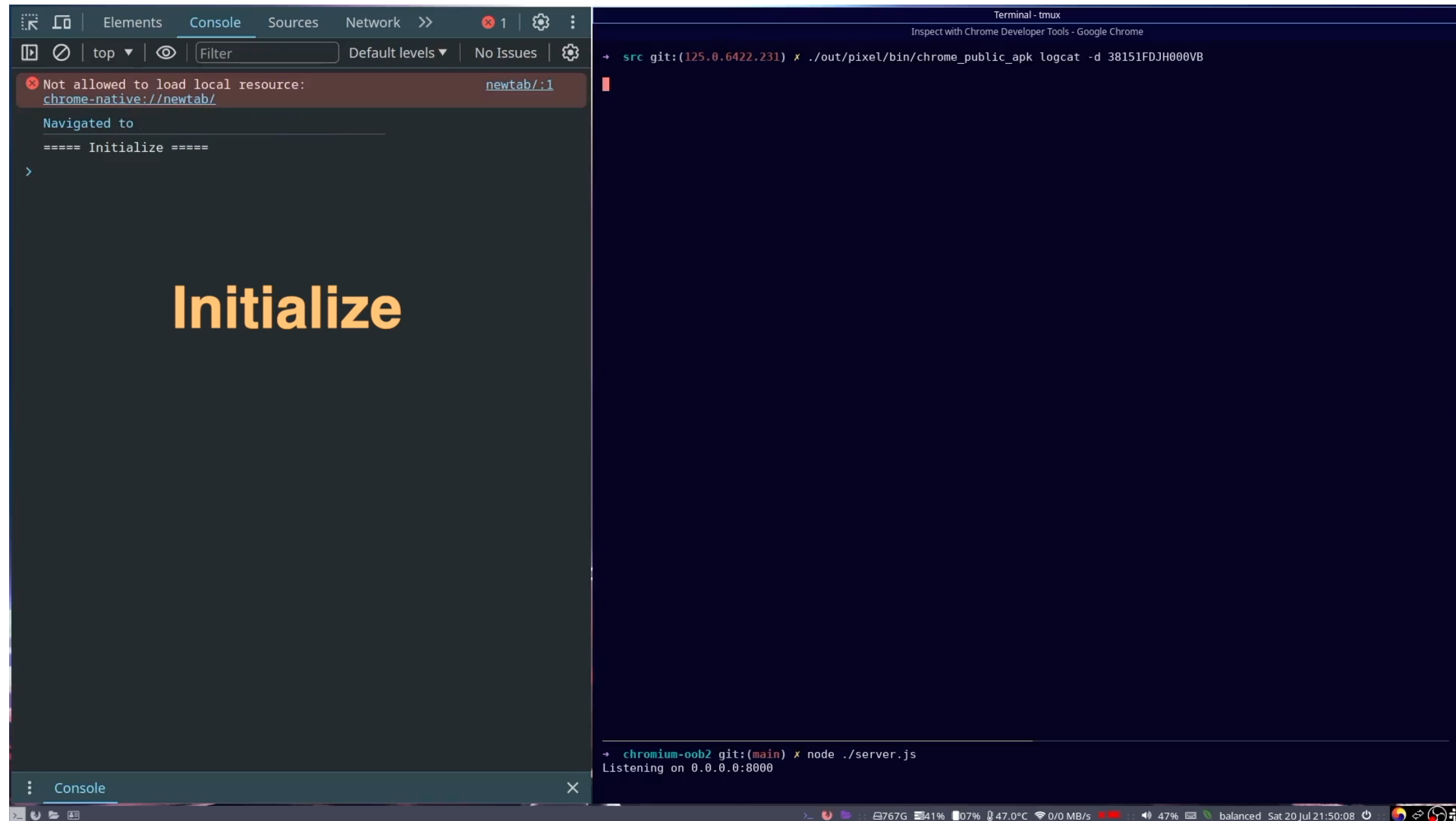
The image shows a screenshot of a Chrome DevTools console and terminal window. The console on the left is empty, and the terminal on the right displays logcat output for a Chrome process. The word "Initialize" is written in large orange text over the console area.

```
src git:(125.0.6422.231) x ./out/pixel/bin/chrome_public_apk logcat -d 38151FDJH000VB
07-20 21:45:33.867 26749 26749 D Zygot : Forked child process 26765
07-20 21:45:33.869 1569 1746 I ActivityManager: Start proc 26765:org.chromium.chrome:sandboxed_process0:org.chromium.conte
nt.app.SandboxedProcessService0:16/u0ai24 for {org.chromium.chrome/org.chromium.content.app.SandboxedProcessService0:16}
07-20 21:45:33.873 26765 26765 I ccessService0:16: Using CollectorTypeCMC GC.
07-20 21:45:33.890 26765 26765 D CompatibilityChangeReporter: Compat change id reported: 171979766; UID 90024; state: ENABLE
D
07-20 21:45:33.890 26765 26765 D CompatibilityChangeReporter: Compat change id reported: 242716250; UID 90024; state: ENABLE
D
07-20 21:45:33.893 26765 26765 D ApplicationLoaders: Returning zygote-cached class loader: /system_ext/framework/androidx.wi
ndow.extensions.jar
07-20 21:45:33.895 26765 26765 D nativeloader: Configuring clns-7 for other apk /data/app/~Wq5RFIknKZXtEjRSzjS3rg==/org.chr
omium.chrome-lRbKRdB7K3uFbjJgnLiqUA==/base.apk. target_sdk_version=34, uses_libraries=, library_path=/data/app/~Wq5RFIknKZX
tEjRSzjS3rg==/org.chromium.chrome-lRbKRdB7K3uFbjJgnLiqUA==/lib/arm64:/data/app/~Wq5RFIknKZXtEjRSzjS3rg==/org.chromium.chrom
e-lRbKRdB7K3uFbjJgnLiqUA==/base.apk!/lib/arm64-v8a, permitted_path=/data:/mnt/expand:/data/user/0/org.chromium.chrome
07-20 21:45:33.902 26765 26765 I cr_SplitCompatApp: version=125.0.6422.231 (642223104) minSdkVersion=26 isBundle=false proce
ssName=org.chromium.chrome:sandboxed_process0:org.chromium.content.app.SandboxedProcessService0:16 isIsolatedProcess=true
07-20 21:45:33.906 26765 26765 I cr_ChildProcessService: Creating new ChildProcessService pid=26765
07-20 21:45:33.915 26765 26781 I cr_LibraryLoader: Successfully loaded native library
07-20 21:45:33.916 26765 26781 I cr_CachingUmaRecorder: Flushed 2 samples from 2 histograms, 0 samples were dropped.
07-20 21:45:33.918 26765 26781 W SystemServiceRegistry: No service published for: uimode
07-20 21:45:33.921 26765 26781 I libc : SetHeapTaggingLevel: tag level set to 3
07-20 21:45:33.972 14168 14168 I cr_E2E_ControllerImpl: E2E_Up Tab 'New tab'
07-20 21:45:33.982 1569 3876 I ActivityManager: Killing 26544:org.chromium.chrome:sandboxed_process0:org.chromium.content.
app.SandboxedProcessService0:15/u0a2661-8977 (adj 0): isolated not needed
07-20 21:45:33.996 26749 26749 I Zygot : Process 26544 exited cleanly (0)
[]

chromium-oob2 git:(main) x node ./server.js
Listening on 0.0.0.0:8000
```

CVE-2023-5217 Chrome libvpx heap overflow

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

The screenshot shows a Chromium issue report for 'Security: V8 Spectre-v1'. The issue is categorized as 'Vulnerability' with a priority of 'P3' and a severity of 'S4'. The status is 'Won't fix (Obsolete)'. The reporter is 'kimjuhi96@snu.ac.kr'. The issue was created on Dec 14, 2023, at 11:10AM. The description includes a template for reporting security bugs and a link to the Chromium FAQ. The vulnerability details section is partially visible, starting with '1. Overview'.

sa...@google.com <sa...@google.com> #12 Dec 15, 2023 07:10PM

From the Objective section of the V8 Sandbox: "Build an in-process sandbox for V8 to prevent an attacker who successfully exploited a V8 vulnerability, and thus is able to corrupt objects inside the V8 heap, from corrupting other memory in the process and thus from executing arbitrary code" I think this is really it. We simply want to prevent memory corruption outside of the sandbox. If we make reading memory outside of the sandbox harder, that's a nice side-effect, but we don't design/implement anything specifically to prevent reads. Similarly, if we make it harder to exploit certain Blink bugs (e.g. ArrayBuffer UaFs or OOB accesses), that's also nice, but is also just by accident.

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*

The screenshot shows a Chromium issue report for "Security: V8 Speculative MTE Tag Leakage". The issue is categorized as "Vulnerability" with a priority of "P3" and a status of "Won't fix (Obsolete)". The reporter is kimjuhi96@snu.ac.kr. The issue was created on Dec 14, 2023, at 11:17 AM. The description details an ARM Memory Tagging Extension (MTE) tag leakage vulnerability in V8, which bypasses the MTE-backed object-granule memory protection of Chrome on recent ARM devices (e.g., Pixel 8). The vulnerability is triggered by malicious JavaScript when optimized into native arm64 instructions.

VULNERABILITY DETAILS

1. Overview

We have identified an ARM Memory Tagging Extension (MTE) tag leakage vulnerability in V8, which bypasses the MTE-backed object-granule memory protection of Chrome on recent ARM devices (e.g., Pixel 8).

Heap objects, allocated through either PartitionAlloc (excluding v8 cage memory) or Android MTE. In this configuration, both PartitionAlloc and Scudo assign a random tag to an allocated object. Upon deallocation, PartitionAlloc increments the tag, and Scudo uses use-after-free.

With a 15/16 chance, two spatially or temporally disjoint objects are tagged with different tags, reducing the probability of memory corruption attacks to 1/16, raising the bar against memory corruption attacks.

Unfortunately, the tag leakage primitive we report here bypasses the MTE protection.

This primitive is triggered by malicious JavaScript when optimized into native arm64 instructions.

Comments: A comment from sr...@google.com (Dec 15, 2023 05:32PM) states: "Status: Won't Fix (Obsolete). Hey, thanks for the report, this is really cool research! However, this is a wontfix for us. We currently don't consider MTE to be a security boundary in the renderer and one of the reasons is exactly side-channel issues like this."

Hardware Mitigations

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



1. Always Block (Bad)

No Fault

Tag Check Fault


No Access

No Access

→ **Performance sacrifice**

No Speculative Execution & No Data Prefetching

Hardware Mitigations

if (cond) { 

2. Always Allos (Better)

// Check
Access(check_ptr);

No Fault

Tag Check Fault

// Test
Access(test_ptr);

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