MTEscape: Bypassing Asynchronous Kernel MTE via Conventional Memory Corruption Vulnerabilities

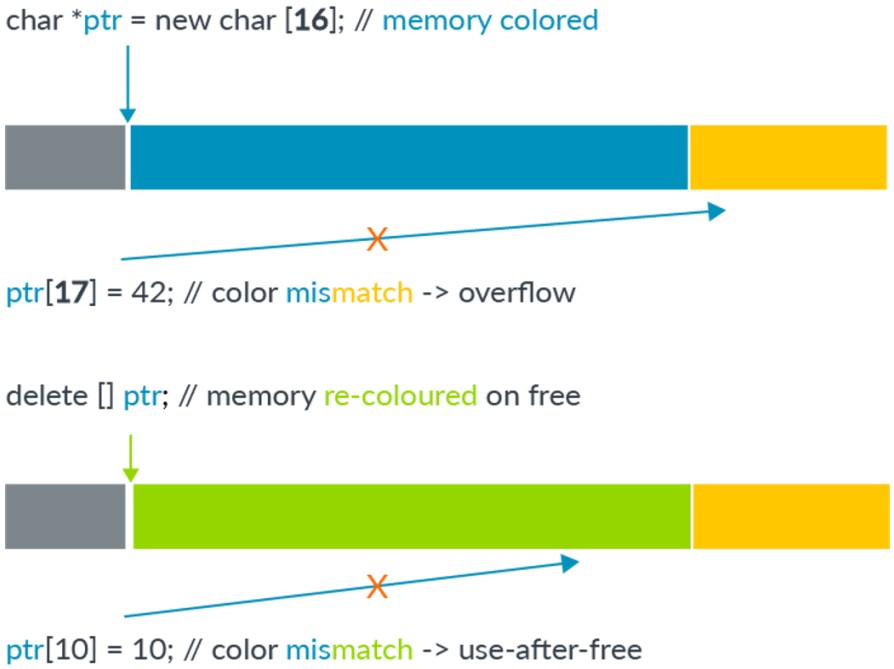
Kyeongmin Kim & Insu Yun @ KAIST Hacking Lab



ACM CCS 2026

Background: Memory Tagging Extension

How MTE Works



MTE compares pointer and memory tags to detect invalid memory accesses.

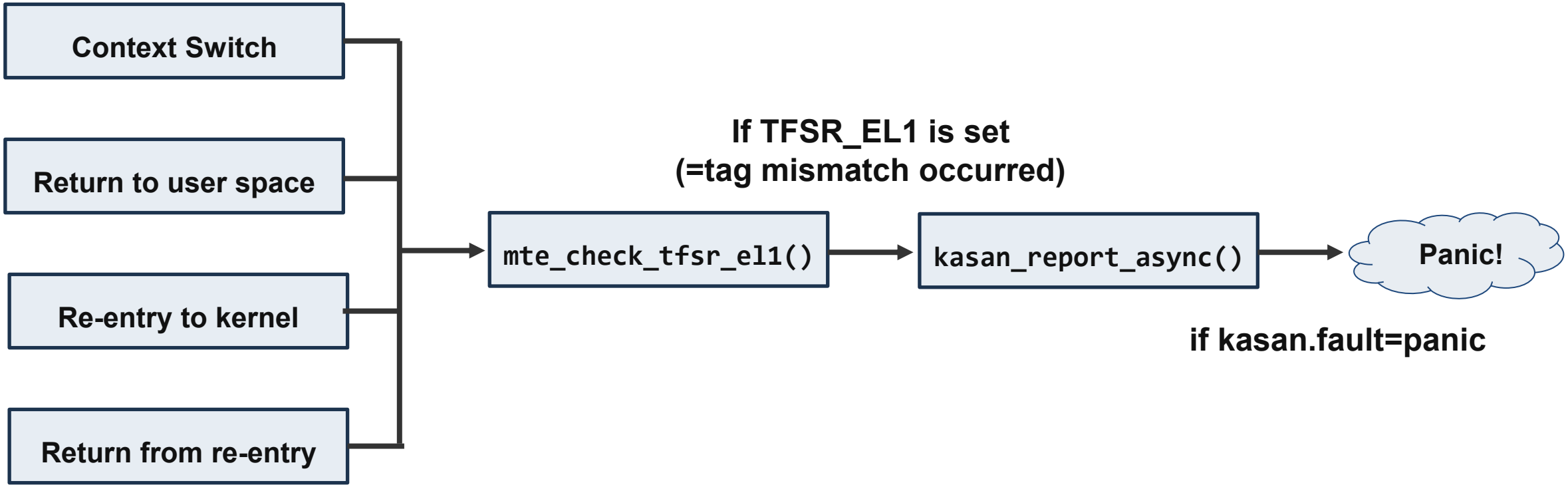
Operating Modes

Mode	Avg. overhead	Worst overhead
Synchronous	56%	564%
Asynchronous	8%	21%

Noh et al. (2026), SPEC CPU2006, Pixel 8 performance core

Synchronous: immediate fault handling.
Asynchronous: delayed handling, less overhead.

Asynchronous Fault Handling



Delayed tag-fault handling creates a short window for exploitation.

Challenges

1. **Suppressing kernel panic**

   Root privileges do not clear a pending tag fault.

2. **Limited vulnerability pivoting**

   Only a short window remains after the initial corruption.

3. **Single-attempt race**

   If the race fails, the kernel panics before another attempt.

Our Approach

1. **Suppressing kernel panic → MTE kill switches**

   Corrupt global variables to skip or block tag-fault handling.

2. **Limited pivoting → Dereference-write objects (DWO)**

   Use a dereference-write object for a single-pivot write to a kill switch.

3. **Single-attempt race → Race Stabilization**

   Use separate techniques for OOB writes and UAFs.

MTE Kill Switches

Example: kasan_flags

```
542  void kasan_report_async(void)
543  {
544          unsigned long flags;
545
546          /*
547           * Do not check report_suppressed_sw(), as
548           * kasan_disable/enable_current() critical sections do not affect
549           * Hardware Tag-Based KASAN.
550           */
551          if (unlikely(!report_enabled()))
552                  return;
553
554          start_report(&flags, false);
555          pr_err("BUG: KASAN: invalid-access₩n");
556          pr_err("Asynchronous fault: no details available₩n");
557          pr_err("₩n");
558          dump_stack_lvl(KERN_ERR);
559          end_report(&flags, NULL);
560  }
```

```
125  static bool report_enabled(void)
126  {
127          if (test_bit(KASAN_BIT_MULTI_SHOT, &kasan_flags))
128                  return true;
129          return !test_and_set_bit(KASAN_BIT_REPORTED, &kasan_flags);
130  }
```

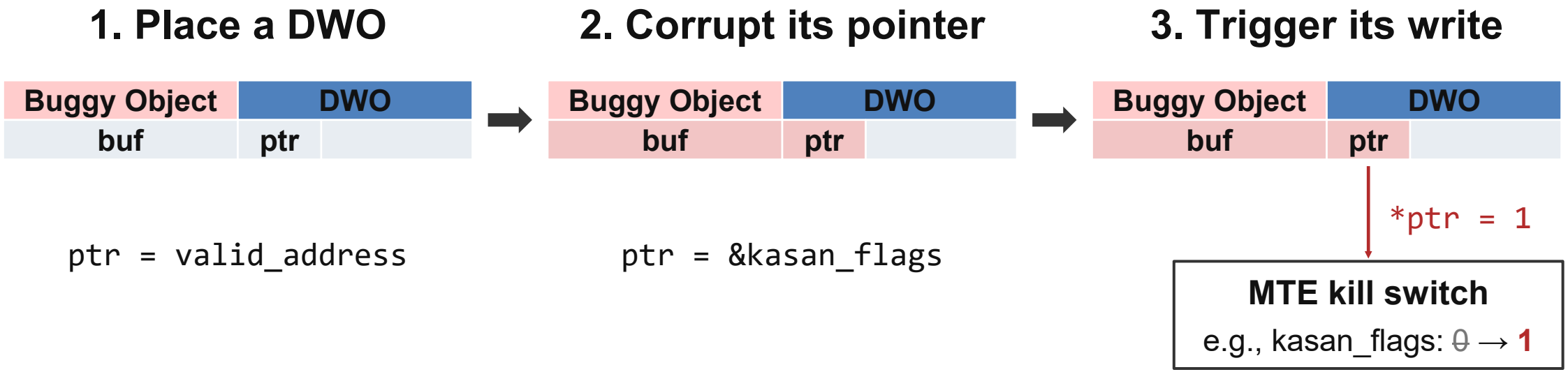
Overwriting kasan_flags with 1 suppresses fault reporting and prevents panic.

Kill-switch variables

Variable	Condition
kasan_flags	& 0x3 = 1
report_lock	≠ 0
console_owner_lock	≠ 0
console_sem.lock	≠ 0
printk_cpu_sync_owner	≠ −1 and ≠ CPU ID
panic_cpu	≠ −1 and ≠ CPU ID

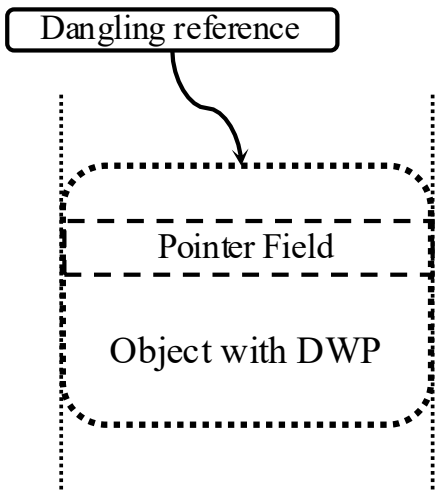
An ARM64 KASLR weakness makes kill-switch addresses predictable.

Single-Pivot Write via DWO

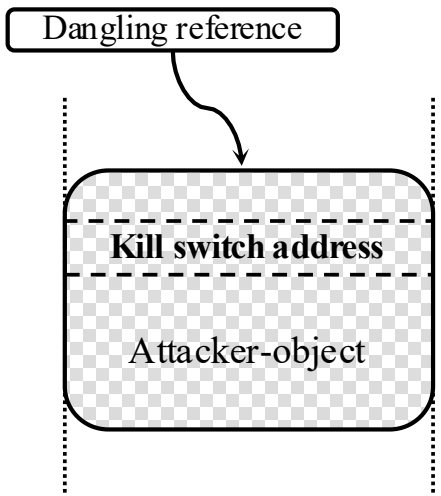


Exploiting UAF

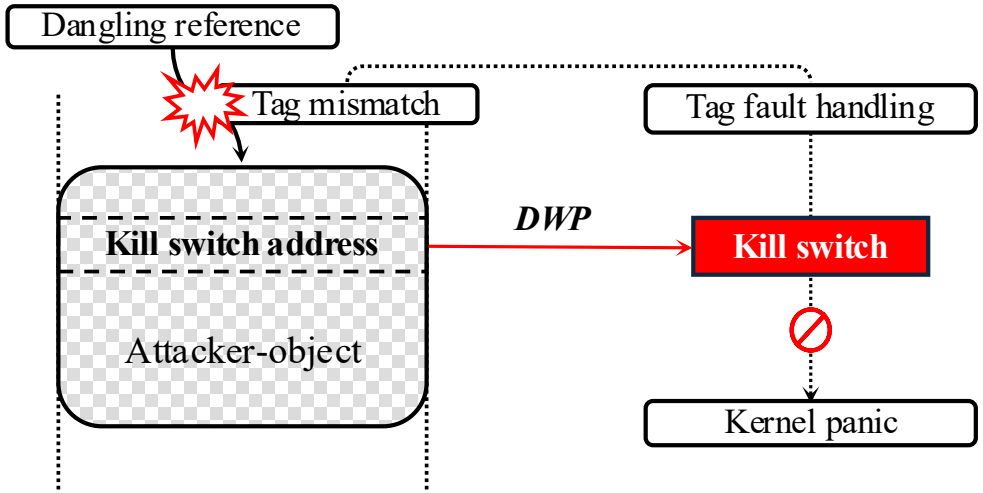
Requirement: the dangling object provides a dereference-write primitive.



① Free an object with DWP while holding a reference



② Overwrite the pointer field through reclaiming

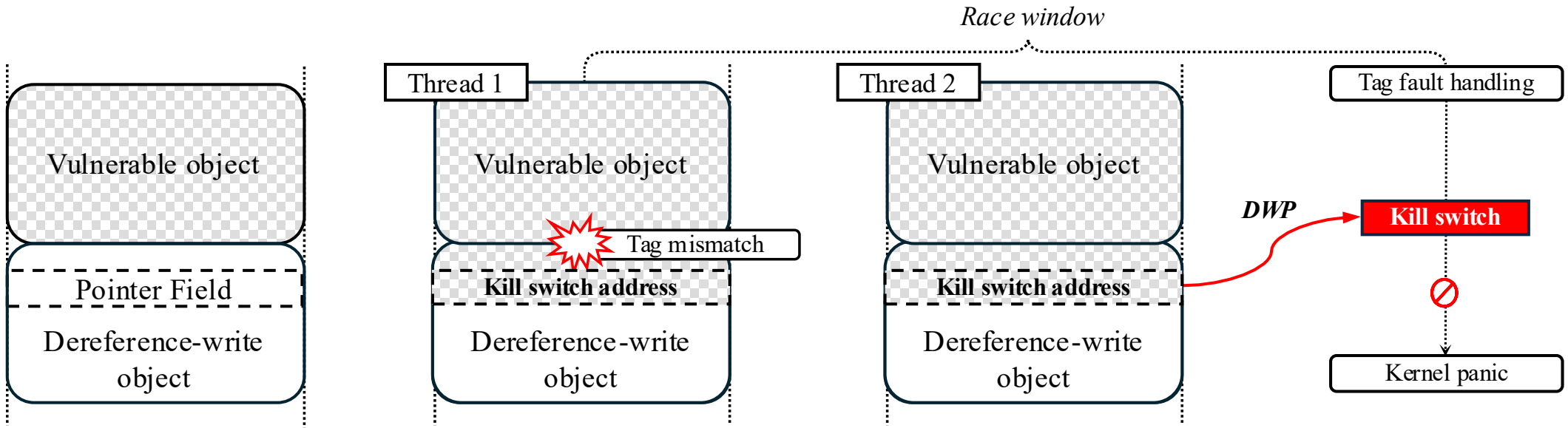


③ Trigger DWP to overwrite kill switch

Result: Tag fault handling is bypassed

The faulting access is immediately followed by the kill-switch write.

Exploiting OOB Write



① Allocate a DWO adjacent to the vulnerable object

② Overwrite the pointer field through OOB write

③ Trigger DWP from another thread to overwrite kill switch

Result: Tag fault handling is bypassed

Race window: after pointer corruption, before tag-fault handling.

Race Stabilization: CARDSHARK

Han et al., USENIX Security ’24

1. Synchronize the start

Coordinator

```
while (!t1_ready || !t2_ready);
synced = 1;
```

T1 · OOB

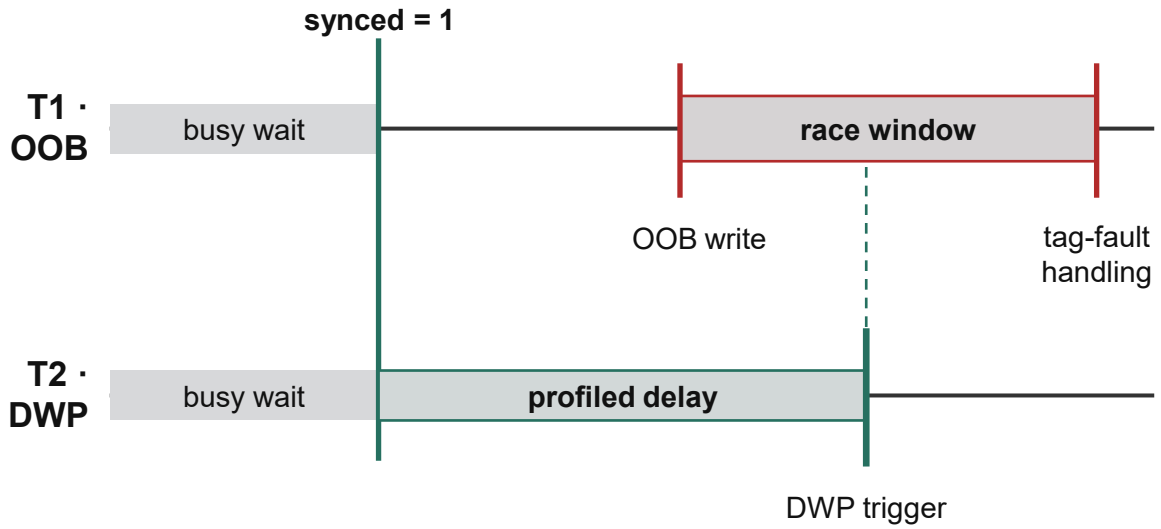
```
t1_ready = 1;
while (!synced);
```

T2 · DWP

```
t2_ready = 1;
while (!synced);
```

Both workers wait for release.

2. Align the DWP



- Busy-wait synchronization reduces start-time variation.
- A profiled delay compensates for path-time differences.

Race Stabilization: Cache Training

```
8  void* thread_OOB(void* arg) {
9      pin_cpu(1);
10
11     if (config == MTESCAPE)
12         for (int i = 0; i < CACHE_TRAINING_COUNT; i++)
13             syscall_OOB(dummy_vuln, INBOUNDS_LEN);
14
15     if (config != NAIVE) {
16         t1_ready = 1;
17         while (!synced);
18     }
19     syscall_OOB(target_vuln, OOB_LEN);
20 }
```

```
22 void* thread_DWP(void* arg) {
23     pin_cpu(2);
24
25     if (config == MTESCAPE)
26         for (int i = 0; i < CACHE_TRAINING_COUNT; i++)
27             syscall_DWP(dummy_dwo);
28
29     if (config != NAIVE) {
30         t2_ready = 1;
31         while (!synced);
32         delay(config == CARDSHARK ? DELAY_CARDSHARK : DELAY_MTESCAPE);
33     }
34     syscall_DWP(target_dwo);
35 }
```

- Run both paths with non-faulting inputs before the race.
- Warming the cache and branch predictor reduces timing variation.

Evaluation: Synthetic Vulnerabilities

Evaluation setup: Radxa Orion O6, Linux v6.16

Cache size (bytes)	OOB write		UAF	
	Idle	Busy	Idle	Busy
64	93%	12%	100%	100%
128	92%	9%	100%	100%
192	94%	13%	100%	100%
256	93%	10%	100%	100%
512	98%	13%	100%	99%
1024	96%	11%	100%	100%
2048	93%	12%	100%	100%
4096	96%	13%	100%	99%
Average	**94.38%**	**11.63%**	**100%**	**99.75%**

Busy: CPU, I/O, and memory load from stress-ng.

Evaluation: Real-World Vulnerabilities

5 OOB writes	5 UAFs
CVE-2021-43267	CVE-2021-3715
CVE-2022-0185	CVE-2022-2588
CVE-2022-34918	CVE-2023-4623
CVE-2024-36016	CVE-2024-58240
CVE-2024-53141	CVE-2025-21702

MTEscape bypassed asynchronous MTE on all 10 real-world CVEs.

Evaluation: Race Stabilization

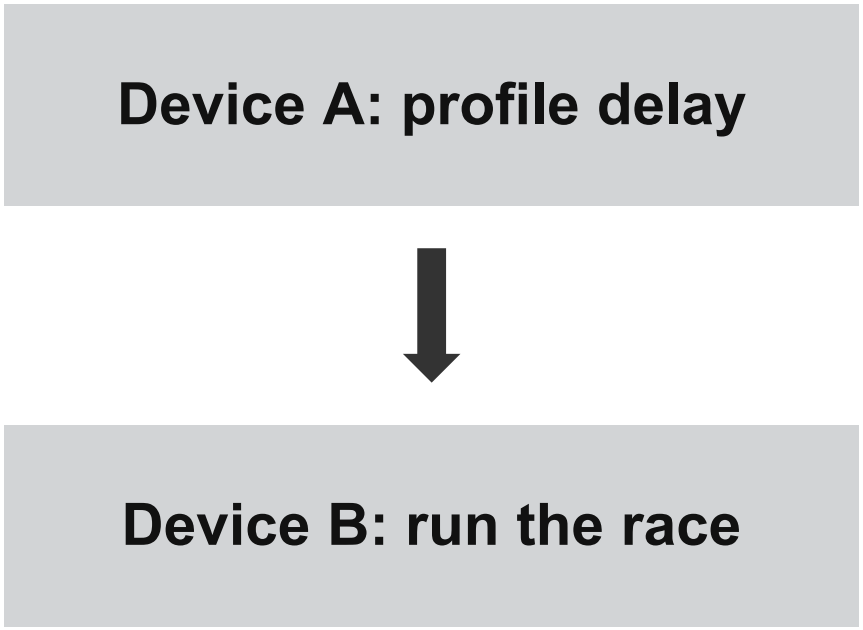
Method	CVE-2021-43267	CVE-2022-0185	CVE-2022-34918	CVE-2024-36016	CVE-2024-53141
Naive	0.0%	24.6%	0.0%	29.0%	82.6%
CARDSHARK	11.4%	99.3%	99.8%	97.8%	95.5%
MTEscape	99.3%	99.2%	99.6%	98.3%	98.6%

1,000 trials per CVE

- Race success rate: 98.3% to 99.6%.
- **CVE-2021-43267**: 11.4% with CARDSHARK, 99.3% with MTEscape.

Evaluation: Cross-Device Validation

Evaluation setup: Two Orange Pi 6 Plus boards of the same model



OOB CVE	Success
CVE-2021-43267	100.0%
CVE-2022-0185	99.9%
CVE-2022-34918	100.0%
CVE-2024-36016	93.4%
CVE-2024-53141	99.6%

≥93.4% race success

Mitigation

- **Remove kill-switch behavior**

  In panic mode, always call panic(), regardless of kasan_flags.

- **Randomize indispensable locks**

  Use a random locked value that attackers cannot predict.

Measured performance change

Suite	Benchmarks	Overhead range
Phoronix Test Suite	Applications	−1.79% to +0.10%
LMbench	System primitives	−1.88% to +1.59%

Summary

- Conventional OOB write and UAF vulnerabilities can bypass asynchronous kernel MTE.
- MTEscape bypassed MTE protection on 10 real-world CVEs and achieved at least 98.3% race success across the evaluated OOB cases.
- Removing kill-switch behavior and randomizing indispensable locks prevents the attack with negligible overhead.

Thank you

Source: https://github.com/kaist-hacking/MTEscape

Contact: newjeans@kaist.ac.kr

