MTEscape: Bypassing Asynchronous Kernel MTE via Conventional Memory Corruption Vulnerabilities

Kyeongmin Kim
KAIST
Daejeon, Republic of Korea
newjeans@kaist.ac.kr

Insu Yun
KAIST
Daejeon, Republic of Korea
insuyun@kaist.ac.kr

Abstract

ARM Memory Tagging Extension (MTE) employs hardware-enforced tagging to mitigate memory corruption vulnerabilities. By detecting prevalent vulnerabilities such as buffer overflows and use-after-free, MTE significantly enhances memory safety. Despite its deployment in modern kernels, security analysis of kernel MTE remains limited. While prior work has proposed bypass techniques, they are constrained to specialized primitives, leaving practical kernel MTE bypasses largely unexplored.

In this paper, we present MTEscape, a novel technique that bypasses asynchronous kernel MTE by exploiting its design weakness. While MTE is intended to mitigate conventional memory corruption vulnerabilities, MTEscape leverages such vulnerabilities to construct reliable exploits. MTEscape exploits the race window in asynchronous kernel MTE, where tag checking is deferred until a context switch. Although this deferred checking is a known theoretical limitation, we are the first to demonstrate its exploitability in the Linux kernel. Within this race window, MTEscape corrupts the kill switch variables of MTE to disable tag fault handling. To overcome the tight temporal constraints, we leverage dereference-write objects for single-step vulnerability pivoting and employ race stabilization techniques to enhance exploit reliability. We evaluate MTEscape on synthetic vulnerabilities and 10 real-world kernel CVEs, successfully bypassing asynchronous MTE in all cases. We further propose a practical mitigation that prevents MTEscape with negligible performance overhead.

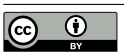
CCS Concepts

• **Security and privacy → Operating systems security**.

Keywords

OS Security, Kernel Exploitation, Memory Tagging Extension

ACM Reference Format:
Kyeongmin Kim and Insu Yun. 2026. MTEscape: Bypassing Asynchronous Kernel MTE via Conventional Memory Corruption Vulnerabilities. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3830454.3846722



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846722

1 Introduction

Memory corruption vulnerabilities remain among the most critical threats to modern software security [34]. Exploiting such flaws, an attacker can hijack control flow or tamper with security-critical state, ultimately executing arbitrary code or escalating privileges. Despite decades of software-based mitigations such as ASLR [38] and CFI [1], vulnerabilities such as buffer overflows and use-after-free remain widely exploited.

To address these threats, ARM introduced the Memory Tagging Extension (MTE) in the ARMv8.5-A architecture [3]. MTE detects spatial and temporal memory violations at runtime via hardware-enforced tag validation. By mitigating major vulnerability classes such as buffer overflows and use-after-free, MTE substantially reduces the attack surface for memory corruption exploits. Given its effectiveness, MTE support has been integrated into major allocators, including Chrome's PartitionAlloc [7], Android's Scudo [26], and the Linux kernel's SLUB allocator [8].

Since its integration into the Linux kernel, only few exploits have demonstrated kernel MTE bypass [32, 33, 47]. Each of these exploits leverages coprocessor vulnerabilities, where the coprocessor itself performs invalid writes. Since MTE does not check coprocessor accesses, these exploits bypass MTE protection. However, this approach does not generalize to non-coprocessor vulnerabilities, which constitute the majority of kernel vulnerabilities. TikTag [17] demonstrated MTE tag leakage via speculative execution, but its gadget in the Linux kernel was only exploitable with kernel modification. Consequently, bypassing kernel MTE through conventional memory corruption vulnerabilities remains largely unexplored.

In this paper, we present MTEscape, a novel technique for bypassing asynchronous kernel MTE using conventional memory corruption vulnerabilities. Specifically, our technique exploits heap out-of-bounds (OOB) write and use-after-free (UAF) vulnerabilities. Since these vulnerabilities trigger tag mismatches, they were previously considered to be mitigated by MTE. MTEscape overturns this assumption by bypassing asynchronous MTE through its race window, where tag checking is deferred until a context switch. Although this race window is a known theoretical weakness, we are the first to demonstrate its exploitability in the Linux kernel.

To achieve this, we must overcome several challenges posed by asynchronous kernel MTE. First, the exploit must bypass the fault handling triggered by its own memory corruption. Second, this bypass must be achieved through single-step vulnerability pivoting, since the kernel panics shortly after the initial corruption. Third, the exploit must succeed in a single attempt: any failure under MTE results in a kernel panic, leaving no opportunity for retry.

To address these challenges, we first identify kill switch variables—kernel variables whose modification disables tag fault handling.

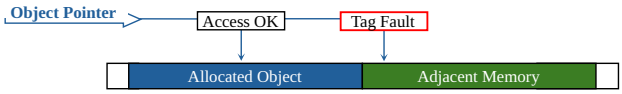
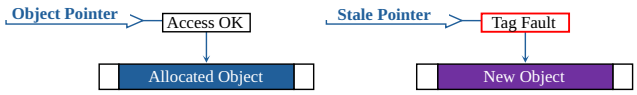


Figure 1: MTE detection of out-of-bounds access.



(a) Accessing with valid pointer (b) Accessing with dangling pointer

Figure 2: MTE detection of use-after-free.

To overwrite these variables under the temporal constraint, we leverage dereference-write objects (DWO). These objects enable an arbitrary write in a single pivoting step, allowing the exploit to complete before fault handling intervenes. Finally, to ensure single-attempt reliability, we employ race stabilization techniques tailored to each vulnerability type.

Our evaluation shows that MTEscape effectively bypasses asynchronous kernel MTE on both synthetic and real-world vulnerabilities. On synthetic vulnerabilities, MTEscape achieves an average success rate of 94.4% on OOB write and 100% on UAF across `kmalloc-64` through `kmalloc-4k`. MTEscape further bypasses asynchronous MTE on 10 real-world Linux kernel CVEs. To mitigate this threat, we propose a lightweight defense that protects kill switch variables with negligible overhead. We responsibly disclosed our findings to the Android Security Team, which acknowledged the vulnerability and awarded a bounty.

In summary, this paper makes the following contributions:

- We present MTEscape, a novel technique that bypasses asynchronous kernel MTE with conventional OOB write and UAF vulnerabilities.
- We demonstrate the effectiveness of MTEscape on synthetic vulnerabilities, achieving 94.4% success on OOB write and 100% on UAF, and on 10 real-world Linux kernel CVEs, succeeding in all cases.
- We propose a mitigation that prevents MTEscape with negligible performance overhead.

We release the source code and evaluation artifacts of MTEscape at https://github.com/kaist-hacking/MTEscape to foster open science.

2 Background

2.1 Memory Tagging Extension

MTE is a hardware feature introduced in the ARMv8.5-A architecture [3] to mitigate memory corruption vulnerabilities. It detects invalid memory accesses at runtime via hardware-based tag validation. Specifically, MTE assigns a 4-bit tag to each memory region and embeds the same tag in the upper bits of pointers that reference this region. On each memory access, the CPU compares the pointer tag with the stored tag and raises an exception on mismatch.

This mechanism can be used to detect both spatial and temporal memory bugs. As shown in Figure 1, for spatial bugs (e.g., buffer overflows), we can assign distinct tags to adjacent allocations. Consequently, any linear overflow crossing object boundaries triggers a tag mismatch. For temporal bugs (e.g., use-after-free), we can retag memory with a different tag when it is freed or reallocated (Figure 2). This ensures that dangling pointers retain the stale tag, causing them to trigger a tag mismatch upon subsequent access.

MTE supports two operating modes: synchronous and asynchronous. In synchronous mode, a tag mismatch immediately raises an exception. This provides precise and timely detection, but incurs significant performance overhead. According to prior work [36], synchronous mode incurs 56% overhead on average and up to 564% in the worst case on SPEC CPU 2006 [13] with a Pixel 8 performance core. This makes synchronous mode impractical for production use.

In contrast, asynchronous mode improves performance by sacrificing detection latency. Specifically, it records tag mismatches in the Tag Fault Status Register (TFSR) and defers their handling to the kernel. This design significantly reduces performance overhead; prior work [36] reports 8% overhead on average and 21% in the worst case under the same setup.

In this work, we focus on the asynchronous mode, as it is the reasonable solution for production use. As mentioned before, synchronous mode incurs significant overhead, making it impractical for production use. Nevertheless, Apple deployed synchronous MTE in the iPhone 17 [2], released in September 2025; however, their approach still exhibits two primary limitations. First, Apple deploys MTE selectively, targeting only small-sized allocations within type buckets to mitigate performance overhead. Consequently, this approach provides only partial protection, leaving its security implications unclear. Second, this implementation is currently constrained to Apple's proprietary hardware, limiting its applicability to other platforms such as Linux. Therefore, it is still important to analyze the security of asynchronous MTE.

2.2 MTE Support in Linux Kernel

The Linux kernel integrates MTE through hardware tag-based KASAN. Unlike other KASAN implementations intended for debugging, this mode is designed for security in production [18]. For that, it extends the SLUB allocator to tag objects upon allocation and deallocation. For each allocation, the kernel assigns a random tag between 0x0 and 0xd; upon deallocation, it is retagged with 0xe. Notably, 0xf is reserved for unchecked pointers required for the kernel to operate. For example, this tag is used when accessing user pages whose tags are unknown to the kernel. Under this policy, the Linux kernel can mitigate OOB and UAF vulnerabilities with a probability of 13/14 (≈ 93%), while it can detect accesses to freed objects deterministically (i.e., 100%).

For asynchronous mode, the kernel inspects the `TFSR_EL1` on (1) context switches, (2) return to userspace, (3) re-entry to kernel, and (4) return from re-entry [9]. Therefore, if a memory corruption occurs during a syscall, it is detected upon returning to userspace or kernel re-entry (e.g., timer interrupts). To address these tag faults, the Linux kernel supports two modes: report mode and panic mode. In report mode, the kernel only logs the tag fault and continues execution. Conversely, in panic mode, the kernel panics and prevents further execution. Since report mode is only

for debugging and provides no protection against exploits [4], this work focuses on panic mode.

3 Overview

3.1 Threat Model

We assume an unprivileged attacker with code execution on a Linux system, whose objective is to achieve privilege escalation. We assume the existence of a memory corruption vulnerability that results in an OOB write or UAF. Regarding defenses, we assume the kernel is protected by asynchronous MTE in panic mode. We further assume that modern kernel exploit mitigations are enabled, including KASLR, PAN, PXN, PAC [29], and KCFI [42]. We assume no kernel modifications or hardware side channels [14, 17, 25] that would facilitate exploitation. Notably, this threat model is consistent with prior work [10, 27, 46], with the additional assumption that asynchronous MTE is enabled as a mitigation.

3.2 Technical Challenges & Our Approach

In this section, we describe the challenges posed by asynchronous kernel MTE and our approaches to overcome them.

3.2.1 Suppressing Panic on Tag Fault. Since tag faults trigger a kernel panic, we must suppress the panic for successful exploitation. This challenge is unique to MTE bypass and absent from traditional kernel exploits. Consequently, prior techniques [5, 21, 24, 27, 46] cannot be directly applied to this problem.

Our Approach. We address this challenge by leveraging *kill switch variables*—kernel variables whose corruption disrupts tag fault handling. These variables fall into two categories: (1) branch condition variables that determine whether tag fault handling is performed, and (2) synchronization locks within the tag fault handling path. If we overwrite these variables with crafted values, we can either bypass or block the tag fault handling. Since these variables reside in the global variable segment, overwriting them typically requires a KASLR bypass and an arbitrary write primitive. However, due to a weakness in the current ARM64 Linux kernel implementation [15], we can overwrite these variables without a KASLR bypass, as detailed in §4.1.

3.2.2 Restricted Vulnerability Pivoting. Given these kill switch variables, an arbitrary write to them suffices to bypass MTE. Under asynchronous MTE, however, obtaining an arbitrary write is nontrivial due to temporal restrictions on vulnerability pivoting. Existing kernel exploitation techniques [24, 27, 46] achieve this primitive in successive stages, each stage corrupting a new object to escalate the attacker's capability. Such an approach fails under asynchronous MTE: the kernel panics shortly after the initial corruption, before subsequent stages can execute.

Our Approach. We address this challenge by employing DWOs, which yield an arbitrary write in a single pivoting step. A DWO is an object that contains a pointer field and allows a write operation through the pointer at an attacker-controlled time; we term this capability the dereference-write primitive (DWP). Specifically, we trigger a memory corruption vulnerability to overwrite the pointer field with the target address (i.e., the address of a kill switch variable). Subsequently, we trigger the write through this corrupted pointer, yielding an arbitrary write in a single step.

3.2.3 Single-Attempt Race Condition. While DWOs reduce vulnerability pivoting to a single step, the resulting arbitrary write must still race against the kernel's tag fault handling. Upon the initial corruption, a tag fault is asynchronously recorded in `TFSR_EL1`, which the kernel later inspects (e.g., upon thread exit). If the kill switch variable is not overwritten before this inspection, the kernel panics, terminating the exploit. Whereas typical kernel exploitation races permit repeated attempts, this race must be won on the first attempt, as a kernel panic precludes any retry.

Our Approach. We address this challenge with dedicated strategies for each vulnerability type. For OOB write vulnerabilities, we win this race by employing CARDSHARK [12] to precisely control the timing of the DWP invocation. We further improve this technique with cache training, detailed in §4.3. For UAF vulnerabilities, we can reliably win this race if the dangling object provides a DWP. In such cases, we overwrite the pointer field of the dangling object through reclamation. Notably, this does not cause a tag fault, since the write is performed through a valid tag assigned during reclamation. Then, we trigger the DWP through the dangling pointer; the kill switch overwrite immediately follows the tag-faulting access, completing before the kernel handles the tag fault.

3.3 MTEscape

Combining our approaches, we introduce MTEscape, a novel technique for bypassing asynchronous kernel MTE. In the following, we detail the specific exploitation procedures for OOB write and UAF vulnerabilities.

OOB Write Vulnerabilities. Figure 3 illustrates how our technique exploits OOB write vulnerabilities. ① We first allocate a DWO adjacent to the vulnerable object. ② Then, using an OOB write, we overwrite the pointer field of the DWO with the address of a kill switch variable. At this point, a tag mismatch occurs, which is handled when the faulting thread exits. ③ Finally, we concurrently trigger the DWP from another thread to overwrite the kill switch variable. We employ our race stabilization technique (§4.3) so that this overwrite completes before the tag fault is handled.

UAF Vulnerabilities. Figure 4 illustrates how MTEscape exploits UAF vulnerabilities. As noted, MTEscape targets UAF in which the dangling object yields a DWP. ① We trigger the UAF vulnerability, freeing the object while retaining a dangling reference to it. ② We then reclaim the freed memory with an attacker-controlled object, thereby overwriting the pointer field of the dangling object. This step does not trigger a tag fault since the write is performed through a valid tag assigned during reclamation. ③ Finally, we invoke the DWP of the dangling object to overwrite the kill switch variable. Although accessing the object via a dangling pointer causes a tag mismatch, the subsequent DWP immediately overwrites the kill switch variable, thus disabling MTE protection.

Requirements. MTEscape imposes the following requirements on the target vulnerability. For OOB write vulnerabilities, the attacker must be able to control the value written by the OOB write, as MTEscape overwrites a DWO with the address of a kill switch

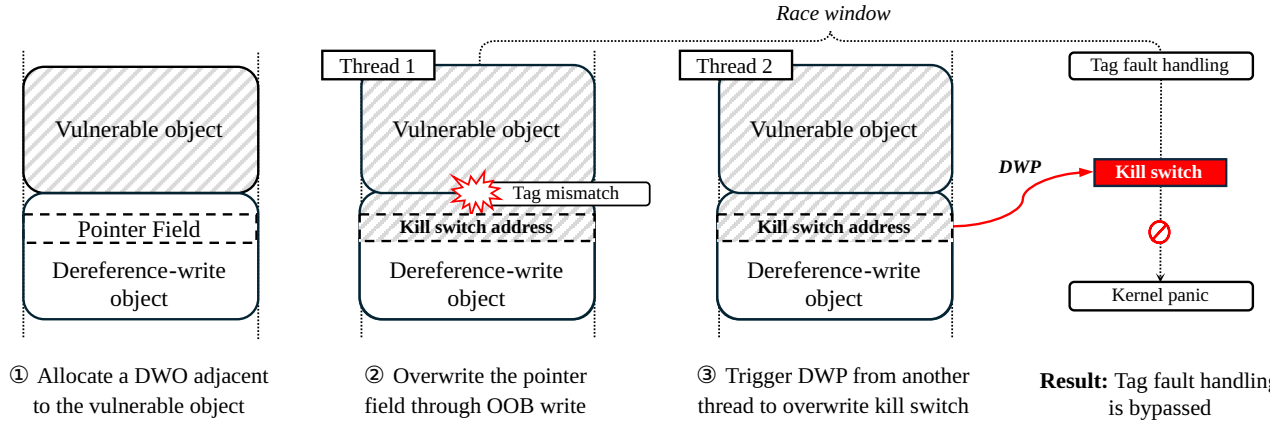


Figure 3: Exploitation of OOB write vulnerability using MTEscape.

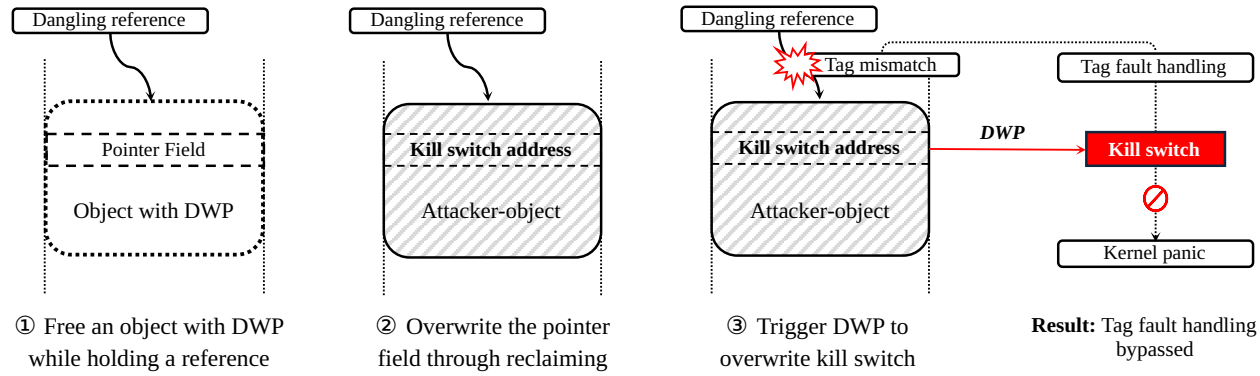


Figure 4: Exploitation of UAF vulnerability using MTEscape.

variable. For UAF vulnerabilities, the dangling object must expose a DWP, as MTEscape directly uses the dangling object as a DWO. These requirements are stricter than those of existing kernel exploitation approaches [21, 27, 46]. However, they are necessitated by the single-step vulnerability pivoting that asynchronous kernel MTE imposes. Furthermore, we demonstrate that these requirements are met in practice by exploiting 10 real-world kernel vulnerabilities, as detailed in §5.2.

4 Technical Details

In this section, we describe the key components of MTEscape in detail: MTE kill switch variables (§4.1), arbitrary write via single-step pivoting (§4.2), and our race stabilization technique (§4.3).

4.1 MTE Kill Switch Variables

To identify potential bypass vectors for the MTE mechanism, we investigated the tag fault handling routines in the Linux kernel. Specifically, we identified three kill switch patterns: (1) branch conditions that skip fault handling, (2) synchronization locks, and (3) loop conditions that stall a thread before panic invocation. We employed CodeQL to discover variables matching these patterns. For each candidate, we manually verified whether corrupting it prevented tag fault handling. We restricted our analysis to global variables, as local variables reside in non-deterministic locations, which makes them difficult to target reliably.

Table 1: MTE kill switch variables in the Linux kernel

Variable	Type†	Feature	Condition
`report_lock`	SL	A lock for KASAN report entry	$\neq 0$
`console_owner_lock`	SL	A lock for kernel logging	$\neq 0$
`console_sem.lock`	SL	A lock for kernel logging	$\neq 0$
`printk_cpu_sync_owner`[1]	AV	Enforce mutual exclusion in call stack dump	$\neq -1$
`panic_cpu`[1]	AV	Enforce mutual exclusion in panic handling	$\neq -1$
`kasan_flags`	BC	KASAN status flag	& 0x3 == 1

† SL: Synchronization lock, AV: Atomic variable, BC: Branch condition

Our analysis identified six kill switch variables, summarized in Table 1. Except for `kasan_flags`, all kill switch variables are atomic primitives for mutual exclusion; these variables prevent concurrent execution of specific paths in the tag fault handling routine, including kernel logging and panic entry. They can be overwritten with almost any non-zero value[1] to mimic a locked state, thus indefinitely stalling the tag fault handling. `kasan_flags`

[1]For `panic_cpu` and `printk_cpu_sync_owner`, the overwritten value must be distinct from the initial state (-1) and the ID of the core currently handling the tag fault, which ranges from 0 to `num_cores` - 1.

```c
struct list_head {
    struct list_head *next, *prev;
};

void __list_del(struct list_head * prev, struct list_head * next)
{
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}
```

Listing 1: Implementation of unlinking in the Linux kernel

```c
void trigger_dwp(struct dwo *obj) {
    void* ptr = obj->ptr;
    // 1. Increment/Decrement
    (*ptr)++;
    // 2. Arithmetic Assignment
    *ptr += x;
    // 3. Direct Assignment
    *ptr = val;
    // 4. Field Dereference Write
    ptr->member = val;
}
```

Listing 2: Representative code patterns of DWOs

is the only variable that serves as a branch condition. Setting it to 1 marks the KASAN status as already reported (Bit 0) without enabling multi-shot mode (Bit 1), causing subsequent faults to be silently dropped.

As these variables reside in the kernel data section, overwriting them typically necessitates an arbitrary write primitive and an information leak to bypass KASLR. However, due to a weakness in the ARM64 Linux kernel [15], an arbitrary write primitive alone is sufficient in practice. Specifically, the ARM64 Linux kernel does not randomize the linear map region, which directly maps physical memory to virtual addresses. Furthermore, physical KASLR on ARM64 is not implemented within the kernel itself, but requires explicit support from the bootloader. Together, these factors render the addresses of kill switch variables deterministic regardless of KASLR. We verified this behavior on the Google Pixel series and Radxa Orion O6, which is our evaluation platform. Without this weakness, MTEscape would additionally require an information-leak primitive to bypass KASLR. In such a case, various side-channel attacks [14, 25, 28] could be employed to bypass KASLR, although our threat model excludes such attacks.

Due to the global scope of kill switch variables, a single overwrite compromises all subsequent tag fault handling. Consequently, a single arbitrary write allows an unprivileged attacker to indefinitely disable MTE protection across the entire system.

4.2 Arbitrary Write via Single-Step Pivoting

Dereference-Write Object. As discussed in §3.2, asynchronous kernel MTE precludes multi-step vulnerability pivoting. To overcome this constraint, we leverage DWOs to achieve an arbitrary write primitive in a single step. A DWO is an object that (i) can be allocated by an unprivileged user, (ii) contains a pointer field, and (iii) allows an attacker to write through the pointer at an attacker-controlled time. The value written through the pointer need not be attacker-controlled, as the majority of kill switch variables accept any non-zero value. By corrupting the pointer field of a DWO and subsequently invoking its write capability (DWP), an attacker can convert a memory corruption primitive into an arbitrary write in a single pivoting step.

Table 2: Dereference-Write Objects in the Linux kernel

Object name	Pointer field	Allocation cache
route4_filter	next	kmalloc-256
crypto_async_request	list_head	kmalloc-1k
hfsc_class	cl_parent	kmalloc-1k
cfg80211_mgmt_registration	list_head	kvmalloc-[64, 64k]
packet_fanout	list_head	kmalloc-[256, 8k]
pipe_buffer	page	kmalloc-cg-[64, 8k]

Objects managed by linked lists constitute a representative class of DWOs in the Linux kernel. Specifically, `list_head` is used for implementing linked lists, and more than 4,000 objects employ this implementation. As shown in Listing 1, this object implements unlinking by dereferencing the `prev` and `next` pointers and writing through them. Thus, an attacker can obtain an arbitrary write by corrupting either `prev` or `next` and unlinking.

`cfg80211_mgmt_registration` is one such object, linked through its `list_head` field. An unprivileged user can allocate it by issuing an NL80211 register frame request over a generic netlink socket. Upon socket close, the kernel unlinks it before freeing it, dereferencing the `list_head` pointers. Since the attacker can trigger both allocation and unlinking, this object serves as a DWO.

Beyond unlinking, various other code patterns illustrated in Listing 2 facilitate a DWP. Since the written value is unconstrained, diverse write operations qualify, such as uncontrolled value writes and arithmetic assignments. As these code patterns are prevalent throughout the kernel codebase, the kernel exposes a broad pool of potential DWOs.

OOB Write Exploitation via Adjacent Allocation. For OOB write vulnerabilities, we allocate a DWO immediately after the vulnerable object. We then trigger the OOB write to overwrite the pointer field with the address of the target kill switch variable. This requires the OOB write to provide attacker-controlled timing and content. Such value control is common in practice; among the kernel OOB write vulnerabilities analyzed in prior work, 20 of 29 provide arbitrary value control [6, 24, 46].

The remaining requirement is that the DWO reside in the same slab cache as the vulnerable object to enable adjacent allocation. To satisfy this across diverse OOB vulnerabilities, we identified DWOs spanning a wide range of general-purpose cache sizes. We employed CodeQL to discover candidates matching the pointer access patterns in Listing 2. For each candidate, we manually verified that an unprivileged user can allocate it and trigger its DWP at a controlled time.

Table 2 lists the identified DWOs along with their pointer fields and allocation caches. Our DWO coverage spans from `kmalloc-64` to `kmalloc-4k`, and further extends to vmalloc allocations beyond that size. We also cover most cgroup-accounted ("cg") caches, enabling exploitation of OOB write vulnerabilities in cg-isolated subsystems such as `io_uring` and `nf_tables`. Therefore, our technique applies broadly to OOB write vulnerabilities across diverse allocation caches in the Linux kernel.

UAF Exploitation via Reclamation. Constrained to single-step pivoting, an attacker exploiting a UAF has two options for targeting

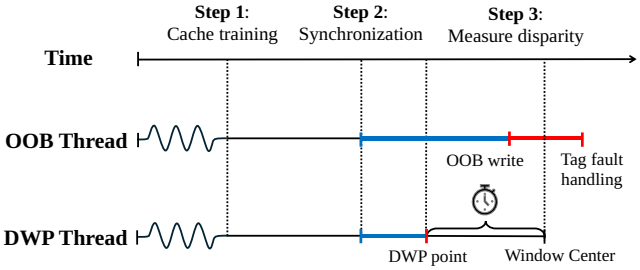


Figure 5: MTEscape's profiling phase

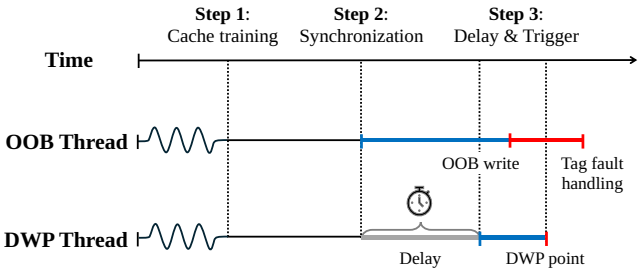


Figure 6: MTEscape's exploitation phase

a kill switch variable. The first is to reclaim the dangling object with an attacker-allocated DWO and trigger a UAF write to overwrite its pointer field. Alternatively, in cases where the dangling object itself provides a DWP, the attacker can reclaim it to directly overwrite the pointer field.

However, the first option is largely infeasible in practice. Most UAF vulnerabilities provide only limited write primitives: the write offset is fixed, and the value written is rarely attacker-controlled, often being a heap pointer, fixed constant, or other kernel-internal value [11, 27, 40, 41, 44]. Consequently, it is often not possible to write the address of a kill switch variable with such primitives. Even if the attacker can control the value written, they still need to find a DWO whose pointer field lies exactly at the offset targeted by the UAF write.

The second option is not subject to these limitations. Unlike the UAF write, which rarely offers value control, a wide range of kernel objects exhibit the code patterns in Listing 2, thus providing a DWP. When a UAF vulnerability leaves such an object dangling, we can leverage its DWP. This also eliminates the offset constraint by overwriting the dangling object entirely with attacker-controlled content.

Furthermore, this approach wins the race against tag fault handling by design. In the first option, the UAF write itself triggers a tag fault, requiring a separate thread to concurrently invoke the DWP before the fault is handled. In contrast, reclamation does not trigger a tag fault, since the kernel assigns a fresh valid tag upon reclamation. Instead, the tag mismatch occurs when the kernel later accesses the dangling object during DWP invocation. After this tag-faulting access, the DWP is sequentially triggered within the same execution flow, reliably completing before tag fault handling. The only exception is a timer interrupt between the access and DWP invocation, which is highly unlikely given their temporal proximity. As a result, the DWP timing need not be controlled, unlike in OOB write exploitation.

Therefore, we adopt the second option in MTEscape. Although this strategy requires a UAF whose dangling object provides a DWP, it is the most feasible UAF exploitation under the single-step pivoting constraint and provides a reliability advantage. We confirm this reliability in §5.1 and show that recent real-world CVEs satisfy the DWP requirement in §5.2.

```c
volatile unsigned int t1_ready, t2_ready, synced;

void delay(unsigned long interval){
    unsigned long end = read_cntvct() + interval;
    while (read_cntvct() < end);
}

void* thread_OOB(void* arg) {
    pin_cpu(1);

    // Train on a dummy vulnerable object with in-bounds inputs
    for (int i = 0; i < CACHE_TRAINING_COUNT; i++)
        syscall_OOB(dummy_vuln, INBOUNDS_LEN);

    t1_ready = 1;
    while (!synced);
    syscall_OOB(target_vuln, OOB_LEN);
}

void* thread_DWP(void* arg) {
    pin_cpu(2);

    // Train on a dummy DWO with the pointer field uncorrupted
    for (int i = 0; i < CACHE_TRAINING_COUNT; i++)
        syscall_DWP(dummy_dwo);

    t2_ready = 1;
    while (!synced);
    if (exploit_phase)
        delay(MEASURED_DISPARITY);

    syscall_DWP(target_dwo);
}

void exploit(){
    pthread_t th1, th2;
    pin_cpu(0);
    pthread_create(&th1, NULL, thread_OOB, NULL);
    pthread_create(&th2, NULL, thread_DWP, NULL);

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

Listing 3: Simplified implementation of MTEscape's profiling & exploitation phases

4.3 Race Stabilization

Bypassing asynchronous MTE ultimately requires winning the race against tag fault handling. For UAF, the reclamation-based approach wins this race by design, requiring no additional timing control. For OOB write, however, we must precisely control the timing of DWP invocation. Specifically, we must trigger the DWP after the OOB write completes but before the kernel handles the resulting tag fault.

CARDSHARK. For that, we leverage CARDSHARK [12], a technique for stabilizing race conditions. It defines misalignment as the time difference between the racing point and the midpoint of

the race window. Then, it identifies two primary sources of misalignment: (1) invocation disparity, the scheduling time difference between two racing threads, and (2) path disparity, the execution time difference in reaching the racing points. CARDSHARK addresses this misalignment in two ways: (1) synchronization via busy-waiting to eliminate invocation disparity, and (2) a delay applied to the shorter-path operation to compensate for path disparity.

Figures 5 and 6 illustrate how CARDSHARK operates, except for Step 1 (cache training), which we introduce in MTEscape. In the profiling phase, CARDSHARK synchronizes the two racing threads and measures their path disparity. The measured disparity is then used during exploitation to delay the shorter-path thread, aligning its race point with the midpoint of the race window.

Cache Training. While CARDSHARK aligns macro-level disparities, residual non-determinism remains at the microarchitectural level. Specifically, microarchitectural state, such as cache or branch predictor states, varies across executions and induces variation in execution time. This variation can perturb the precise alignment of racing operations.

To mitigate this microarchitectural variation, we introduce a cache training step prior to the actual race attempt. Specifically, we iteratively exercise the execution paths of both the OOB write and the DWP invocation without triggering the vulnerability. For this, we allocate dummy objects used only for cache training, distinct from those involved in the actual race. For the OOB path, we trigger the execution path on a dummy vulnerable object with in-bounds inputs. For the DWP path, we trigger the DWP on a dummy DWO with the pointer field uncorrupted. We empirically train each path five times; this suffices to saturate standard 2-bit branch predictors while minimizing the training duration. A prolonged training phase increases the risk of context switches, which would flush the microarchitectural state and nullify the training effects.

Two-Phase Procedure. Our procedure for OOB write comprises two phases: profiling and exploitation, illustrated in Figure 5 and Figure 6, respectively. The profiling phase employs kernel instrumentation to quantify the execution time disparity between the OOB write and the DWP trigger. The exploitation phase then leverages these measurements to mount the actual attack. Both phases begin with cache training to stabilize execution.

Since the profiling phase involves kernel modification, it necessitates an attacker-controlled kernel on hardware identical to the victim's environment. As our threat model assumes an attacker with code execution on the victim system, it is feasible to identify the target hardware and obtain an identical device for profiling. The exploitation phase, in contrast, runs directly on the victim system, where kernel modification is infeasible.

In the profiling phase, we insert timing instrumentation into the kernel at three critical points: (1) immediately following the OOB write (race window start), (2) at the location of the DWP execution, and (3) immediately prior to accessing the target kill switch variable (race window end). We utilize the `CNTVCT_EL0` register to obtain precise timing measurements. With this instrumentation in place, we first perform cache training to stabilize execution. Subsequently, we employ the methods of CARDSHARK. Specifically, we synchronize the execution to simultaneously initiate the OOB write and the DWP invocation, and measure the resulting time disparity through our instrumentation.

In the exploitation phase, we employ the same cache training and synchronization mechanisms as in the profiling phase. Then, instead of initiating the OOB write and the DWP simultaneously, we delay the DWP triggering by the average time difference between the race window center and the DWP execution point measured during profiling. This delay aligns the DWP execution to the center of the race window. Consequently, the DWP is reliably executed between the OOB write and the subsequent tag fault handling.

Listing 3 outlines a simplified implementation of our technique. For both phases, the main thread spawns two worker threads—one for the OOB write and the other for the DWP invocation—each pinned to a distinct core. The threads first perform cache training on dummy objects by iteratively executing the target code paths with benign inputs. They are then synchronized via busy loops. Upon synchronization, the execution logic diverges depending on the phase. In the profiling phase, the threads simultaneously trigger the OOB write and DWP to measure the execution time disparity. In the exploitation phase, the DWP thread instead delays its trigger by the profiled offset.

5 Evaluation

In this section, we aim to answer the following questions:

- **RQ1:** How effective is MTEscape in bypassing asynchronous kernel MTE? (§5.1)
- **RQ2:** How practical is MTEscape for exploiting real-world vulnerabilities? (§5.2)
- **RQ3:** How does MTEscape's race stabilization improve over prior work? (§5.3)
- **RQ4:** How well does MTEscape's race stabilization transfer across devices of the same model? (§5.4)

Evaluation Setup. We conducted our evaluation on a Radxa Orion O6, a publicly available ARMv9 platform with MTE support. The platform features a CIX P1 CD8180 processor with 12 cores and 16 GB of RAM. We selected this platform instead of Pixel devices due to their restricted debugging capabilities and the need for frequent kernel flashing. We performed the experiments on Linux v6.16, the most recent release at the time of evaluation, which includes up-to-date MTE support and kernel exploit mitigations. Consistent with our threat model, we configured the kernel to enforce all standard mitigations, including KASLR, PAN, PXN, PAC, and KCFI, alongside asynchronous MTE in panic mode. The cross-device validation in §5.4 required two devices of the same model. Since a second Radxa Orion O6 was unavailable at the time of evaluation, we used two Orange Pi 6 Plus devices only for this experiment, which use the same CIX P1 SoC as the Radxa Orion O6.

5.1 Synthetic Vulnerabilities

Evaluation Design. We first evaluate MTEscape's effectiveness across different cache sizes. For that, we evaluate it on synthetic vulnerabilities, which allow us to control the target cache. We cover `kmalloc-64` through `kmalloc-4k`, the range spanned by our discovered DWOs. To this end, we develop a custom kernel module with OOB write and UAF vulnerabilities. The module provides

```c
struct vuln_obj {
    void* pointer;
    char data[];
} *objs[100];

static long vuln_ioctl(struct file *_, unsigned int cmd, unsigned long arg)
{
    copy_from_user(&req, arg, sizeof(req));
    obj = objs[req.idx];
    switch (cmd) {
    case ALLOC:
        obj = kmalloc(req.size, GFP_KERNEL);
        memcpy(obj, req.data, req.size);
        break;
    case FREE:
        kfree(obj);
        break;
    case OOB_WRITE:
        memcpy(obj, req.data, req.len);
        break;
    case UAF_INVOKE_DWP:
        *(obj->pointer) = req.val;
        break;
    }
}
```

Listing 4: Simplified implementation of the custom kernel module.

interfaces for allocating and releasing a custom object, as well as primitives for OOB write and DWP on a freed object. A simplified implementation of the module is shown in Listing 4.

For each vulnerability type, we construct an exploit following Listing 3 to overwrite a kill switch variable. For OOB write, we used `cfg80211_mgmt_registration` as the DWO, which is allocatable in all evaluation caches. For UAF, we leveraged the DWP embedded in our custom module (i.e., `UAF_INVOKE_DWP` in Listing 4). For the kill switch variable, we selected `panic_cpu`. As it is the last kill switch variable along the tag fault handling path, it provides the widest race window, maximizing the probability of a successful race.

We evaluate each exploit under both idle and busy system conditions, consistent with prior evaluations of kernel exploitation techniques [16, 19, 27, 31]. In the busy condition, we use `stress-ng` to generate background workload that stresses the CPU, I/O, and virtual memory; in the idle condition, we introduce no background workload. To avoid assuming that an attacker knows or can reproduce the victim's background workload, we do not repeat OOB profiling under noise. Instead, we use the delay obtained from idle profiling for the busy-condition experiments.

Given non-determinism in kernel exploitation, we conduct 100 trials for each exploit at each cache size under both system conditions and measure the success rate. We consider a trial successful if the kill switch variable is overwritten and the kernel remains alive. This indicates that tag fault handling is bypassed and subsequent MTE protection is disabled. Otherwise, we consider it failed. After each trial, we reboot the system to eliminate the effects of previous trials.

Results. Our evaluation results are summarized in Table 3. Under the idle condition, MTEscape achieved an average success rate of 94.38% for OOB write, ranging from 92% to 98% across cache sizes, and 100% for UAF. Under the busy condition, the UAF success rate remained nearly unchanged at 99.75%, whereas the OOB write success rate decreased to 11.63%.

Table 3: Success rates of MTEscape on synthetic vulnerabilities by cache size and system state.

Cache Size	MTEscape (OOB)		MTEscape (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%
Overall	94.38%	11.63%	100%	99.75%

Table 4: Exploitability of MTEscape on real-world CVEs. For UAF, the DWO column denotes the dangling object providing the DWP.

CVE	Type	Cache	DWO
CVE-2021-3715	UAF	kmalloc-256	`route4_filter`
CVE-2021-43267	OOB	kmalloc-1k	`packet_fanout`
CVE-2022-0185	OOB	kmalloc-4k	`cfg80211_mgmt_registration`
CVE-2022-2588	UAF	kmalloc-256	`route4_filter`
CVE-2022-34918	OOB	kmalloc-cg-64	`pipe_buffer`
CVE-2023-4623	UAF	kmalloc-1k	`hfsc_class`
CVE-2024-36016	OOB	kmalloc-2k	`cfg80211_mgmt_registration`
CVE-2024-53141	OOB	kmalloc-cg-2k	`pipe_buffer`
CVE-2024-58240	UAF	kmalloc-1k	`crypto_async_request`
CVE-2025-21702	UAF	kmalloc-1k	`hfsc_class`

We identify three factors contributing to failures of the OOB write exploit: (1) execution time variation, (2) synchronization failure, and (3) non-contiguous heap allocation. First, despite our race control, natural variations in execution time can shift the DWP outside the race window. Second, during thread synchronization in Listing 3, a timer interrupt may schedule out a thread after setting the ready flag. Since the flag remains set, the main thread sets the synchronization flag once the other thread is ready. However, the scheduled-out thread starts only after being rescheduled, causing the two threads to start at different times. Third, OOB write exploitation requires contiguous allocation of the vulnerable object and the DWO, which may not always succeed. Despite these factors, OOB write exploitation with MTEscape remains reliable under the idle condition. Under the busy condition, however, this reliability degrades, as frequent thread switching can increase the likelihood of synchronization failures and increased heap activity can interfere with contiguous heap allocation.

Overall, MTEscape effectively bypasses asynchronous MTE through UAF under both idle and busy system conditions, and through OOB write under the idle condition. Although OOB write exploitation with MTEscape is less reliable under the busy condition, it remains a practical threat under our threat model because the attacker may choose to launch the attack when system noise is low. Moreover, although a failed attempt triggers a kernel panic, the attacker can retry once the system reboots.

5.2 Real-World Vulnerabilities

We evaluate the practicality of MTEscape by applying it to 10 real-world vulnerabilities. As summarized in Table 4, we selected OOB write and UAF vulnerabilities spanning diverse cache types, all disclosed within the past five years. Each vulnerability satisfies the assumptions of MTEscape: for OOB write, the attacker controls the value written by the OOB write; for UAF, the dangling object exposes a DWP. For each vulnerability, we backported it to Linux v6.16 and evaluated it under the same configuration. We consider an exploit successful if the kill switch variable is overwritten and the kernel remains alive. This allows us to achieve privilege escalation with existing techniques [21, 27, 46], as MTE protection has been disabled.

Our technique successfully bypassed MTE protection in all 10 cases, demonstrating its practicality. Through case studies, we demonstrate how MTEscape exploits real-world vulnerabilities to bypass asynchronous MTE and subsequently achieve privilege escalation.

Case 1: CVE-2024-36016 (OOB). CVE-2024-36016 is an OOB write vulnerability in the `n_gsm` subsystem. The vulnerability arises from inconsistent boundary validation between two encoding modes. Both modes share a single `count` field that persists across mode transitions: the advanced mode bounds `count` against a fixed maximum receive unit, whereas the basic mode terminates a frame only when `count` equals an attacker-controlled frame-length field. Exploiting this asymmetry, an attacker first inflates `count` beyond the frame-length field in the advanced mode; upon switching to the basic mode, `count` already exceeds the length field, so the equality condition never holds. Consequently, the attacker can write attacker-controlled data of arbitrary length beyond the buffer boundary.

We exploit this vulnerability in four steps: (1) groom the heap so that a DWO is placed immediately after the vulnerable buffer; (2) issue the OOB write to overwrite the DWO with the address of the kill switch variable; (3) after a profiled delay, invoke the DWP to overwrite the kill switch variable; and (4) repeat the same procedure against `modprobe_path` to achieve privilege escalation.

As the DWO, we employ `cfg80211_mgmt_registration`, which can be allocated in the same cache (`kmalloc-2k`) as the vulnerable buffer. To place it immediately after the vulnerable buffer, we must allocate them consecutively from a fresh slab cache, as fragmented free lists in existing slabs yield non-sequential allocations. Furthermore, a fresh slab cache lets us avoid placing the vulnerable buffer in the final slot, where slab boundaries prevent contiguous allocation. We therefore employ PSPRAY [19], a software side channel to detect fresh slab allocations.

With this heap layout in place, we first run the profiling phase to measure the delay. In the subsequent exploitation phase, we trigger the OOB write to overwrite the DWO's `list_head` with the address of `panic_cpu`, a kill switch variable providing the widest race window. After the profiled delay, we close the socket to invoke the DWP, which unlinks the DWO and overwrites `panic_cpu`. When the asynchronous tag fault is eventually handled, the overwritten `panic_cpu` suppresses the panic, bypassing MTE.

After bypassing MTE, we proceed to privilege escalation by repeating the same process to overwrite `modprobe_path`. This time, race control is unnecessary since MTE protection remains globally disabled. We trigger the OOB write to overwrite the DWO's `list_head` field with the address of `modprobe_path+1` and a kernel address ending with `0x2f706d74`, which reads as `"tmp/"` in little-endian memory. We then close the socket to trigger the unlink, which overwrites `modprobe_path` from `/sbin/modprobe` to `/tmp/xxxxprobe`. We plant a payload at this path and trigger the modprobe usermode helper by binding an `AF_ALG` socket [43], which executes the payload as root.

Case 2: CVE-2024-58240 (UAF). CVE-2024-58240 is a UAF vulnerability in the TLS subsystem. The vulnerability arises because the TLS decryption path releases the `crypto_async_request` object while `cryptd` still holds it for asynchronous processing. To trigger this vulnerability, the attacker registers a custom asynchronous `ccm(aes)` implementation with higher priority than the kernel's default synchronous one. Linux's crypto API then selects the custom implementation, routing TLS decryption through `cryptd`. Consequently, `cryptd`'s later access to the freed object results in a use-after-free.

We exploit this vulnerability in four steps: (1) trigger the UAF by forcing TLS decryption through `cryptd`; (2) reclaim the freed object to overwrite a pointer field with the address of the kill switch variable; (3) invoke the DWP to overwrite the kill switch variable; and (4) repeat the same procedure against `modprobe_path` to achieve privilege escalation.

We achieve a DWP through the dangling `crypto_async_request`'s `list_head` field. Specifically, `cryptd` queues the request via this field, then unlinks the object after processing, thereby dereferencing the `list_head` pointers. We reclaim the freed object before `cryptd` handles it, overwriting the `list_head` with controlled pointers. The subsequent unlink then performs an arbitrary write.

Overwriting the `list_head` via reclamation, however, requires satisfying two constraints. First, the primitive must overwrite the object from offset zero, since `list_head` sits at the start. This rules out primitives like `user_key_payload` that begin partway through. Second, it must preserve the object's remaining fields, which `cryptd` accesses before the unlink. We satisfy both using a `setxattr` reclaim combined with FUSE [37]. FUSE delegates page faults on the user buffer to our userspace handler, letting us pause the kernel's copy at an arbitrary offset; we halt right after the `list_head`, overwriting it from offset zero while leaving the trailing fields intact.

With the object reclaimed, we overwrite the `list_head` with the address of `kasan_flags` and a kernel address ending with `0x1`. When `cryptd` unlinks the object, it overwrites `kasan_flags` with our crafted address; the `0x1` ending marks KASAN as already reported in single-shot mode, thus disabling tag fault handling. Once MTE is bypassed, we escalate privileges by overwriting `modprobe_path`, as in Case 1.

5.3 Race Stabilization

Evaluation Design. To quantify how MTEscape's race stabilization improves over prior work, we measure the precision of race control and the resulting race success rate across real-world vulnerabilities. As MTEscape does not require race stabilization for UAF, we perform this evaluation only on OOB write vulnerabilities.

Table 5: Race timing and success rates across five real-world OOB CVEs (median ± MAD). Relative misalignment is defined as $2M/W$, where M is the misalignment and W is the race window duration. Per CVE, the lowest relative misalignment and the highest success rate are shown in bold and underlined.

CVE	Method	Win. center	Win. duration	DWP point	Misalignment	Rel. Mis. (%)	Succ. (%)
CVE-2021-43267	Naive	2360.2 ± 96.7	116.8 ± 2.6	119.4 ± 13.4	2220.4 ± 95.1	3764.7 ± 139.1	0.0
	CARDSHARK	2734.6 ± 152.9	124.2 ± 5.1	2654.3 ± 0.5	202.9 ± 83.7	326.7 ± 145.5	11.4
	MTEscape	64.8 ± 2.8	94.9 ± 2.0	65.0 ± 0.6	2.9 ± 1.8	**6.1 ± 3.7**	**99.3**
CVE-2022-0185	Naive	54.0 ± 3.2	97.1 ± 5.2	14.2 ± 40.5	68.4 ± 36.8	136.4 ± 62.9	24.6
	CARDSHARK	83.4 ± 12.2	137.2 ± 23.9	81.8 ± 4.4	11.6 ± 7.1	13.9 ± 10.1	**99.3**
	MTEscape	63.9 ± 8.0	122.8 ± 15.2	60.1 ± 0.4	6.8 ± 1.2	**11.6 ± 2.6**	99.2
CVE-2022-34918	Naive	155.2 ± 32.9	261.0 ± 56.5	−4212.6 ± 1394.1	4374.9 ± 1428.4	3314.4 ± 417.4	0.0
	CARDSHARK	228.2 ± 12.4	374.8 ± 19.2	209.1 ± 0.3	25.9 ± 4.9	**14.0 ± 3.3**	**99.8**
	MTEscape	205.3 ± 16.3	370.1 ± 21.2	189.3 ± 0.5	27.3 ± 4.9	15.2 ± 2.3	99.6
CVE-2024-36016	Naive	108.5 ± 10.6	149.1 ± 6.7	212.5 ± 542.4	563.2 ± 517.9	716.9 ± 656.3	29.0
	CARDSHARK	183.7 ± 4.2	246.2 ± 2.9	187.8 ± 0.9	7.7 ± 3.9	6.3 ± 3.2	97.8
	MTEscape	163.8 ± 1.6	245.7 ± 2.8	160.8 ± 0.3	3.3 ± 1.6	**2.7 ± 1.3**	**98.3**
CVE-2024-53141	Naive	178.3 ± 13.6	188.3 ± 19.8	104.7 ± 8.2	78.1 ± 15.1	82.8 ± 14.0	82.6
	CARDSHARK	271.2 ± 22.3	187.0 ± 18.9	258.8 ± 1.0	21.9 ± 12.7	24.4 ± 13.1	95.5
	MTEscape	122.4 ± 11.9	158.8 ± 21.2	114.4 ± 0.1	16.5 ± 8.0	**19.6 ± 9.1**	**98.6**

(Unit for timing values: 10^3 `CNTVCT_EL0`)

We use two metrics to quantify race stabilization: the race success rate and the relative misalignment. The race success rate is the fraction of trials in which the DWP trigger falls within the race window. Because success rates can saturate near 100%, we additionally adopt the relative misalignment, a finer-grained measure of timing precision. We define it as $2M/W$, where W is the race window duration and M is the misalignment, defined as the absolute difference between the race window center and the DWP trigger point. The factor of 2 reflects that misalignment is measured from the window center, so 100% corresponds to a DWP trigger at the window edge. A lower value indicates that the DWP trigger lands closer to the window center, reflecting more precise race control.

We compare MTEscape against a naive approach and CARDSHARK [12], which MTEscape builds upon. The three configurations are defined as follows:

(1) **Naive**: We launch two threads sequentially without synchronization: the first issues the OOB write, and the second invokes the DWP. This ordering gives the OOB write a chance to overwrite the DWO before the DWP executes.
(2) **CARDSHARK**: As described in CARDSHARK [12], we launch two threads and synchronize their start time. The DWP-triggering thread is delayed by a profiled execution-time gap between the two threads.
(3) **MTEscape**: In addition to CARDSHARK, we apply cache training before synchronization.

A simplified implementation of the three configurations is shown in Listing 5. In all configurations, we pin the main, OOB, and DWP threads to cores 0, 1, and 2, respectively. For CARDSHARK and MTEscape, we run the profiling phase independently for each configuration, and delay the DWP thread by the corresponding value. In the exploitation phase, we run the exploit with the profiling instrumentation and record the timing values from each trial. For each configuration, we run 100 trials per boot and repeat across 10 reboots.

Results. Our evaluation results are summarized in Table 5. All timing values are reported relative to the point immediately before triggering the OOB write in userspace. We report each timing metric as the median ± MAD (median absolute deviation), which is robust to the extremely large values occasionally produced by synchronization failures. These failures are instead reflected in the race success rate.

In terms of race success rate, both CARDSHARK and MTEscape exceed 95% on four of the five CVEs; only MTEscape reaches this level on CVE-2021-43267. The naive approach achieves only 0–29% except for CVE-2024-53141, where it achieves 82.6%. On this CVE, the sequential spawn delay coincidentally aligns the DWP trigger with the race window. Since the race window differs per vulnerability and asynchronous kernel MTE requires a one-shot bypass, the naive approach does not generalize across vulnerabilities.

CVE-2021-43267 shows the largest gap between CARDSHARK and MTEscape: MTEscape achieves 99.3% while CARDSHARK reaches 11.4%. Under CARDSHARK, the DWP point is stable with MAD 0.5k, but the window center has an MAD of 152.9k, larger than the window duration median of 124.2k. MTEscape's cache training reduces the window center MAD to 2.8k, cutting the relative misalignment from 326.7% to 6.1%. MTEscape thus enables race stabilization on this CVE, which CARDSHARK cannot handle.

On the remaining four CVEs, MTEscape outperforms CARDSHARK in race success rate by 3.1 and 0.5 percentage points on two CVEs, while trailing by 0.1 and 0.2 points on the others. In relative misalignment, MTEscape achieves more precise control on three of the four CVEs, trailing only on CVE-2022-34918. On these three CVEs, MTEscape's cache training narrows the race window compared to CARDSHARK. Nevertheless, the misalignment

```c
// Profiled delay values, measured independently for each configuration.
#define DELAY_CARDSHARK  /* profiled value */
#define DELAY_MTESCAPE   /* profiled value */

enum { NAIVE, CARDSHARK, MTESCAPE } config;
volatile unsigned int t1_ready, t2_ready, synced;

void* thread_OOB(void* arg) {
    pin_cpu(1);

    if (config == MTESCAPE)
        for (int i = 0; i < CACHE_TRAINING_COUNT; i++)
            syscall_OOB(dummy_vuln, INBOUNDS_LEN);

    if (config != NAIVE) {
        t1_ready = 1;
        while (!synced);
    }
    syscall_OOB(target_vuln, OOB_LEN);
}

void* thread_DWP(void* arg) {
    pin_cpu(2);

    if (config == MTESCAPE)
        for (int i = 0; i < CACHE_TRAINING_COUNT; i++)
            syscall_DWP(dummy_dwo);

    if (config != NAIVE) {
        t2_ready = 1;
        while (!synced);
        delay(config == CARDSHARK ? DELAY_CARDSHARK : DELAY_MTESCAPE);
    }
    syscall_DWP(target_dwo);
}

void exploit(){
    pthread_t th1, th2;
    pin_cpu(0);

    pthread_create(&th1, NULL, thread_OOB, NULL);
    pthread_create(&th2, NULL, thread_DWP, NULL);

    if (config != NAIVE) {
        while (!t1_ready || !t2_ready);
        synced = 1;
    }
}
```

Listing 5: Simplified implementations of the Naive, CARDSHARK, and MTEscape race stabilization configurations.

median shrinks by a larger factor, lowering the relative misalignment. Overall, MTEscape does not provide uniform performance gains over CARDSHARK, but achieves reliable race stabilization for CVE-2021-43267, whereas CARDSHARK fails to do so.

5.4 Cross-Device Validation

We evaluate whether MTEscape's OOB write exploitation remains effective when the attacker profiles the delay on one device and runs the exploit on another device of the same model. Because profiling requires kernel modification, it must take place on attacker-controlled hardware that matches the victim environment, while exploitation runs on the victim system. Under our threat model, the attacker can identify the target hardware and obtain an identical device for profiling. For this approach to work, however, the profiled delay must remain valid across devices.

Evaluation Design. We perform this evaluation on the five real-world OOB write CVEs in §5.2. We exclude UAF cases because they win the race by design and do not depend on profiled delay. For each CVE, we profile the delay on one device and run the exploit on the other device using the measured delay. This setup directly tests whether the profiled delay transfers across devices of the same model. As in §5.3, we run 100 trials per boot and repeat across 10 reboots for each CVE. We define race success as the DWP trigger falling within the race window, and report the race success rate over these trials.

Table 6: Cross-device race success rates for real-world OOB write CVEs.

CVE	Success Rate
CVE-2021-43267	100.00%
CVE-2022-0185	99.90%
CVE-2022-34918	100.00%
CVE-2024-36016	93.40%
CVE-2024-53141	99.60%

Table 7: Kill switch variables and proposed mitigations.

Variable	Mitigation
`kasan_flags`	Remove kill switch capability
`report_lock`	Boot-time randomized lock
`console_owner_lock`	Remove kill switch capability
`console_sem.lock`	Remove kill switch capability
`printk_cpu_sync_owner`	Remove kill switch capability
`panic_cpu`	Boot-time randomized lock

Results. Table 6 reports the cross-device race success rates for the five OOB write CVEs. MTEscape achieves at least 93.4% race success on all five CVEs and exceeds 99.6% on four of them. These results suggest that hardware differences have only a limited effect on execution time when the same execution path runs on CPUs of the same model. Overall, a delay profiled on one device remains effective when applied to another device of the same model.

6 Mitigation

In this section, we present a mitigation to prevent MTEscape.

6.1 Overview

One straightforward approach to prevent MTEscape is to use synchronous MTE. This will eliminate the race window that arises from asynchronous fault handling, thereby preventing MTEscape. However, this approach is impractical in performance-sensitive environments. As detailed in §2.1, synchronous MTE introduces an average overhead of 56% with a worst case of 564% on SPEC CPU 2006 [13] with a Pixel 8 performance core [36].

Instead, we mitigate MTEscape by hardening kill switch variables. As summarized in Table 7, we apply two approaches according to the characteristics of each kill switch variable: (1) removing kill switch capability, and (2) boot-time randomized locks.

Removing Kill Switch Capability. To mitigate MTEscape, we first eliminate the kill switch capability where possible. For example, we protect `kasan_flags` by first checking whether the kernel is in panic mode; if so, the handler proceeds with fault handling without inspecting `kasan_flags`, neutralizing the kill switch capability. As another example, to protect the locks used for kernel logging, we relocate kernel logging to a safe point within the panic routine—specifically, after all other cores have been halted. At this point, even

```c
int random_lock_ro __ro_after_init;
atomic_t random_lock;

void __init init_random_lock() {
    while (!random_lock_ro)
        get_random_bytes(&random_lock_ro, sizeof(random_lock_ro));
}

void acquire_random_lock() {
    while (atomic_xchg(&random_lock, random_lock_ro) == random_lock_ro);
}

void release_random_lock() {
    atomic_set(&random_lock, 0);
}
```

Listing 6: Implementation of a boot-time randomized lock

if an attacker overwrites these locks, exploitation cannot proceed: all other cores have been stopped, and the panic-handling core spins indefinitely on the lock.

Boot-time Randomized Lock. However, this approach is not applicable to functionally indispensable kill switch variables. In the current implementation, such variables include `report_lock` and `panic_cpu`, both of which are locks protecting critical sections in tag fault handling.

To address this limitation, we introduce boot-time randomized locks, as described in Listing 6. These prevent an attacker from inducing deadlocks without knowing the randomized lock value. A boot-time randomized lock consists of two variables: `random_lock_ro` and `random_lock`. During boot time, the kernel initializes `random_lock_ro` with a random value and sets it to read-only. The lock state is represented by `random_lock`: if it equals `random_lock_ro`, the lock is held; otherwise, it is released. Acquiring the lock atomically compares and exchanges `random_lock` with `random_lock_ro`; if the previous value was already `random_lock_ro`, the caller spins. Releasing the lock sets `random_lock` to zero.

In a boot-time randomized lock, the value representing the locked state is unknown to the attacker. This contrasts with conventional locks, where any non-zero value denotes the locked state. Since the attacker lacks knowledge of `random_lock_ro`, they cannot manipulate the lock variable into the locked state to force a deadlock.

6.2 Evaluation

Security Analysis. By applying our mitigation, we can defend against MTEscape regardless of vulnerability type. Even if an attacker achieves arbitrary write capability through memory corruption, overwriting kill switch variables either fails to block tag fault handling or requires knowledge of a randomized value unknown to the attacker. This makes it infeasible to bypass tag fault handling, and the resulting kernel panic terminates the exploit.

Performance. To evaluate the performance overhead of our mitigation, we employed two complementary benchmark suites: LMbench [30] for micro-benchmarking and the Phoronix Test Suite [39] for macro-benchmarking. We selected LMbench to isolate the impact on low-level system primitives, specifically system call latency, context switching, and memory bandwidth. Conversely, we used the Phoronix Test Suite to assess the performance of real-world applications, such as Apache and OpenSSL. We conducted all tests in the same environment described in §5. In this evaluation, we report average values from 50 runs for LMbench and 3 runs for the Phoronix Test Suite. We ran the Phoronix Test Suite fewer times because, as a macro-benchmark suite, each run takes substantial time to complete.

Table 8: Performance overhead of our mitigation.

Benchmark	Baseline	Mitigation	Overhead
Phoronix Test Suite			
Apache (req/s)	19.7k	20.0k	-1.75%
nginx (req/s)	18.3k	18.5k	-0.78%
OpenSSL (sign/s)	1063.8	1062.7	+0.10%
OpenSSL (verify/s)	72.8k	72.7k	+0.10%
Redis (req/s)	1842k	1876k	-1.79%
7-Zip Compress (MIPS)	42.3k	42.4k	-0.24%
7-Zip Decompress (MIPS)	34.4k	34.4k	-0.03%
LMbench			
Context Switch (μs)	31.97	32.03	+0.17%
Read Syscall (μs)	0.313	0.318	+1.59%
Write Syscall (μs)	0.316	0.318	+0.49%
Stat Syscall (μs)	1.842	1.829	-0.68%
Memory Read (MB/s)	20.1k	20.4k	-1.59%
Unix Socket (MB/s)	3.87k	3.95k	-1.88%
File Read (MB/s)	3.29k	3.34k	-1.28%

As shown in Table 8, our mitigation introduces negligible runtime overhead across all benchmarks. This is because our mitigation affects only the tag fault handling path, which is not invoked during normal execution. In macro-benchmarks, performance deviations range from -1.79% on Redis to +0.10% on OpenSSL, falling within run-to-run variation. Likewise, micro-benchmarks exhibit similar variations ranging from -1.88% on Unix Socket to +1.59% on Read Syscall. Overall, our mitigation hardens asynchronous MTE against MTEscape while preserving its performance benefit.

7 Discussion

Responsible Disclosure. We have responsibly reported our findings to the Android Security Team. Our report included a detailed description of the technique with a working exploit and a kernel patch implementing the proposed mitigation. The Android Security Team acknowledged the issue as a security vulnerability and awarded a bug bounty. Our disclosure timeline provided more than 90 days between the initial report and the earliest publication date.

Bypass via TFSR_EL1 Manipulation. In asynchronous mode, tag check results are asynchronously delivered to the `TFSR_EL1` register. Thus, manipulating `TFSR_EL1` could serve as a potential vector to bypass MTE. However, we conclude that this approach is impractical. Since `TFSR_EL1` is a system register, we need to execute specific instructions to manipulate it. For that, we first need to perform control flow hijacking (CFH)—either forward or backward. However, under modern ARM64 kernel mitigations, this is extremely difficult, as KCFI mitigates forward CFH and PAC mitigates backward CFH. Therefore, bypassing MTE via `TFSR_EL1` manipulation requires circumventing these additional mitigations, which introduces further challenges.

Limitation on Double-Free Vulnerabilities. Unlike other exploitation techniques [10, 21, 27], MTEscape is not applicable to double-free vulnerabilities. This is because the majority of modern exploitation techniques attack double-free vulnerabilities by pivoting them into UAFs. However, this pivoting is impossible in an

MTE-enabled kernel. Notably, the Linux kernel explicitly checks an object's tag during deallocation, even in asynchronous mode. Specifically, the kernel uses the LDG instruction to load the tag and compare it with the expected value. Consequently, there is no race window for exploitation, making MTEscape infeasible for double-free vulnerabilities.

Asymmetric Mode. MTEscape remains partially applicable even under MTE's asymmetric mode. The asymmetric mode, introduced in ARMv8.7-A [45], supports heterogeneous checking for memory operations: synchronous checking for memory reads and asynchronous checking for memory writes. Under this mode, if we apply MTEscape to OOB write vulnerabilities, we can overwrite the kill switch variables before tag fault handling, making MTEscape still feasible. However, for UAF vulnerabilities, we must read a corrupted object before an arbitrary write. Thus, the kernel will detect a tag fault and trigger a panic during this read operation. In summary, MTEscape can work even under the asymmetric mode, but it will be limited to OOB write vulnerabilities.

8 Related Work

Kernel MTE Bypass. A limited number of exploits [32, 33, 47] have been shown to bypass MTE in the Linux kernel. These exploits rely on invalid writes performed by coprocessors, which MTE does not check. As these exploits depend on such specialized capabilities, they do not constitute a general bypass method for the majority of kernel memory corruption vulnerabilities. TikTag [17] demonstrated that transient execution gadgets can leak MTE tags via speculative tag checking. However, none of the discovered gadgets can be triggered from userspace without kernel modification, substantially limiting its practicality. Unlike these works, MTEscape bypasses asynchronous kernel MTE using conventional OOB write and UAF vulnerabilities, without relying on specialized capabilities.

Kernel Exploitation Techniques. Several techniques have been proposed to transform memory corruption bugs into arbitrary write primitives or privilege escalation exploits. KOOBE [5] automates the construction of OOB write exploits by converting them into UAF primitives and subsequently performing control flow hijacking. SLUBStick [27] elevates constrained heap vulnerabilities into arbitrary memory read-write primitives via cross-cache page table corruption. BridgeRouter [46] automates locating and manipulating kernel objects to refine an uncontrolled OOB write into a controlled one, and ultimately into an arbitrary write. These approaches are infeasible under asynchronous MTE protection, which prevents the multi-step pivoting they require.

Other approaches reclaim freed objects with attacker-controlled data to achieve stronger primitives. Cross-cache attacks [16, 31] broaden such reclaiming across slab boundaries by returning the backing pages of a freed object to the page allocator. Leveraging this technique, DirtyCred [21] achieves privilege escalation by freeing the attacker's credential and reclaiming it with a privileged one, while Page Spray [10] sprays attacker-controlled pages to overwrite dangling objects. In both techniques, overwriting the dangling object through reclaiming does not trigger a tag fault, since reclaiming assigns a new valid tag. However, any subsequent access through the dangling pointer triggers a probabilistic tag fault, making these techniques infeasible for an MTE-enabled kernel.

Race Stabilization. Several techniques have been proposed to stabilize the race condition for exploitation. FUSE [22] and userfaultfd [23] control the race window by pausing kernel execution through userspace handlers [21, 35, 37]. EXPRACE [20] utilizes interrupts to extend the race window of kernel exploits. These techniques cannot be applied to extend the race window of asynchronous tag fault handling, since tag checks occur on every kernel exit, including interrupts and return to userspace. CARDSHARK [12] stabilizes race reproducibility by introducing invocation delays to align race points across threads. MTEscape builds on CARDSHARK to stabilize race exploits and further improves it by introducing cache training.

9 Conclusion

In this paper, we presented MTEscape, a novel technique that bypasses asynchronous kernel MTE using conventional OOB write and UAF vulnerabilities. While these vulnerabilities trigger tag mismatches, MTEscape exploits the race window of asynchronous fault handling to bypass MTE before the kernel handles the violation. Specifically, MTEscape disables tag fault handling by corrupting kill switch variables, using DWOs for single-step pivoting and race stabilization for single-attempt reliability. MTEscape effectively bypasses asynchronous MTE on synthetic vulnerabilities and on 10 real-world Linux kernel CVEs. To address this threat, we proposed a mitigation that protects kill switch variables with negligible performance overhead. Our findings suggest that asynchronous kernel MTE can provide its intended protection through careful software implementation.

Acknowledgments

We thank the anonymous reviewers for providing valuable feedback. This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No. RS-2026-25509873, Development of a Simulator for Indicator-Based Real-Time Cyber Attack Defense Assessment and Life-Cycle Autonomous Security) and by Samsung Electronics Co., Ltd. (IO251218-14782-01).

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009). https://doi.org/10.1145/1609956.1609960

[2] Apple Security Engineering and Architecture. 2025. Memory Integrity Enforcement. https://security.apple.com/blog/memory-integrity-enforcement/. Accessed: 2026-04-29.

[3] Arm. [n.d.]. Introduction to the Memory Tagging Extension. https://developer.arm.com/documentation/108035/0100/Introduction-to-the-Memory-Tagging-Extension. Accessed: 2026-04-29.

[4] Mark Brand. 2023. MTE As Implemented, Part 3: The Kernel. https://googleprojectzero.blogspot.com/2023/08/mte-as-implemented-part-3-kernel.html. Accessed: 2026-04-29.

[5] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[6] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*. Virtual. https://doi.org/10.1145/3372297.3423353

[7] Chromium Project. [n.d.]. PartitionAlloc. https://chromium.googlesource.com/chromium/src/+/main/base/allocator/partition_allocator/PartitionAlloc.md. Accessed: 2026-04-29.

[8] corbet. 2007. The SLUB allocator. https://lwn.net/Articles/229984/. Accessed: 2026-04-29.

[9] Vincenzo Frascino. 2021. arm64: ARMv8.5-A: MTE: Add async mode support. https://lwn.net/Articles/844750/. Accessed: 2026-04-29.

[10] Ziyi Guo, Dang K Le, Zhenpeng Lin, Kyle Zeng, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé, and Xinyu Xing. 2024. Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA.

[11] Cedric Halbronn. 2022. Settlers of Netlink: Exploiting a limited UAF in nf_tables (CVE-2022-32250). https://www.nccgroup.com/research-blog/settlers-of-netlink-exploiting-a-limited-uaf-in-nf_tables-cve-2022-32250/. Accessed: 2026-04-29.

[12] Tianshuo Han, Xiaorui Gong, and Jian Liu. 2024. CARDSHARK: Understanding and Stabilizing Linux Kernel Concurrency Bugs Against the Odds. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA.

[13] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[14] Hyerean Jang, Taehun Kim, and Youngjoo Shin. 2024. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)* (Salt Lake City, U.S.A., October 14–18, 2024). Salt Lake City, U.S.A. https://doi.org/10.1145/3658644.3690189

[15] Seth Jenkins. 2025. Defeating KASLR by doing nothing at all. https://projectzero.google/2025/11/defeating-kaslr-by-doing-nothing-at-all.html. Accessed: 2026-04-29.

[16] Dong-ok Kim, Juhyun Song, and Insu Yun. 2025. CROSS-X: Generalized and Stable Cross-Cache Attack on the Linux Kernel. In *Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS)* (Taipei, Taiwan, October 13–17, 2025). Taipei, Taiwan. https://doi.org/10.1145/3719027.3765152

[17] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. 2025. Tiktag: Breaking ARM's Memory Tagging Extension with Speculative Execution. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA. https://doi.org/10.1109/SP61157.2025.00039

[18] Andrey Konovalov. 2020. kasan: add hardware tag-based mode for arm64. https://lwn.net/Articles/831624/. Accessed: 2026-04-29.

[19] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. 2023. Pspray: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA.

[20] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual.

[21] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA. https://doi.org/10.1145/3548606.3560585

[22] Linux. [n.d.]. FUSE. https://www.kernel.org/doc/html/next/filesystems/fuse.html. Accessed: 2026-04-29.

[23] Linux. [n.d.]. Userfaultfd. https://www.kernel.org/doc/html/next/admin-guide/mm/userfaultfd.html. Accessed: 2026-04-29.

[24] Danjun Liu, Xuan Meng, Pengfei Wang, Xu Zhou, Wei Xie, and Baosheng Wang. 2025. Constructing arbitrary write via puppet objects and delivering gadgets in Linux kernel. *Computers & Security* 150 (2025), 104189. https://doi.org/10.1016/j.cose.2024.104189

[25] William Liu, Joseph Ravichandran, and Mengjia Yan. 2023. EntryBleed: A Universal KASLR Bypass against KPTI on Linux. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy* (Toronto, Canada) *(HASP '23)*. Association for Computing Machinery, New York, NY, USA, 10–18. https://doi.org/10.1145/3623652.3623669

[26] LLVM Project. [n.d.]. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html. Accessed: 2026-04-29.

[27] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. 2024. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA.

[28] Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. 2025. When good kernel defenses go bad: reliable and stable kernel exploits via defense-amplified TLB side-channel leaks. In *Proceedings of the 34th USENIX Security Symposium (Security)*. Seattle, WA.

[29] Mark Rutland. 2017. Pointer authentication in AArch64 Linux. https://docs.kernel.org/arch/arm64/pointer-authentication.html. Accessed: 2026-04-29.

[30] Larry McVoy and Carl Staelin. 1996. lmbench: portable tools for performance analysis *(ATEC '96)*. USENIX Association, USA.

[31] Claudio Migliorelli, Andrea Mambretti, Alessandro Sorniotti, Vittorio Zaccaria, and Anil Kurmus. 2026. Cross-Cache Attacks for the Linux Kernel via PCP Massaging. In *Proceedings of the 2026 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[32] Man Yue Mo. 2024. Gaining kernel code execution on an MTE-enabled Pixel 8. https://github.blog/security/vulnerability-research/gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/. Accessed: 2026-04-29.

[33] Man Yue Mo. 2025. Bypassing MTE with CVE-2025-0072. https://github.blog/security/vulnerability-research/bypassing-mte-with-cve-2025-0072/. Accessed: 2026-04-29.

[34] National Security Agency and Cybersecurity and Infrastructure Security Agency. 2025. Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development. https://media.defense.gov/2025/Jun/23/2003742198/-1/-1/0/CSI_MEMORY_SAFE_LANGUAGES_REDUCING_VULNERABILITIES_IN_MODERN_SOFTWARE_DEVELOPMENT.PDF. Accessed: 2026-04-29.

[35] Vitaly Nikolenko. 2018. Linux Kernel universal heap spray. https://duasynt.com/blog/linux-kernel-heap-spray. Accessed: 2026-04-29.

[36] Taehyun Noh, Yingchen Wang, Tal Garfinkel, Mahesh Madhav, Daniel Moghimi, Mattan Erez, and Shravan Narayan. 2026. ARM MTE Performance in Practice (Extended Version). arXiv:2601.11786 [cs.CR] https://arxiv.org/abs/2601.11786

[37] Valentina Palmiotti. 2022. Put an io_uring on it: Exploiting the Linux Kernel. https://chomp.ie/Blog+Posts/Put+an+io_uring+on+it+-+Exploiting+the+Linux+Kernel. Accessed: 2026-04-29.

[38] PaX Team. [n.d.]. PaX Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt. Accessed: 2026-04-29.

[39] Phoronix Media. [n.d.]. Phoronix Test Suite. https://www.phoronix-test-suite.com/. Accessed: 2026-04-29.

[40] Alexander Popov. 2021. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux Kernel. https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html. Accessed: 2026-04-29.

[41] Alexander Popov. 2025. Kernel-hack-drill and a new approach to exploiting CVE-2024-50264 in the Linux kernel. https://a13xp0p0v.github.io/2025/09/02/kernel-hack-drill-and-CVE-2024-50264.html. Accessed: 2026-04-29.

[42] Sami Tolvanen. 2022. KCFI support. https://lwn.net/Articles/893164/. Accessed: 2026-04-29.

[43] Theori Frontier Squad. 2025. Reviving the modprobe_path Technique: Overcoming search_binary_handler() Patch. https://theori.io/blog/reviving-the-modprobe-path-technique-overcoming-search-binary-handler-patch. Accessed: 2026-04-29.

[44] st424204 and d4em0n. 2025. CVE-2025-40364 Exploit for kernelCTF. https://github.com/google/security-research/blob/7f68f7f1b8dcdfb10098bbf457121ab3b3ddcedc/pocs/linux/kernelctf/CVE-2025-40364_lts_cos/docs/exploit.md. Accessed: 2026-04-29.

[45] Vincenzo Frascino. 2021. arm64: ARMv8.7-A: MTE: Add asymm mode support. https://lwn.net/Articles/869055/. Accessed: 2026-04-29.

[46] Dongchen Xie, Dongnan He, Wei You, Jianjun Huang, Bin Liang, Shuitao Gan, and Wenchang Shi. 2025. BridgeRouter: Automated Capability Upgrading of Out-Of-Bounds Write Vulnerabilities to Arbitrary Memory Write Primitives in the Linux Kernel. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA. https://doi.org/10.1109/SP61157.2025.00132

[47] Pan Zhenpeng and Jheng Bing Jhong. 2025. Cracking the Pixel 8: Exploiting the Undocumented DSP to Bypass MTE. HITCON 2025 Agenda. https://hitcon.org/2025/en-US/agenda/b7635c13-282e-4673-8297-43ed3550b3d3/. Accessed: 2026-04-29.

A Open Science

Our artifacts are available at https://github.com/kaist-hacking/MTEscape, covering all of our evaluations: exploiting synthetic vulnerabilities (§5.1), exploiting real-world kernel vulnerabilities (§5.2), race stabilization performance (§5.3), and the runtime overhead of our proposed mitigation (§6.2).

B Ethical Considerations

We conducted our experiments on devices in a controlled local environment, causing no harm to real-world systems. We provide details regarding responsible disclosure in §7.

C Generative AI Usage

We used Claude for partial proofreading and polishing and Claude Code to assist in preparing the research artifact. We manually verified all AI-assisted material. All evaluations were conducted through actual experiments without LLM involvement.