USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

DMGuard: Safeguarding Kernels from Physical-Page Use-After-Free Vulnerabilities

Juhee Kim, Jaeyoung Chung, Dae R. Jeong, and Byoungyoung Lee, *Seoul National University*

https://www.usenix.org/conference/usenixsecurity26/presentation/kim-juhee-dmguard

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

DMGUARD: Safeguarding Kernels from Physical-Page Use-After-Free Vulnerabilities

Juhee Kim*
Seoul National University

Jaeyoung Chung*
Seoul National University

Dae R. Jeong†
Seoul National University

Byoungyoung Lee†
Seoul National University

Abstract

Modern kernels depend on the integrity of page tables to enforce advanced security measures. Although these defenses have effectively mitigated various attacks including memory corruption, adversaries have shifted their focus to compromise the page table itself to bypass existing protections. Such threats are exacerbated by the rise of heterogeneous address translation domains including separate CPU, GPU, and IOMMU page tables, which impose heavy demands on synchronization and coherence management. When a virtual address remains mapped to a physical page that has already been freed or reallocated, attackers can exploit this to access arbitrary physical memory. We call this physical-page use-after-free, distinct from traditional heap use-after-free that operates on virtual addresses.

In this paper, we present DMGUARD, the first runtime mitigation that comprehensively addresses physical-page use-after-free vulnerabilities across diverse translation domains. DMGUARD leverages a lightweight, lockless mechanism to manage a state machine of physical pages to ensure no dangling mappings exist in the page tables. Evaluation of DMGUARD on Android devices demonstrates that it effectively blocks all known physical-page use-after-free vulnerabilities with negligible performance overheads, demonstrating the practicality and effectiveness against emerging attacks.

1 Introduction

The integrity of the page table lies at the heart of modern kernel security. As a result of extensive efforts in fortifying the kernel, advanced protection techniques, such as address space randomization [28], control- [13, 50, 60] and data-flow integrity [44, 73, 81], and hardware security features [19, 26, 39, 75] have been developed to defend against malicious attackers. These mechanisms fundamentally rely on correct virtual-to-physical address mappings and page-level access control enforcement in the page tables. Together with page table correctness, defense mechanisms have proven effective at reducing attack surface and the risk of various attacks [53].

However, attackers have developed increasingly sophisticated threats that undermine the integrity of the page table [33, 65, 74, 87]. When the kernel fails to correctly manage the page table, for instance by not revoking mappings of freed physical pages, a virtual address might remain mapped to a physical page that has already been freed or reallocated. This allows an attacker to access the physical page after it has been freed.

We call this *physical-page use-after-free* vulnerability, which is fundamentally distinct from traditional page-granularity heap use-after-free [33]. While heap use-after-free involves dangling pointers in virtual address space, physical-page use-after-free occurs in the address translation layer, where virtual-to-physical mappings in page tables outlive the physical pages they reference. Once exploited, it allows an attacker to place attacker-controlled physical pages in the virtual address space, effectively bypassing nearly all mitigation techniques as they rely on page tables now compromised by the attacker.

Making matters worse, modern systems employ separate translation structures for specialized accelerators and peripherals, including CPU and GPU page tables [70], devices with own page tables (*e.g.*, smart NICs [69]), and IOMMU page table [38] [1]. These structures are managed by independent components such as the kernel, device drivers, or even user processes [18]. Moreover, performance-oriented designs tightly couples page management across components (*e.g.*, zero-copy memory sharing [68]) and further increases coordination complexity. We think this complexity has led to frequent mistakes in page table management, resulting in a surge of vulnerabilities in last few years [1, 7, 64, 65, 87], allowing attackers to exploit them to bypass all mitigations deployed on production systems.

This work introduces a principled approach to securely

*These authors contributed equally to this work.
†Corresponding authors

[1] Unless otherwise stated, we use the terms *page table* to denote any type of address translation table, and *page* to denote a physical page.

manage page tables across these diverse and independent components. We observe that page use-after-free vulnerabilities stem from *incorrect management of a page's life cycle*. A correct implementation should ensure that a mapping never outlives its backing page—*i.e.*, the mapping should be established only after the page is allocated and removed before the page is freed. If this temporal-ordering is violated, a dangling mapping remains in the page table, leading to page use-after-free. Our insight is that the current independent page table management across various components complicates this life cycle management, resulting in frequent errors in ensuring the correct temporal-ordering.

Following this insight, we present DMGUARD, a lightweight runtime solution that detects and prevents dangling page mappings and physical-page use-after-free across diverse components. DMGUARD models a state machine that captures the lifecycle of every physical page (*e.g.*, allocation, mapping, and freeing). It then monitors each page's state during runtime to identify invalid state transitions—such as when a page is freed while it is still mapped—enabling proactive prevention of physical-page use-after-free. DMGUARD is integrated with minimal changes into kernel APIs that manage CPU page tables (*e.g.*, `set_pte_at()`) as well as device drivers managing other non-CPU address translation structures, such as GPU page tables and IOMMU page tables.

A notable advantage of DMGUARD is its precise, lockless mechanism for tracking state transitions, which enables immediate detection of dangling mappings without heavy synchronization on multiple page tables. Because page table management is both performance-sensitive and security-critical, this design of DMGUARD imposes minimal runtime overhead while immediately detecting the incorrect page table management (*e.g.*, a CPU frees a page while a mapping in a GPU page table remains), completely stopping the physical-page use-after-free attacks.

We confirmed the correctness of the lockless design against the Linux Kernel Memory Model (LKMM) [11, 83], showing that DMGUARD is able to detect dangling mappings in all architectures that the Linux kernel supports (*e.g.*, x86 or ARM).

We have implemented DMGUARD in Linux kernels (*i.e.*, v4.9, v6.1, and v6.6) and on Android devices (*i.e.*, Pixel 8 and Pixel 3 XL). Our evaluation demonstrates that DMGUARD effectively detects and blocks 24 physical-page use-after-free vulnerabilities in managing CPU, GPU, and IOMMU page tables. Additionally, we show that DMGUARD introduces low runtime overheads (approximately 1% decrease in GeekBench), while maintaining compatibility with existing kernel defenses.

We summarize the contributions of this work in threefold:

- To the best of our knowledge, DMGUARD is the first mitigation that prevents dangling mapping and physical-page use-after-free vulnerabilities across diverse page tables.
- We show that DMGUARD's lockless mechanism incurs the negligible overhead while correctly working on all architectures that the Linux kernel supports.
- We implement and evaluate DMGUARD on Android devices, demonstrating its effectiveness in successfully preventing physical-page use-after-free vulnerabilities.

We open sourced DMGUARD in `https://doi.org/10.5281/zenodo.17970502` to secure modern operating systems against physical-page use-after-free vulnerabilities and to facilitate further research.

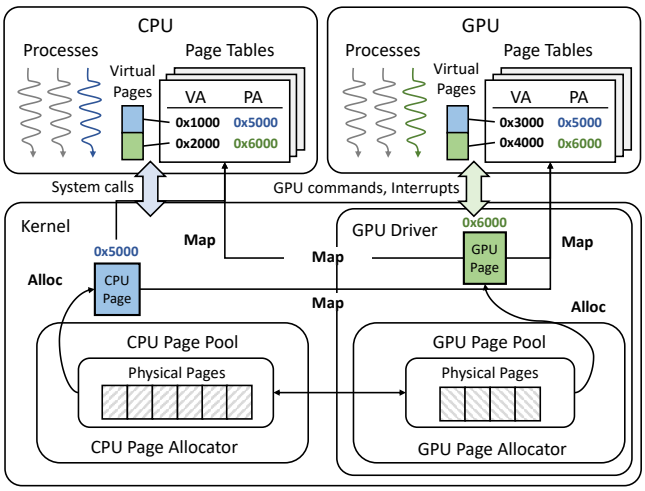


Figure 1: A simplified view of CPU and GPU memory management, where each maintains its own page tables and allocation pools. When the GPU needs memory, it may allocate a page from its pool (*i.e.*, green box at physical address `0x6000`) and map it into its page table. The CPU can then also create a mapping to the same page to access GPU-processed data. Conversely, the CPU may allocate a page (blue box at `0x5000`), which both CPU and GPU can map for shared access.

2 Motivation

We first explain the page management in the Linux kernel (§2.1) and how it can be ill-managed, resulting in dangling mappings, which eventually lead to physical-page use-after-free vulnerabilities (§2.2). Then, we illustrate real-world examples of vulnerabilities (§2.3) to motivate our approach.

2.1 Complexity in Page Management

Figure 1 illustrates the complex page table management with a CPU and an integrated GPU, such as ARM Mali or Qualcomm Adreno GPUs commonly found in mobile devices. These integrated GPUs share system DRAM with the CPU without dedicated video memory (VRAM), creating a unified

physical memory architecture. Both processing units have their own page tables, referenced by their own memory management units (MMU). Integrated GPUs have their own MMU or leverage existing IOMMU to access the shared physical memory, resulting in multiple heterogeneous translation domains operating over the same physical address space. CPUs and GPUs also have separate page pools, coordinated by the kernel and a dedicated device driver. To avoid the high cost of copying data between different physical pages, the kernel and the device driver often map the same physical page into the page tables of both processing units, allowing them to access the data directly.

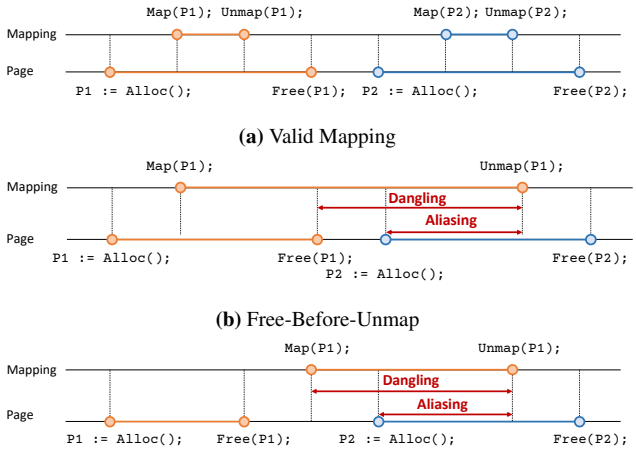
Consider a typical scenario: the GPU driver maintains its own page pool by pre-allocating pages from the CPU's buddy allocator (*e.g.*, using `alloc_pages()`). When the GPU needs memory for its operations, it allocates pages from this dedicated pool and maps them into the GPU's page table. Later, when the CPU needs to access GPU-processed data, the kernel creates CPU mappings to these GPU-allocated pages in the CPU's page table. Through these dual mappings, both the CPU and GPU can access the same data without having a cost to copy the data into the other's address space. Conversely, the CPU may allocate pages and the GPU driver may create GPU mappings to these CPU-allocated pages in the GPU's page table.

While sharing pages across computing units provides significant performance benefits, this creates burden of correctly managing a page's lifecycle and subtle opportunities for error. For example, allocation may be handled by one component (*e.g.*, the GPU device driver), while mappings are created and destroyed by another (*e.g.*, the kernel). If the correct sequence of operations is not rigorously enforced across all components, the device driver may free a page while it is still mapped in the CPU's page table, resulting in inconsistent mappings between the two page tables.

2.2 Dangling Mapping Causing Page Use-After-Free

A *dangling mapping* is a mapping that remains in a page table after the page it refers to has been freed. If a processor accesses memory through the dangling mapping, it accesses a page that has been freed, which is called a *physical-page use-after-free* vulnerability [65, 74, 87]. Preventing dangling mapping is challenging because multiple partially independent page tables (*i.e.*, CPU, GPU and IOMMU page tables) must be orchestrated, along with the lifecycles of physical pages.

Figure 2 contrasts the normal case with a potential dangling mapping. In the normal path (Figure 2a), a mapping should never outlive a physical page allocation. The kernel allocates the page, maps it into the page table. Later when the page is no longer needed, the kernel first unmaps the entry and only then returns the page to the allocator. However, if pages and



(a) Valid Mapping

(b) Free-Before-Unmap

(c) Map-After-Free

Figure 2: Page allocation and mapping timeline of valid and invalid cases.

mappings are not managed properly, a dangling mapping may occur commonly in two scenarios.

Scenario 1: Free-Before-Unmap. When a page is freed while it is still mapped in a page table, existing mapping becomes stale, which we call a *free-before-unmap* vulnerability (Figure 2b).

Scenario 2: Map-After-Free. If a mapping is created after a page is freed, the mapping is invalid, leading to a vulnerability that we refer to a *map-after-free* (Figure 2c).

Physical-Page Use-After-Free Vulnerability. A physical-page use-after-free vulnerability occurs when a processor *accesses* a page with the dangling mapping. If the accessed page has been reallocated (*i.e.*, `P2` in Figure 2b and Figure 2c), accesses would target memory that has been occupied by different content (*i.e.*, the *Aliasing* period in Figure 2). With page spraying techniques [33], attackers can deliberately steer the allocator to reassign the page that dangling mapping points to. This can create a direct alias to the attacker-controlled content, such as kernel page tables or credential structures, allowing immediate privilege escalation. Even worse, state-of-the-arts kernel protections (*e.g.*, CFI [13, 50, 60], Intel MPK [26], ARM PAC [75], and MTE [19]) cannot prevent this because they assume correct address translation in the page table.

Challenges in Page Management. Correctly managing page lifecycles is challenging due to two fundamental reasons in modern kernels. First, systems with multiple different page tables distribute lifecycle management across independent subsystems. Each subsystem (*e.g.*, GPU drivers) manages its own page allocation pools and mappings, separated from the core memory management subsystem, making coordinated lifecycle enforcement difficult. Furthermore, page management resides in a performance-sensitive path where synchronization overhead across different subsystems would significantly

```
1  // mmap handler
2  unsigned long do_mmap()
3  {
4      // T1: Reserve VA1
5      // T3: Reserve VA2
6      unsigned long vaddr = kbase_get_unmapped_area();
7      // T2: Allocate P1 and store its PA1 in queue->phys
8      // T4: Allocate P2 and store its PA2 in queue->phys
9      kbase_mem_pool_alloc_pages(queue->phys);
10     return vaddr;
11 }
12 // page fault handler
13 static vm_fault_t kbase_csf_user_io_pages_vm_fault()
14 {
15     // T5: Map P2 to the userspace (VA2->PA2)
16     input_page_pfn = PFN_DOWN(as_phys_addr_t(queue->phys[0]));
17     vmf_insert_pfn_prot(input_page_pfn);
18 }
19 // munmap handler
20 static int do_munmap(unsigned long vaddr)
21 {
22     // T6: Remove CPU mapping on VA if exists
23     unmap_region(vaddr);
24     // T7: Free the GPU page in queue->phys
25     kbase_mem_pool_free_pages(queue->phys);
26 }
```

Figure 3: Simplified code snippet of CVE-2025-0072. T, VA, and P are timestamp of events, virtual address, and individual page, respectively.

degrade system performance.

Second, the kernel provides low-level mapping interfaces that bypass standard page management mechanisms (*e.g.*, reference counting). PFN-based mappings (*e.g.*, `VM_PFNMAP`) exemplify this issue. They use raw Page Frame Numbers (PFNs) instead of page descriptors, intentionally avoiding reference count updates to support pages not managed by the kernel buddy allocator (*e.g.*, MMIO region). However, device drivers often use PFN-based mappings for pages from buddy allocator, creating a risk of dangling mappings when pages are freed without considering active PFN mappings.

2.3 Real-World Example

To demonstrate physical-page use-after-free vulnerabilities, we analyze a real-world vulnerability in the Mali GPU driver (CVE-2025-0072) [65]. Mali GPU is a widely used mobile GPU, and its kernel driver manages its own page allocator and handles mappings in both CPU and GPU page tables. In the vulnerable version, the driver fails to properly remove the CPU mapping when freeing a GPU page, allowing userspace access through the dangling CPU mapping even after the page has been freed, leading to a free-before-unmap vulnerability.

Figure 3 illustrates the vulnerability. The Mali GPU driver provides a `mmap()` interface for userspace to allocate GPU command buffers and map them into the CPU's address space. The `mmap` handler first reserves a CPU virtual address (`vaddr`) (line 6). Then, it allocates a page from the GPU page allocator and stores its physical address (PA) in `queue->phys` (line 9), which it returns to userspace (line 10). The actual CPU mapping is created lazily: when the user process accesses `vaddr`, a page fault occurs and the driver creates a CPU mapping from `vaddr` to the GPU page pointed by `queue->phys` (line 17). When `munmap()` is called, the kernel removes the CPU mapping if it exists (line 23), then the driver frees the GPU page pointed by `queue->phys` (line 25).

The vulnerability occurs when a GPU page is allocated twice. During the first `mmap()` (T1-T2), the driver reserves a virtual address VA1 and allocates a GPU page P1, storing its physical address PA1 in `queue->phys`. The second `mmap()` (T3-T4) reserves VA2 and allocates page P2, overwriting `queue->phys` with PA2. When the process accesses VA2, the page fault handler installs mapping `VA2` $\rightarrow$ `PA2` (T5). Later, when `munmap()` is called on VA1, the driver does not remove any mapping, since VA1 was never mapped (T6). However, the driver frees P2 as it is pointed by `queue->phys` (T7), leaving a dangling CPU mapping `VA2` $\rightarrow$ `PA2` and resulting in a free-before-unmap vulnerability.

3 Threat model

We assume a modern Linux kernel that incorporates state-of-the-art in-kernel defenses, including data execution prevention [59], control- [50, 89] and data-flow integrity [44], security-sensitive data protection [78], kernel address sanitizer [77, 79, 82], and page-table hardening features (*e.g.*, Intel VT-rp [46] or Apple SPTM [17]). We also assume that the underlying hardware (*e.g.*, CPU, MMU, and GPU) is fully trusted.

We assume a strong local attacker who runs an unprivileged user process and invokes arbitrary system calls. The system calls may create, modify, or delete page mappings in the user page table. We also assume that the attacker has access to integrated GPUs and can run arbitrary GPU kernels from a user-space GPU process, which is a common threat model in untrusted mobile applications [14, 35].

We focus on physical-page use-after-free vulnerabilities, where virtual-to-physical address mappings in page tables outlive the physical pages they reference. This is distinct from traditional heap use-after-free, which involves dangling virtual pointers within a process's address space. Our threat model addresses the integrity of address translation, not virtual memory allocators.

Under these assumptions, we focus particularly on mappings in *user page tables*, which defines user-privileged process's view of memory. A dangling mapping in a user page table immediately grants the user process an arbitrary physical memory access, only with memory instructions such as load or store. Therefore, preventing dangling user space mappings reduces a critical attack surface. Protecting the integrity of higher-privilege level page tables (e.g., kernel page tables) is also critical, but exploiting them from the user space often requires additional complexity (*e.g.*, finding a specific system call that can access specific kernel addresses). We also ex-

clude side-channel and speculative-execution vulnerabilities from our scope.

Lastly, DMGUARD assumes a unified physical memory architecture where CPU and integrated GPUs share the same physical memory space. This architecture is common in mobile and embedded systems, where integrated GPUs access host memory through either their own MMU or through the system IOMMU. We do not consider the case where the target system has dedicated discrete GPUs with their own physical memory (e.g., NVIDIA GPUs and AMD GPUs), although DMGUARD could be extended to support such scenarios (§7).

4 DMGUARD

While correctly managing multiple page tables is complicated and error-prone, we observe that 1) each page's lifecycle can be modeled as a manageable state machine (§4.1), and 2) dangling mappings happen when an invalid state transition occurs (§4.2). Based on this observation, we present DMGUARD, a lightweight runtime mechanism that manages page lifecycle across diverse page tables (§4.3).

Goals. We design DMGUARD with mainly two goals. First, considering the severity of such vulnerabilities, DMGUARD is to detect them instantly and prevent any access through dangling mappings without missing any vulnerable cases across various processors and page tables in the system. Second, memory management often lies on performance-critical paths, and thus DMGUARD must impose minimal runtime overhead to maintain system performance while providing comprehensive protection.

To achieve these goals, DMGUARD tracks state transitions of each page through a lightweight, lockless design that integrates seamlessly with existing kernel page management logic, ensuring both complete coverage and high performance.

4.1 Page Lifecycle as a State Machine

In Figure 4, we describe the lifecycle of a page using a simplified state machine, where each state captures the allocation and mapping statuses. This state machine abstracts the page status into four key states, simplifying the complexity of page management across diverse kernel components. The fundamental invariant is that a page should be mapped only when it is allocated, and violating this invariant leads to dangling mapping.

Freed-Unmapped. In the initial state (*i.e.*, *Freed-Unmapped*), a page resides in a page pool of an allocator (e.g., kernel buddy allocator or GPU driver's page allocator). The page is not mapped in any page table and is inaccessible from any virtual address. Upon a page allocation request (*e.g.*, thorugh `alloc_pages()`), the page allocator selects a free page from the pool and returns it, where the page transitions to the *Allocated-Unmapped* state.

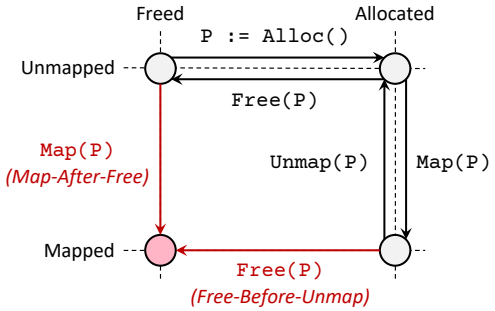


Figure 4: State machine illustrating the lifecycle of a single page. Each state encodes both allocation and mapping status (*e.g.*, the upper-right denotes a page that is allocated but not mapped in any page table). Red arrows indicate operations that lead to a vulnerable state (*i.e.*, dangling mapping).

Allocated-Unmapped. In this state, the page has been removed from the page pool, but has not yet been mapped into any page table. Thus, it remains non-addressable, although the kernel retains a reference to the page descriptor (*i.e.*, `struct page *`) to maintain its metadata and to coordinate subsequent mapping or freeing operations.

Allocated-Mapped. The page enters the *Allocated-Mapped* state when the kernel creates a mapping between a virtual address and the page. This mapping is established by updating a page table entry (PTE) in the page table of the target processor (*e.g.*, CPU or GPU), which is done by invoking a mapping function (*e.g.*, `set_pte_at()` for CPU page tables). At this point, the page's physical address is associated with the virtual address, making it accessible to the process that owns the page table. The kernel may create multiple mappings for the same page, each associated with a different virtual address in the same or different page tables (*e.g.*, sharing a page between CPU and GPU, or different user processes).

All the Way Back to Freed-Unmapped. Freeing a page can be seen as the reverse path through the page's lifecycle. When the kernel removes all page-table entries associated with the page in the *Allocated-Mapped* state, the page transitions back to the *Allocated-Unmapped* state. The page is no longer accessible by any user process but remains allocated. Finally, when the kernel explicitly releases the page by passing the page descriptor to a page freeing function (*e.g.*, `__free_pages()` for CPU-allocated pages), it returns to the *Freed-Unmapped* state and is added back to the allocator's pool. The page is then available for future allocations and can re-enter a new lifecycle as needed.

4.2 Twofold State Transition Tracking

The core of DMGUARD is to encode the state machine described in §4.1 into the page management logic, and while monitoring, immediately flag any dangling mapping caused by an incorrect state transition.

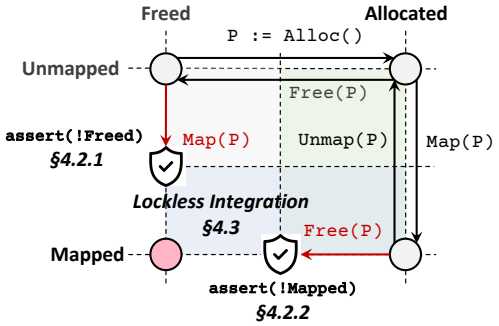


Figure 5: Overview of DMGUARD's page state transition tracking.

Figure 5 illustrates the simplified state machine that DMGUARD leverages to track each transition of a page state. In this state machine, DMGUARD adopts two lightweight primitives for different types of state transitions. First, it maintains a *per-page mapping count* to describe state transitions between the *Mapped* and *Unmapped* states (*i.e.*, the vertical arrows in Figure 5). When a page is freed, DMGUARD leverages the mapping count; if it is nonzero–meaning the page is still marked *Mapped*–it reports the free-before-unmap vulnerability (§4.2.1). Second, it employs a *per-page tag* to express transitions between the *Freed* and *Allocated* states (*i.e.*, the horizontal arrows). When a page is being mapped, DMGUARD inspects its tag; if it indicates *Freed* state instead of *Allocated*, it reports a map-after-free vulnerability (§4.2.2).

Together, these two primitives form a unified state machine, and DMGUARD integrates them to provide a consistent view across all page tables without resorting to expensive synchronization primitives (*e.g.*, locks) or extra monitoring, thereby avoiding heavy runtime overhead without lossing its protection ability (§4.3).

4.2.1 Tracking Mapping Status with Reference Counter

Tracking state transitions between *Mapped* and *Unmapped* should accommodate multiple mappings from different processors' page tables. To do this, DMGUARD leverages a `MapCount`, a reference counter-style mechanism that counts the number of user-space mappings across different page tables given a physical page. DMGUARD focuses on user-space mappings because they present the most immediate attack surface, as explained in §1. When a user-space page mapping is created, the page's `MapCount` is incremented, and when a mapping is removed, `MapCount` is decremented. A non-zero `MapCount` indicates a *Mapped* state, while a zero value signifies an *Unmapped* state. To avoid free-before-unmap, the `MapCount` should be zero when a page is freed, guaranteeing it is not mapped in any user-space page table. DMGUARD stores `MapCount` in per-page metadata, which we detail in §5.

This approach is inspired by `CONFIG_PAGE_TABLE_CHECK` [51], which manages `MapCount`

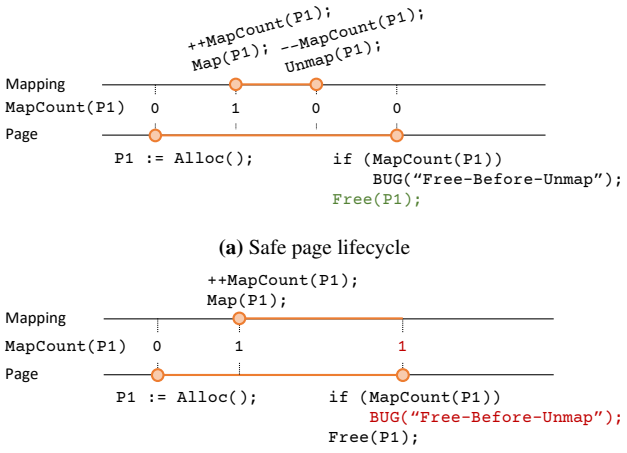


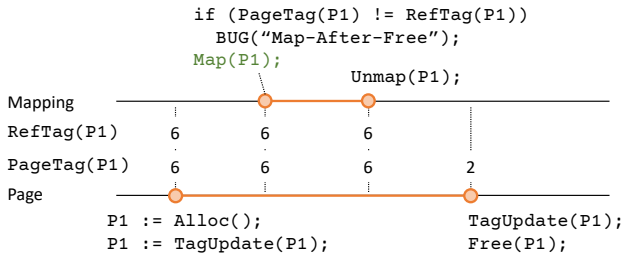
Figure 6: Mapping count to prevent free-before-unmap. `P1` and `P2` are page structure pointers (i.e., `struct page*`) to the same physical page.

in CPU-only context. DMGUARD extends this mechanism to support for page operations across various processors that were previously ignored. For instance, DMGUARD accounts for mappings in GPU page tables by properly updating `MapCount` when a page is mapped or unmapped in a GPU page table. This GPU-side `MapCount` is checked when the page is freed either from the GPU page allocator or from the CPU page allocator. Filling this gap in page management logic is crucial, as modern processors increasingly offload tasks to special-purpose computing units (*e.g.*, GPU, NPU, and SmartNIC) that access host physical memory through their own page tables or IOMMU-managed page tables.

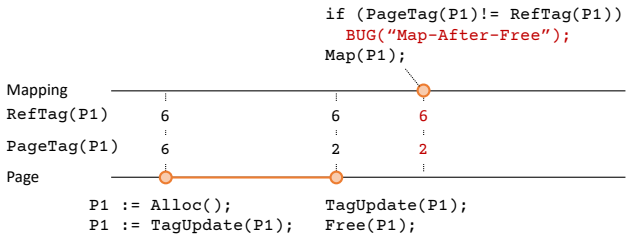
Detecting Free-Before-Unmap. Figure 6 illustrates how DMGUARD manages `MapCount` for a page throughout its lifecycle. First, in a safe lifecycle (Figure 6a), the initial `MapCount(P1)` is zero, indicating that the page is not mapped to any user process. When the page `P1` is mapped, `MapCount(P1)` is incremented to 1. When `P1` is unmapped, `MapCount(P1)` is decremented back to zero, and once `P1` is freed, `MapCount(P1)` confirms that the page is not mapped, allowing the page to be safely freed.

In contrast, free-before-unmap case (Figure 6b) is detected by `MapCount`. Here, the page `P1` is freed while `MapCount(P1)` remains greater than zero, violating the invariant that `MapCount` must be zero at the time of page reclamation. DMGUARD detects this violation and prevents the page from being freed, thus avoiding free-before-unmap.

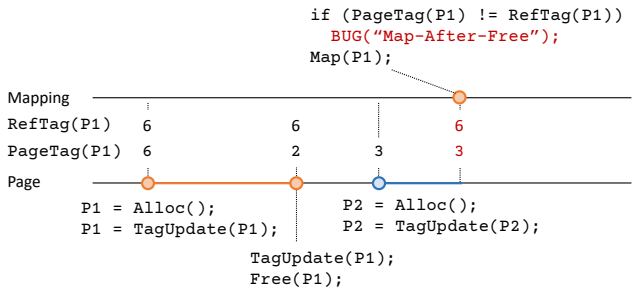
In the case of `CVE-2025-0072` [65] described in §2.3, a page allocated by the Mali GPU page allocator is mapped into the CPU page table, but is freed without being unmapped. As a result, a dangling mapping remains in a user page table, allowing malicious users to access the freed page and exploit a page-use-after-free vulnerability. DMGUARD detects the



(a) Safe page lifecycle



(b) Map-after-free (map while freed)



(c) Map-after-free (map while remapped)

Figure 7: Page tagging to prevent map-after-free. `P1` and `P2` are page structure pointers (i.e., `struct page*`) to the same physical page.

dangling mapping by checking `MapCount(P1)` when the page is freed and returned to the GPU page allocator's page pool. Likewise, DMGUARD detects other free-before-unmap cases involving non-CPU page allocators or page tables.

4.2.2 Tracking Allocation Status with Page Tag

To track page allocation status, DMGUARD should track transitions between the *Freed* and *Allocated* states, and should also distinguish between different allocation cycles of a single physical page. For instance, when a page is freed, the page should not be mapped. Similarly, when the page is reallocated, page references from the previous allocation cycle must be invalidated to prevent unauthorized access. Without this temporal distinction, a stale reference from a previous lifecycle could be used to incorrectly map a freed or reallocated page, leading to a map-after-free vulnerability.

To address this, DMGUARD introduces *page tagging*, which associates a small, randomly generated tag with each page's allocation cycle. This mechanism uses two tags: a *page tag* (`PageTag`) stored in the page's metadata, and a *reference tag* (`RefTag`) embedded within any reference to the page, such as `struct page*` or a PFN. When a page is allocated, a new `PageTag` is generated, and any references created for this allocation are tagged with the same value in their `RefTag`. When the page is freed, the `PageTag` is invalidated. Before mapping a page, DMGUARD validates the reference by comparing its `RefTag` with the page's current `PageTag`. A match confirms that the reference is valid for the current allocation cycle. A mismatch indicates a stale reference, and DMGUARD prevents the operation, thus detecting a map-after-free vulnerability.

Crucially, the `RefTag` is carried along with both page descriptor pointer and Physical Frame Number (PFN). For instance, when a PFN is derived from a `struct page*` pointer, the `RefTag` from the pointer is also associated with the PFN. This ensures the temporal context is maintained regardless of the types of page reference. We detail the `RefTag` propagation in §5.

Detecting Map-After-Free. Figure 7 illustrates how page tagging prevents map-after-free. In a safe lifecycle (Figure 7a), a newly allocated page `P1` has the same `PageTag` and `RefTag` (*e.g.*, as `6`). The subsequent mapping is safe because the tags match. In contrast, if the page is freed (Figure 7b), its `PageTag` is updated to a new random value (*e.g.*, `2`). An attempt to map the page using the old reference `P1` fails because its `RefTag` (`6`) no longer matches the page's `PageTag` (`2`). Similarly, if the page is reallocated (Figure 7c), its `PageTag` is updated again (*e.g.*, `3`), while the old reference `P1` remains invalid.

The finite size of page tags means tag collisions allow a small possibility of missing detection of map-after-free, when a new `PageTag` randomly matches a stale `RefTag`. DMGUARD uses an 8-bit tag with one value reserved for a default tag (`0`) for pages that are not managed by an allocator (*e.g.*, MMIO regions). This provides a 254/255 (99.6%) detection probability for map-after-free vulnerabilities across different page allocations. To guarantee detection of immediate map-after-free, where a page is mapped right after being freed, DMGUARD ensures the new `PageTag` always differs from the previous one. This provides a 100% detection rate for the case in Figure 7b.

4.3 Putting It All Together: Lockless Integration

Figure 8 shows the five core operations of DMGUARD, which encode state transition tracking described in §4.2. For instance, `dmcAlloc` allocates a page (`L#32`) that has its `PageTag` originally initialized to `6`. It then invokes `TagUpdate` to generate a new random tag `2` different from `6` (`L#4`), and updates the page's `PageTag` with this new value (`L#7`). Thus, the page moves from *Freed-Unmapped* states to *Allocated-Unmapped*. `dmcAlloc` also sets the `RefTag` of the refernece (*i.e.*, `p`) to the same value (`L#35`).

`dmcMap` is responsible for the state transition from

```
struct page *
TagUpdate(struct page *p) {
  u8 newtag =
    RandomExclude(PageTag(p));
  u8 reftag = RefTag(p);
  // Update PageTag
  u8 pagetag = atomic_xchg(
    &PageTag(p), newtag);
  // Ensure tag match
  if (pagetag != reftag)
    BUG("Double Free");
  return newtag;
}

// Wrapper for Map()
void dmcMap(struct page* p) {
  MapCountInc(p);
  full_memory_barrier();
  // Page tag check
  if (PageTag(p)!=RefTag(p))
    BUG("Map-After-Free");
  Map(p);
  // require(!(freed(p) &&
  //         mapped(p)));
}
```

```
// Wrapper for Alloc()
struct page *dmcAlloc() {
  struct page *p = Alloc();
  u8 tag = TagUpdate(p);
  // Update RefTag
  SetRefTag(p, tag);
  return p;
}

// Wrapper for Free()
void dmcFree(struct page* p) {
  TagUpdate(p);
  full_memory_barrier();
  // Map count check
  if (MapCount(p) != 0)
    BUG("Free-Before-Unmap");

  Free(p);
  // require(!(freed(p) &&
  //         mapped(p)));
}

// Wrapper for Unmap()
void dmcUnmap(struct page *p){
  Unmap(p);
  MapCountDec(p);
  if (MapCount(p) < 0)
    BUG("Mapcount underflow");
}
```

Figure 8: Simplified code snippet of lockfree implementation of DMGUARD. Among three metadata, `MapCount` and `PageTag` are stored in memory allocated for each physical page (i.e., `PageMD`). `RefTag` is attached to a page reference (*e.g.*, `struct page*` or a PFN).

Allocated-Unmapped to *Allocated-Mapped* by incrementing the `MapCount` (`L#17`). In addition, it also checks if a map-after-free occurs, as described in §4.2.2—if the page has not been freed, `RefTag` and `PageTag` should be the same, but they are different if the page is freed (`L#20`). Similarly, `dmcUnmap` decrements the `MapCount`, and when it reaches to zero, the state of the page moves back to *Allocated-Unmapped* (`L#55`).

Lastly, `cmdFree` moves the page to *Freed-Unmapped* by invalidating `PageTag` (`L#41`), and check if a free-before-unmap occurs according to §4.2.1 (`L#44`).

Safe Concurrent Operations. As mentioned, the page management resides in a performance-critical path, and DMGUARD is designed to avoid using heavy synchronization primitives, such as lock. However, it is non-trivial to implement a lockless algorithm in the Linux kernel as it supports various architectures (*e.g.*, x86_64 [40], ARM [20], and RISC-V [12]) that have different memory models [11, 41, 83]. For instance, if a memory barrier at `L#18` does not exist, it is possible that in architectures with weak memory models (*e.g.*, ARM or RISC-V), updating a `MapCount` at `L#17` is conducted *after* checking a `PageTag` and a `RefTag` at `L#20` due to out-of-order execution [41]. Then it allows an execution order of `L#20` → `L#41` → `L#44` → `L#17` when `dmcMap` and `dmcFree` run concurrently, missing a map-after-free.

The correctness of DMGUARD relies on properly updating page metadata under concurrent execution. To this end, DMGUARD adopts two strategies. First, all metadata accesses use atomic instructions to prevent tearing (*e.g.*, avoiding non-atomic read-add-write instructions). Second, for instructions with dependencies, DMGUARD uses memory barriers to prevent undesired instruction reordering.

For example, in `dmcMap`, the memory barrier at `L#18` ensures that the map count increment at `L#17` occurs before the tag validation at `L#20`, so that any concurrent `dmcFree` will observe the updated map count. Similarly, in `dmcFree`, the tag invalidation at `L#41` happens before the map count check at `L#44`, guaranteeing that concurrent mappings will see the new tag and fail validation. The `atomic_xchg` operation in `TagUpdate` ensures that tag modifications are atomic across all CPU architectures.

We validate the correctness of DMGUARD against the Linux Kernel Memory Model (LKMM) [83] using litmus tests under concurrent execution scenarios. The LKMM provides a precise specification of memory ordering semantics across all architectures supported by the Linux kernel, allowing us to confirm that our lockfree design prevents race conditions between concurrent `Map` and `Free` operations. Specifically, we confirmed that the two invariants expressed as `require()`s in `dmcFree` and `dmcMap` are always satisfied when `BUG()` is not executed, ensuring that no dangling mappings can be created through race conditions. The litmus tests exhaustively explore possible race scenarios, including concurrent `Map`-`Free`, `Alloc`-`Map`, and `Free`-`Unmap` operations, confirming that the memory barriers and atomic operations are correctly placed to maintain correctness across different architectures.

Preventing Page Double-Free. Besides vulnerabilites directly related to a mapping, page double-free might occur when a page is freed twice without being allocated in between. Although a page double-free does not directly lead to a dangling mapping, it may corrupt the page allocator's internal state, leading to undefined behavior and potentially causing severe security problems.

DMGUARD prevents page double-free as well as free-before-unmap and map-after-free discussed in §4.2, by checking the `PageTag` against the `RefTag` when updating the page tag in `TagUpdate` (`L#9`). If the `PageTag` does not match the `RefTag`, it indicates that the page has already been freed, and thus double free is detected. This detection is atomic and concurrency-safe by using atomic operations to update and load `PageTag` from memory (`L#7`).

Compatibility with Other Page Operations. DMGUARD is compatible with various types of page operations, such as mapping a huge page [25] or page migrations. For huge pages, DMGUARD ensures that the `MapCount` of each underlying 4KB page is updated consistently. When a huge page is split into 4KB pages, the `MapCount` is correctly distributed to each page, maintaining accurate tracking. During page migration, DMGUARD ensures that the metadata is correctly handled. The `MapCount` and tags of the old page are checked before it is freed, and the new page receives its own lifecycle tracking, preventing any dangling references to the old page. Every page operation that changes the mapping of a page can be expressed as a combination of basic operations (*e.g.*, `Alloc`,

Table 1: Comparison of DMGUARD and `CONFG_PAGE_TABLE_CHECK` (`ptcheck`).

Vuln.	Page Context	Mapping Context	`ptcheck`	DMGUARD
Free-before-unmap (§4.2.1)	CPU	CPU	✓	✓
	CPU	Device	×*	✓
	Device	CPU	×*	✓
	Device	Device	×*	✓
Map-after-free (§4.2.2)	Any	Any	×†	✓
Race condition (§4.3)	Any	Any	×‡	✓

*`ptcheck` manages mapping status on the CPU side and requires additional device-side mapping status tracking to detect such cases.
†`ptcheck` does not track allocation status and fundamentally cannot detect mappings created to freed pages.
‡`ptcheck` is not concurrency-safe and allows TOCTOU-style vulnerabilities under race conditions.

`Map`, `Unmap`, and `Free`) and correctly updating the status of each affected page ensures the safe lifecycle of the page. In our evaluation, we show that DMGUARD successfully passes the Android Vendor Test Suite kernel tests (§6.4), which cover a wide range of page operations, demonstrating its robustness.

Caveats. DMGUARD relies on a set of well-defined kernel APIs in monitoring page operations. If a kernel driver manipulates a page table directly without using standard interfaces, DMGUARD is limited in tracking such changes and thus cannot provide protection. However, we believe such direct manipulation is rare and highly discouraged in modern kernel development. We detail GPU driver APIs in §5.

Comparison with CONFIG_PAGE_TABLE_CHECK. Table 1 summarizes the detection coverage comparison between DMGUARD and `CONFIG_PAGE_TABLE_CHECK` (*i.e.*, `ptcheck`). Correct virtual-to-physical address translation requires preventing invalid page states, where a page is *mapped* while *freed*. To this end, DMGUARD tracks both a page's allocation status and mapping status coherently across all processing units (*e.g.*, CPU and GPU). In contrast, `ptcheck` partially addresses physical-page UAF by tracking only CPU-side mapping status. As a result, for free-before-unmap vulnerabilities, DMGUARD detects device-level and cross-device scenarios (§4.2.1), whereas `ptcheck` cannot. For map-after-free vulnerabilities, DMGUARD detects them by tracking page allocation status (§4.2.2), while `ptcheck` cannot determine whether a physical page is freed when creating a mapping because it does not track page allocation status. Furthermore, `ptcheck` is not concurrency-safe, as it lacks proper synchronization to ensure correct metadata updates during concurrent page operations. This limitation allows TOCTOU-style vulnerabilities, where concurrent `Map` and `Free` operations create dangling mappings if `Free` intervenes between checking a page's allocation status (*i.e.*, L#20 in Figure 8) and creating a mapping (*i.e.*, L#22). In contrast, DMGUARD employs a lockless design with proper synchronization mechanisms to prevent such race conditions. We empirically verified this limitation of `ptcheck` with a synthetic vulnerability in our evaluation (§6.2).

```
1  // Map Count (Per-page metadata)
2  #define MapCount(p) PageMD[p].mapcount
3
4  // Page Tags
5  // - Page Tag (Per-page metadata)
6  #define PageTag(p) PageMD[p].pagetag
7  // - Reference Tag (Per-pointer metadata)
8  #define TOP_BYTE_MASK 0xFF00000000000000ULL
9  #define RefTag(p) ((unsigned long long) p >> 56) & 0xFF
10 #define SetRefTag(p, tag) \
11     (struct page *) ((unsigned long long) p & ~TOP_BYTE_MASK | \
12     (tag << 56))
```

(a) Pseudo code of APIs to manage metadata in DMGUARD.

```
1  /* mm/page_table_check.c */
2  struct page_table_check {
3      atomic_t anon_map_count;
4      atomic_t file_map_count;
5  +   atomic_t dev_map_count;
6  +   atomic_t page_tag;
7  };
```

(b) Modifications on `struct page_table_check` to support per-page metadata in DMGUARD.

Figure 9: DMGUARD implementation.

5 Implementation

In this section, we describe the implementation of DMGUARD. We implemented DMGUARD on Linux kernel v6.6.0 and Android kernels (Linux v6.1.99 and v4.9.270), with support for Mali GPU drivers and Qualcomm Adreno GPU drivers.

The core implementation DMGUARD, including page metadata to track page state machine and detect dangling mappings, consists of approximately 400 lines of code. We additionally implemented new kernel macros for tagged PFN without affecting existing kernel page management, which adds approximately 560 lines of code. To support DMGUARD in GPU drivers, minimal modifications were made, approximately 90 lines of code for Mali and 80 lines of code for Adreno. This suggests that integrating DMGUARD into other device drivers would also require only modest modifications.

Per-Page Metadata. DMGUARD leverages per-page metadata, which we call `PageMD`, to track page map count and page tag (Figure 9b). `PageMD` is allocated for each physical page in the system when the memory is initialized (*e.g.*, during boot time). We implemented `PageMD` by extending `struct page_table_check`, which is a per-page metadata structure used in `PAGE_TABLE_CHECK`. `MapCount` is the sum of `anon_map_count`, `file_map_count`, and `dev_map_count`, each of which contains the numbers of CPU anonymous mappings, CPU file mappings, and device mappings (both GPU and IOMMU), respectively. We added a field `dev_map_count` to

track mappings in both GPU and IOMMU page tables. To support page tagging, we created a field `page_tag` to store the 8-bit allocation tag (`PageTag`). This modification increases the size of `PageMD` from 8 bytes to 16 bytes, which is negligible in memory footprint as shown in §6.3.

In-Pointer Metadata. A reference tag (`RefTag`) is an 8-bit metadata to track a page's allocation status (§4.2.2), and is stored in the top byte of the page descriptor pointer (`struct page*`). On the ARM64 architecture, we implemented `RefTag` using the Top Byte Ignore (TBI) feature [49], which instructs the processor to ignore the top byte during address translation, enabling its use for metadata without impacting memory access. A similar feature on modern x86-64 processors, Intel Linear Address Masking (LAM) [52] that also ignores upper bits during address translation, makes our design portable.

DMGUARD randomly assigns `RefTag` from the `0x00` to `0xFE` on page allocation and deallocation. The value `0xFF` is reserved as the default tag for mappings without a corresponding physical pages, such as memory-mapped I/O. These mappings are exempt from tag checks because they lack a backing physical page and are therefore not subject to the page use-after-free vulnerabilities that DMGUARD targets.

Reference Tag Propagation. While `PageTag` is stored in `PageMD` for the page's lifetime, `RefTag` is set in the top byte of a pointer (*i.e.*, `struct page*`) and propagates with the pointer as the kernel runs. A challenge, however, arises when this pointer is converted into a physical frame number (PFN) and propagated to a page table entry (PTE). Tagging PFNs can be problematic because PFNs are mostly a generic integer type (i.e., `unsigned long`) and are frequently used in arithmetic operations, such as subtractions or comparisons, and the `RefTag` in the top byte of the PFN would corrupt the results and break the semantics of these operations.

For safe propagation of `RefTag`, we introduce a new wrapper type, `tagged_pfn_t`, which stores a 64-bit tagged PFN value. We modified several existing page and PFN-related macros in the kernel (*e.g.*, `page_to_pfn()`, `set_pte_at()`) to return `tagged_pfn_t` or receive `tagged_pfn_t` as an input, instead of the original integer PFN. Consequently, the compiler generates an error if arithmetic is performed directly on a `tagged_pfn_t`, preventing its accidental use in calculations. Any arithmetic operation that requires the raw PFN value must explicitly unwrap the tagged PFN, making tag stripping an explicit, auditable event.

To achieve this, the `page_to_pfn()` macro was changed to convert a `struct page*` to a tagged PFN. Similarly, `pfn_to_page()` is modified to propagate the `RefTag` from a tagged PFN to the `struct page*`. We also introduced helper functions, such as `tagged_pfn_add()`, to support arithmetics on wrapped PFNs. For PFNs that do not originate from a `struct page` and therefore lack an associated `RefTag`, a default tag value of `0x00` is assigned using `pfn_to_default_tagged_pfn_t()`. Note that the `RefTag` of `struct page*` stores the inverted value of the tag, such that a `struct page*` pointer with `0xFF` in its top byte is considered to carry a default tag. This design ensures seamless integration even if the page pointer does not carry a tag.

Compatibility with Other In-Pointer Metadata. DMGUARD is compatible with other in-pointer metadata-based mitigations in the Linux kernel, such as Pointer Authentication Code (PAC) [75] or Memory Tagging Extension (MTE) tag [19] in ARM64 Linux kernels. This is because DMGUARD tags page descriptor pointers, which reference page-related metadata. In contrast, prior defenses targeting virtual address-level memory corruption apply in-pointer metadata to pointers that reference actual data stored in memory, such as heap objects or stack variables.

Integration with GPU Drivers. GPU drivers manage their own page allocator and page table for GPU processes. We audited the kernel GPU drivers for Mali and Adreno GPUs, and identified functions responsible for page `Alloc`, `Map`, `Unmap`, and `Free` operations (see §A.1). We modified the GPU drivers to track the state of GPU pages and check if unsafe state transitions can lead to dangling mappings, as described in Figure 8.

6 Evaluation

This section evaluates DMGUARD in terms of its security, performance, and robustness. First, we describe our evaluation setup (§6.1). Next, we evaluate the security of DMGUARD on real-world and synthetic page use-after-free vulnerabilities (§6.2). Then, we demonstrate the performance and memory overhead of DMGUARD (§6.3). Finally, we evaluate the robustness of DMGUARD (§6.4).

6.1 Evaluation Setup

We evaluated DMGUARD in three environments: i) a virtualized environment running Linux kernel version 6.6.0 on QEMU [21] with AArch64 emulation and a Mali GPU driver with a simulated dummy GPU device (i.e., `MALI_NO_MALI` configuration); and two real-world environments, ii) a Pixel 8 running Android 15 (Linux kernel version 6.1.99), equipped with a 64-bit ARMv8.5 CPU and a Mali GPU; and iii) a Pixel 3 XL device running Android 12 (Linux kernel version 4.9.270), equipped with a 64-bit ARMv8.2 CPU and a Qualcomm Adreno GPU. We performed performance evaluations on the Pixel 8 device, while security evaluations were conducted on all three environments.

In both security (§6.2) and performance (§6.3) evaluations, we compared DMGUARD with the baseline system (i.e., `Baseline`) with a default kernel configuration, which does not have any dangling mapping mitigation. We also compared DMGUARD with the Linux kernel's `CONFIG_PAGE_TABLE_CHECK` (i.e., `ptcheck`) [51]. From a defense perspective, `ptcheck` is incomplete in two aspects. First, it does not provide comprehensive protection across diverse

page management components. In particular, it is unaware of pages managed by GPU allocators, and does not consider mappings created in GPU or IOMMU page tables. Consequently, `ptcheck` cannot detect page use-after-free attacks that occur in these contexts. Second, `ptcheck` does not detect map-after-free vulnerabilities. If a mapping occurs after the corresponding page has been freed, `ptcheck` fails to detect the resulting dangling mapping, leaving the system vulnerable to page use-after-free attacks. `ptcheck` is also vulnerable to race conditions, because its check during page free is non-atomic: the mapping count is verified before the actual page deallocation, creating a race window in which a concurrent mapping operation can introduce a dangling mapping.

Known Vulnerability Collection. In security evaluation (§6.2), we collected 19 real-world page use-after-free vulnerabilities in the Linux kernel from syzbot [30], and vulnerabilities in mobile GPU drivers (*i.e.*, Mali, Adreno, and PowerVR) from GitHub security lab [29], and Google Project Zero Issue Tracker [95]. Our manual inspection categorized every collected page use-after-free vulnerability as free-before-unmap vulnerability.

Most vulnerabilities had publicly available PoCs, which we used directly in our empirical security evaluation. The only exception was CVE-2023-33107, for which no PoC was available. To evaluate this case, we slightly modified the Adreno GPU driver so that the vulnerability could be triggered more easily while preserving its semantics.

Synthetic Vulnerabilities. To compare the security guarantees of DMGUARD and `ptcheck` under diverse conditions, we created five synthetic vulnerabilities that cover both map-after-free and concurrency-based scenarios. Four vulnerabilities were introduced as map-after-free cases by modifying a custom kernel module and the Mali GPU driver, each involving different combinations of page and mapping types. In addition, we created one race condition-based free-before-unmap vulnerability to assess whether DMGUARD and `ptcheck` are safe under concurrent execution. Specifically, we modified the CPU page free path to insert an artificial delay before the actual page reclamation. This race window allows a concurrent `Map` operation to occur in parallel, potentially bypassing the intended safety check and resulting in a dangling mapping.

6.2 Security Evaluation

We evaluated the security of DMGUARD by analyzing its effectiveness in detecting page use-after-free vulnerabilities. In the following, we first evaluate the effectiveness of DMGUARD in detecting known vulnerabilities, comparing it with the effectiveness of `ptcheck`. Next, we analyze the accuracy of DMGUARD in terms of false positives and false negatives. Finally, we evaluate DMGUARD's effectiveness in finding unknown page use-after-free vulnerabilities by running Syzkaller [31] on a Pixel 8 device with DMGUARD enabled.

Table 2: Empirical evaluation of DMGUARD and `ptcheck` on page use-after-free vulnerabilities.

Subsystem	CVE	Page Context	Mapping Context	`ptcheck`	DMGUARD
io_uring	CVE-2024-0582 [1]	CPU	CPU	✓	✓
usb	CVE-2024-47674 [7]	CPU	CPU	✓	✓
Linux mm	CVE-2024-53096 [8]	CPU	CPU	✓	✓
Mali	CVE-2024-1065 [2]	CPU	CPU	✓	✓
Mali	CVE-2022-36449 [92]	CPU	CPU	✓	✓
Mali	CVE-2025-0072 [65]	GPU	CPU	×	✓
Mali	CVE-2022-33917 [91]	GPU	CPU	×	✓
Mali	CVE-2024-0671 [94]	GPU	GPU	×	✓
Mali	CVE-2023-6241 [64]	GPU	GPU	×	✓
Adreno	CVE-2023-33107 [80]	GPU	IOMMU	×	✓
Synthetic	Map-after-free	CPU	GPU	×	✓
Synthetic	Map-after-free	CPU	CPU	×	✓
Synthetic	Map-after-free	GPU	GPU	×	✓
Synthetic	Map-after-free	GPU	CPU	×	✓
Synthetic	Free-before-unmap Race	CPU	CPU	×	✓

Table 3: Theoretical analysis of DMGUARD and `ptcheck` on page use-after-free vulnerabilities.

Subsystem	CVE	Page Context	Mapping Context	`ptcheck`	DMGUARD
Mali	GHSL-2023-005 [63]	GPU	GPU	×	✓
Mali	CVE-2022-38181 [62]	GPU	GPU	×	✓
Mali	CVE-2022-20186 [61]	GPU	GPU	×	✓
Adreno	CVE-2023-21666 [93]	GPU	CPU	×	✓
Adreno	CVE-2024-23384 [3]	GPU	IOMMU	×	✓
Adreno	CVE-2024-23380 [87]	GPU	IOMMU	×	✓
PowerVR	CVE-2024-31335 [4]	GPU	GPU	×	✓
PowerVR	CVE-2024-34729 [5]	GPU	GPU	×	✓
PowerVR	CVE-2024-34747 [6]	GPU	GPU	×	✓

Known Vulnerability Detection. Table 2 and Table 3 summarize the detection results of DMGUARD and `ptcheck` on 24 page use-after-free vulnerabilities. We categorized all vulnerabilities using two criteria. *Page Context* denotes the allocator that produced the page: CPU if allocated by the kernel buddy allocator, and GPU if allocated by a GPU page allocator. *Mapping Context* denotes the page table where the mapping resides: CPU if in the CPU page table, GPU if in a GPU page table, and IOMMU if in an IOMMU page table.

We empirically tested 15 vulnerabilities (10 real-world and 5 synthetic) across our three environments: CVE-2025-0072 and CVE-2023-6241 on the Pixel 8, CVE-2023-33107 on the Pixel 3 XL, and the remaining 12 vulnerabilities in the virtualized environment. We also theoretically analyzed 9 additional real-world vulnerabilities that could not be reproduced in any of our environments. The main reasons for non-reproducibility were i) lack of the required hardware, ii) missing virtual environment support for the target GPU driver, and iii) incompatibility between the versions of the vulnerable driver and the kernel setup. We identified the Page Context and Mapping Context of these vulnerabilities through manual analysis on the bug reports and code, and determined whether DMGUARD or `ptcheck` could detect them.

In the empirical evaluation (Table 2), DMGUARD successfully detected all 15 tested vulnerabilities, whereas `ptcheck` detected only 5. This gap reflects the limited detection ca-

pability of ptcheck. By contrast, DMGUARD comprehensively tracks unsafe page state transitions across allocation and mapping contexts, enabling it to capture all cases. In the theoretical evaluation (Table 3), our analysis confirmed that DMGUARD can detect all 9 vulnerabilities by design, while ptcheck cannot, as they all involve GPU pages with mappings in GPU or IOMMU contexts.

False Positives and Negatives. The accuracy of DMGUARD depends on correctly maintaining page state across Alloc, Free, Map, and Unmap. False positives and false negatives may arise if these operations are not correctly tracked. Our evaluation indicates that DMGUARD did not suffer from false positives or false negatives in practice. During performance evaluation (§6.3) and robustness testing (§6.4), DMGUARD raised no false alarms, demonstrating the absence of false positives. During empirical testing of known vulnerabilities (§6.2), DMGUARD successfully detected all cases, suggesting the absence of false negatives. While DMGUARD did not encounter inaccuracies in our evaluation, we further analyzed the potential sources of false positives and false negatives in the following.

In map counting, false positives can occur if a page is unmapped but the MapCount is not decremented. When the page is later freed, DMGUARD observes a positive MapCount and incorrectly reports a dangling mapping, even though the mapping has already been removed. Conversely, false negatives can occur if a page is mapped but the MapCount is not incremented, causing DMGUARD to treat the page as unmapped and miss a free-before-unmap vulnerability when the page is freed.

In page tagging, false positives are avoided because DMGUARD always synchronizes PageTag and RefTag when a page is allocated. However, false negatives can still occur if a mapping is created but the tag check is not performed, or if a page is freed and reallocated but its tag is not updated, failing to detect a map-after-free vulnerability. Another possible case is a tag collision; since DMGUARD uses 8-bit tags, with only 255 distinct values available (excluding the default tag), a later allocation may by chance reuse the same tag as a previous allocation (§4.2.2). With the collision probability 1/255, DMGUARD might miss a map-after-free vulnerability. However, this is difficult to exploit in practice, as the probability of a collision is low and the tag value is randomly chosen for each allocation and free.

Moreover, as DMGUARD relies on standard kernel APIs and specific GPU driver functions to track page status, if kernel developers bypass these interfaces and use ad-hoc mechanisms, DMGUARD may miss page state updates and incur false positives or false negatives. However, such practices are uncommon in kernel development, and our inspection of Linux kernel and GPU drivers did not reveal such cases.

Finding Unknown Vulnerabilities. We evaluated DMGUARD's ability to find unknown page use-after-free vulnerabilities by running Syzkaller [31] on a Pixel 8 device with DMGUARD enabled. This experiment uncovered a dangling mapping issue in the upstream GPU driver. Further inspection showed that when a process terminated without explicitly calling munmap() on GPU-allocated regions, GPU page table entries remained after the physical pages had been freed. We disclosed this finding to ARM and provided a detailed analysis and proof-of-concept code. ARM acknowledged the existence of dangling mappings but clarified that they are not exploitable, as they are created after all GPU tasks using the page table have terminated. Nonetheless, this finding shows that DMGUARD can discover in-the-wild page use-after-free vulnerabilities in modern operating systems. DMGUARD was able to detect this subtle bug despite the limited coverage of Syzkaller on the Mali GPU driver (about 15% due to implicit syscall dependencies), demonstrating its practicality as a detection tool in complex page management.

Table 4: Performance overhead on LMBench (average of 11 runs).

Test	ptcheck	DMGUARD
Simple syscall	-0.23%	+0.38%
Simple read	+0.17%	-0.92%
Simple write	+0.06%	-0.37%
Simple fstat	-0.55%	+5.66%
Simple open/close	-0.19%	+1.76%
Select on 100 fd's	-0.39%	-0.48%
Signal handler installation	+0.60%	-0.11%
Signal handler overhead	-0.73%	+0.61%
Protection fault	+3.81%	+2.55%
Process fork+exit	+6.89%	+10.05%
Process fork+execve	+9.59%	+12.58%
Process fork+/bin/sh -c	+5.46%	+4.81%
Geomean	+1.99%	+2.96%

6.3 Performance Evaluation

We evaluated the runtime and memory overhead of DMGUARD to assess its practicality in production systems. We conducted four types of measurements on a Pixel 8 device: (i) system call microbenchmarks using LMBench, (ii) application benchmarks using Geekbench 6 for CPU and GPU workloads, (iii) kernel boot time and application cold-startup latency, and (iv) memory overhead introduced by per-page metadata. All results are compared against the Baseline and ptcheck configurations.

Microbenchmark. We evaluated system call level performance using LMBench [57], reporting the average of 11 runs for each configuration. Table 4 shows the performance overhead of ptcheck and DMGUARD compared to the Baseline. On average, DMGUARD incurs minimal overhead (under 3%). DMGUARD exhibits slightly higher overhead compared to ptcheck, due to its additional page tagging mechanism. The fork system call shows relatively high overhead for both ptcheck and DMGUARD, as it sets a large number of page table entries when creating a new process.

Real-World Workload Benchmarks. We evaluated

Table 5: Geekbench results (average of 11 runs). The higher the better.

	Baseline	ptcheck	DMGUARD
CPU Single	1629.45	1622.82 (-0.41%)	1622.27 (-0.44%)
CPU Multi	4605.09	4593.73 (-0.25%)	4545.73 (-1.29%)
GPU (OpenCL)	7691.36	7683.64 (-0.10%)	7674.18 (-0.22%)

Table 6: Performance overhead on Phoronix Test Suite (average of 11 runs).

Test	ptcheck	DMGUARD
OpenSSL (sign)	+0.46%	+0.56%
OpenSSL (verify)	+0.51%	+0.61%
PyBench	+0.01%	+0.02%
Apache	+1.42%	+2.82%
Nginx	+0.75%	+1.16%
Redis	+0.31%	+1.91%
Unpack-Linux	+0.71%	+1.46%
Build-Linux-Kernel	+1.11%	+1.59%
Geomean	+0.66%	+1.26%

DMGUARD using the Phoronix Test Suite [58] to measure its impact on widely used real-world workloads, reporting the average of 11 runs for each configuration. Table 6 shows the results for seven representative benchmarks. The benchmarks cover diverse workload types including cryptographic operations (OpenSSL), interpreted language execution (PyBench), web servers (Apache and Nginx), database operations (Redis), and system operations (Unpack-Linux and Build-Linux-Kernel). DMGUARD incurs an average overhead of 1.26% across these workloads, demonstrating that it maintains low performance impact on real-world applications.

Application Benchmark. We also evaluated application-level performance using Geekbench 6 [48], reporting the average of 11 runs for each configuration. Table 5 shows the CPU single-core, CPU multi-core, and GPU (OpenCL) scores for the `Baseline`, `ptcheck`, and DMGUARD. Higher scores indicate better performance. DMGUARD incurred less than 0.5% overhead on CPU single-core test, and less than 0.3% overhead on GPU (OpenCL) test. Notably, DMGUARD incurs negligible overhead on GPU workloads, despite its comprehensive detection support for both CPU and GPU dangling mappings. The overhead was relatively higher on the CPU multi-core benchmark, which is likely due to the full memory barriers inserted to ensure safe concurrent execution between `Map` and `Free` operations. Overall, these results indicate that DMGUARD's impact on typical application workloads is minimal.

Kernel Boot and Application Cold Startup Time. We measured the impact of DMGUARD on kernel boot time and application cold-startup time, reporting the average of 11 independent runs. According to the LMBench results in Table 4, DMGUARD's overhead primarily stems from page table initialization, resulting in increased boot and cold-startup times.

Table 7: Boot time of Android kernel (average of 11 runs).

	Baseline (s)	ptcheck (s)	DMGUARD (s)
Kernel boot	0.510	0.516 (+1.18%)	0.528 (+3.53%)

Table 8: Cold-startup latency of AOSP system apps (average of 11 runs).

App	Baseline (ms)	ptcheck (ms)	DMGUARD (ms)
Calendar	142.82	147.36 (+3.18%)	148.91 (+4.26%)
Camera	206.18	207.91 (+0.84%)	208.45 (+1.10%)
Clock	190.18	191.73 (+0.82%)	195.55 (+2.82%)
Files	181.73	185.27 (+1.95%)	188.91 (+3.95%)
Gallery	187.36	198.27 (+5.82%)	199.91 (+6.70%)
Messaging	152.55	154.27 (+1.13%)	156.27 (+2.44%)
Phone	158.18	160.91 (+1.73%)	164.18 (+3.79%)
Search	116.27	118.09 (+1.57%)	119.36 (+2.66%)
Settings	209.64	214.55 (+2.34%)	212.73 (+1.47%)
WebView Shell	346.18	347.36 (+0.34%)	352.91 (+1.94%)
Geomean	—	+1.96%	+3.10%

Table 7 shows the boot time measurements and Table 8 shows the cold-startup times for ten Android Open Source Project (AOSP) system apps [15]. Note that these overheads are one-time costs and do not affect the ongoing system performance. For cold-startup time, the overhead is at most 13 ms, which does not significantly harm the user experience.

Memory Overhead. We measured the additional memory consumption introduced by DMGUARD on a Pixel 8 device. Since the memory overhead of DMGUARD and `ptcheck` comes from boot-time memory allocations and does not change dynamically during runtime, we measured the idle-time physical memory usage. Table 9 summarizes these measurements. As DMGUARD manages additional per-page metadata (including `page_tag` and `mapcount`), it introduces an extra 8 bytes per page compared to a kernel with only `ptcheck` enabled. On the Pixel 8, the total physical memory (7,739,468 KB) corresponds to 1,934,867 pages. At 8 bytes of overhead per page, DMGUARD's additional memory usage is roughly 15,478 KB. This matches the observed difference in free memory between `ptcheck` and DMGUARD. Compared to the `Baseline`, DMGUARD incurs a 1% memory overhead (50MB), which is negligible in modern systems.

6.4 Robustness Evaluation

As DMGUARD modifies the Linux kernel's page management, it is essential to ensure that the implementation does not compromise system stability. To this end, we evaluated the robustness of DMGUARD on a Pixel 8 device using the Android Vendor Test Suite (VTS) [32] kernel test plan (`vts-kernel`). It combines Kselftest [43] and the Linux Test Project (LTP) [72] to test a wide range of kernel functionalities, including memory management, under diverse and stress-testing conditions. We executed `vts-kernel` 10 consecutive times, during which

Table 9: Idle-time physical memory status and overhead.

	Baseline (KB)	ptcheck (KB)	DMGUARD (KB)
MemTotal	7,739,468	7,739,468	7,739,468
MemFree	4,841,540	4,801,772	4,791,660
Overhead	—	+0.84%	+1.05%

no crashes or hangs were observed, and all tests passed successfully. In addition, throughout the performance experiments reported in §6.3, DMGUARD did not induce any unexpected errors. These results demonstrate that DMGUARD preserves kernel correctness in practice and remains stable even under stress conditions.

7 Discussion

Kernel Memory Safety and Remaining Attack Surfaces. Comprehensive kernel security requires addressing two complementary aspects of memory safety. First, correct memory access within the virtual address space, which includes preventing traditional heap use-after-free via stale object pointers, buffer overflows, and other memory corruption issues. Second, correct virtual-to-physical address translation, which ensures that page table mappings remain valid throughout a physical page's lifecycle. DMGUARD addresses the second aspect by preventing physical-page use-after-free attacks. Prior works have largely focused on the first aspect, implicitly assuming correct address translation. Recent memory safety mechanisms [10, 23, 44, 56, 66, 67, 85, 90] enhance protection against traditional use-after-free and memory corruption vulnerabilities in the virtual address space, complementing DMGUARD's focus on the address translation layer. While DMGUARD focuses on ensuring correct page table integrity, attackers may still exploit vulnerabilities in other kernel subsystems, such as DMA race conditions [42] and side-channels [55, 76].

Page Table Corruption. DMGUARD focuses on preventing dangling mappings by enforcing correct page lifecycle management. However, it does not address memory corruption attacks that directly overwrite page mappings without using legitimate kernel APIs [34, 54, 86, 88]. These exploits may break the page state machine of DMGUARD by directly manipulating PTEs with a memory corruption vulnerability. Mitigating such threats requires complementary memory safety defenses, including MTE-based KASAN [27] or DFI [22], and DMGUARD is compatible with them.

Page Descriptor Pointer Corruption. Due to kernel defenses [9, 19, 75] against memory corruption attacks, attackers are incentivized to directly corrupt page descriptor pointers to gain arbitrary physical memory access. `PageJack` [74] leverages a heap memory corruption vulnerability to partially overwrite a page descriptor pointer, redirecting it to a freed page. Then, the freed page is accessed from user space using user-kernel page copy functions (e.g., `copy_page_to_iter` and `copy_page_from_iter`). Combined with page spraying, this allows the attacker to achieve page use-after-free attacks.

While DMGUARD does not currently address `PageJack`-like attacks that leverage page copy functions, DMGUARD can be extended to mitigate them. Similar to `Map`, DMGUARD can validate the `PageTag` and `RefTag` of the page descriptor pointer during page copy operations, as they also provide user access to physical page.

Page Metadata Protection. As DMGUARD rely on per-page metadata to track page status, it is vulnerable to memory corruption attacks that manipulate page metadata, similar to other defenses that rely on additional metadata [47, 71]. This risk can be mitigated by adopting orthogonal defenses against memory corruption attacks. Hardware memory tagging mechanisms [16, 19, 39] can be leveraged to further protect page metadata against side-channel leakage (e.g., Spectre [45]), because memory tags are expected to be inaccessible through normal memory access instructions as well as during CPU speculation.

Dangling TLB Entries. Similar to page tables, Translation Lookaside Buffer (TLB) entries can remain mapped to freed physical pages, enabling page use-after-free attacks [36, 37]. This situation arises when a PTE is removed and its backing page is freed, but its TLB entry is not properly flushed. Since TLBs reside in hardware MMUs, monitoring TLB status of a page is inherently difficult. We leave exploration of this direction for future work.

Discrete GPU Support. While DMGUARD currently focuses on integrated GPUs in mobile systems (e.g., ARM and Qualcomm GPUs), it can be extended to discrete GPUs (e.g., NVIDIA and AMD GPUs) with their own physical memory space. Discrete GPUs are likely to suffer from dangling mappings, because they often support a unified memory architecture with the host CPU memory [24], which involves complex page mapping relationships across CPUs and GPUs. DMGUARD can be extended to discrete GPUs to define a similar page state machine across their disjoint physical memory spaces to mitigate page UAF vulnerabilities.

8 Related work

We review existing attacks and defenses on page tables in the Linux kernel.

Attacks on Page Tables. KSMA [88] overwrites a PTE and sets its permission bits to create a user-space alias of a privileged kernel page, enabling direct user access to kernel memory. Page-Oriented Programming [34] similarly relies on arbitrary kernel memory read and write primitives to overwrite a targeted PTE, thereby hijacking the control flow of direct branches inside the kernel, which remain unprotected by existing control-flow integrity (CFI) mechanisms. Dirty Pagetable [86] and SLUBStick [54] begin from heap memory corruption vulnerabilities and subsequently corrupt PTEs,

granting the attacker arbitrary physical memory access.

Defenses on Page Tables. Prior research [73, 84] has proposed data flow integrity techniques to protect page tables from kernel memory corruption attacks. More recently, industry efforts have introduced hardware-based page table protection mechanisms (e.g., Intel HLAT [46], Apple SPTM [17]). However, none of these approaches can prevent dangling mappings created by legitimate yet buggy page management logic in the kernel.

The Linux kernel supports various debugging features for page management. `CONFIG_PAGE_TABLE_CHECK` [51] tracks only mapping status in the CPU context, and therefore falls short of providing comprehensive mitigation as explained in §4.3. `CONFIG_DEBUG_VM` enables various sanity checks for virtual memory management operations to catch bugs during development. `CONFIG_PAGE_POISONING` clears pages with poison patterns whenever pages are freed to prevent data leakage. While these features help detect some bugs during development, they do not provide comprehensive protection against physical-page use-after-free attacks.

9 Conclusion

Despite the deployment of advanced mitigations in production systems, physical-page use-after-free vulnerabilities continue to threaten the integrity of page tables–the foundation of nearly all defense mechanisms. In response, this paper introduces DMGUARD, a lightweight runtime mechanism that detects the occurrence of dangling mappings before they can be exploited. We implement DMGUARD in recent Linux kernels and on Android devices, demonstrating that it effectively detects dangling mappings with negligible performance overhead. This result shows that DMGUARD represents a practical step forward in strengthening kernel security by preventing physical-page use-after-free attacks.

Acknowledgment

This work was supported by Mobile eXperience(MX) Business, Samsung Electronics Co., Ltd. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

Ethical Considerations

We present a stakeholder-based ethics analysis following The Menlo Report principles for our page use-after-free detection research.

Stakeholders. Research team seeking career advancement and scientific contribution, Linux kernel developers gaining enhanced debugging tools, system administrators receiving improved defenses, end users obtaining security protection, hardware vendors benefiting from responsible disclosure to ARM for Mali GPU issue, malicious actors potentially misusing tools, and society experiencing overall cybersecurity improvement.

Ethical Principles. *Beneficence*: Research provides systematic protection against kernel vulnerabilities with minimal performance overhead. Open-source release maximizes benefits across Linux ecosystem. *Respect for Persons*: Lab-only evaluation respects autonomy without collecting personal data or requiring user consent. *Justice*: Open-source availability ensures equitable access regardless of resources. *Respect for Law and Public Interest*: Complies with laws, serves public interest, includes responsible disclosure of Mali GPU issue to ARM.

Harms and Mitigation. *Tangible*: System instability risks mitigated through extensive VTS testing and comprehensive documentation. *Intangible*: Potential misuse for vulnerability discovery, though defensive benefits outweigh offensive applications. *Unmitigated*: Risk of evasion technique development exists but doesn't justify withholding security improvements.

Decision. Both utilitarian and deontological analyses support publication. Open-source defensive research benefits society more than potential harms. No active threats require delayed publication.

Open Science

In accordance with USENIX Security's open science policy, we provide the following artifacts to support the reproducibility and evaluation of our research. We provide DMGUARD source code and kernel patches in https://doi.org/10.5281/zenodo.17970502, including build scripts and configuration files, along with setup documentation. All artifacts are available at submission time through the anonymized links provided above. Complete build and execution instructions are included in the respective README files. The source code can be compiled and deployed following the step-by-step instructions provided. After paper acceptance, non-anonymized permanent links will be provided for long-term availability and will be verified by the Artifact Evaluation Committee.

References

[1] Cve-2024-0582, 2024. https://project-zero.issues.chromium.org/issues/42451653.

[2] Cve-2024-1065, 2024. https://project-zero.issues.chromium.org/issues/42451668.

[3] Cve-2024-23384, 2024. https://project-zero.issues.chromium.org/issues/42451701.

[4] Cve-2024-31335, 2024. https://project-zero.issues.chromium.org/issues/42451686.

[5] Cve-2024-34729, 2024. https://project-zero.issues.chromium.org/issues/42451685.

[6] Cve-2024-34747, 2024. https://project-zero.issues.chromium.org/issues/42451698.

[7] Cve-2024-47674, 2024. https://project-zero.issues.chromium.org/issues/366053091.

[8] Cve-2024-53096, 2024. https://project-zero.issues.chromium.org/issues/374117290.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. In *Proceedings of the ACM Transactions on Information and System Security*, Nov. 2009.

[10] J. Ahn, J. Lee, K. Lee, W. Gwak, M. Hwang, and Y. Kwon. {BUDAlloc}: Defeating {Use-After-Free} bugs by decoupling virtual address management from kernel. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[11] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[12] S. Andrew Waterman, Krste Asanović. The RISC-V Instruction Set Manual, 2017. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[13] Android. Kernel control flow integrity, 2024. https://source.android.com/docs/security/test/kcfi.

[14] Android. Graphics, 2025. https://source.android.com/docs/core/graphics.

[15] Android. Android open source project, 2025. https://source.android.com/.

[16] Apple. Memory integrity enforcement: A complete vision for memory safety in apple devices, 2025. https://security.apple.com/blog/memory-integrity-enforcement/.

[17] Apple. Operating system integrity, 2025. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b.

[18] A. Arcangeli. Userfaultfd: User-space page fault handling, 2015. https://lwn.net/Articles/636226/.

[19] Arm. Memory tagging extension, 2019. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[20] ARM Holdings. Arm® Architecture Reference Manual for A-profile architecture, 2022. https://developer.arm.com/documentation/ddi0487/latest.

[21] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.

[22] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[23] K. Dinh Duy, K. Cho, T. Noh, and H. Lee. Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenghagen, Denmark, Nov. 2023.

[24] L. K. Documentation. Heterogeneous memory management, 2025. https://docs.kernel.org/mm/hmm.html.

[25] L. K. Documentation. HugeTLB pages, 2025. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html.

[26] L. K. Documentation. Memory protection keys, 2025. https://docs.kernel.org/core-api/protection-keys.html.

[27] L. K. Documentation. Memory tagging extension in AArch64 linux, 2025. https://docs.kernel.org/arch/arm64/memory-tagging-extension.html.

[28] J. Edge. Kernel address space layout randomization, 2013. https://lwn.net/Articles/569635/.

[29] GitHub. GitHub security lab, 2025. https://github.blog/category/security/.

[30] Google. syzbot: continuous kernel fuzzing, 2025. https://syzkaller.appspot.com/.

[31] Google. syzkaller: kernel fuzzer, 2025. https://github.com/google/syzkaller.

[32] Google. Vendor test suite (vts) and infrastructure, 2025. https://source.android.com/docs/core/tests/vts.

[33] Z. Guo, D. K. Le, Z. Lin, K. Zeng, R. Wang, T. Bao, Y. Shoshitaishvili, A. Doupé, and X. Xing. Take a step further: understanding page spray in linux kernel exploitation. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[34] S. Han, S.-J. Kim, W. Shin, B. J. Kim, and J.-C. Ryou. {Page-Oriented} programming: Subverting {Control-Flow} integrity of commodity operating system kernels with {Non-Writable} code pages. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[35] B. Hawkes. Attacking the qualcomm adreno gpu, 2020. https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html.

[36] J. Horn. Taking a page from the kernel's book: A tlb issue in mremap(), 2019. https://googleprojectzero.blogspot.com/2019/01/taking-page-from-kernels-book-tlb-issue.html.

[37] J. Horn. Android: Spf in aosp 5.10/5.15 kernels can create dangling tlb entries by misdirecting tlb flushes on race with mremap() [and other miscellaneous issues in spf], 2024. https://project-zero.issues.chromium.org/issues/377569381.

[38] Intel. Iommu and dma remapping, 2023. https://www.intel.com/content/www/us/en/docs/vtd-spec/intel-virtualization-technology-directed-io-architecture-specification/4-0/.

[39] Intel. Chktag: x86 memory safety, 2025. https://community.intel.com/t5/Blogs/Tech-Innovation/open-intel/ChkTag-x86-Memory-Safety/post/1721490.

[40] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals, 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[41] D. R. Jeong, Y. Choi, B. Lee, I. Shin, and Y. Kwon. Ozz: Identifying kernel out-of-order concurrency bugs with in-vivo memory access reordering. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, Austin, TX, Nov. 2024.

[42] B. Johannesmeyer, R. Isemann, C. Giuffrida, and H. Bos. Dynamic detection of vulnerable dma race conditions. In *Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS)*, Taipei, Taiwan, Oct. 2025.

[43] T. kernel development community. Linux kernel selftests, 2025. https://docs.kernel.org/dev-tools/kselftest.html.

[44] J. Kim, J. Park, Y. Lee, C. Song, T. Kim, and B. Lee. Petal: Ensuring access control integrity against data-only attacks on linux. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, Oct. 2024.

[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2019.

[46] KunalMehta. Intel® virtualization technology - redirect protection (intel® vt-rp), 2025. https://community.intel.com/t5/Blogs/Tech-Innovation/Client/Intel-Virtualization-Technology-Redirect-Protection-Intel-VT-rp/post/1672593.

[47] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, Oct. 2014.

[48] P. Labs. Geekbench 6, 2024. https://www.geekbench.com/.

[49] Linaro. Top byte ignore for fun and memory savings, 2019. https://www.linaro.org/blog/top-byte-ignore-for-fun-and-memory-savings/.

[50] Linux. Kcfi support, 2022. https://lwn.net/Articles/907639/.

[51] Linux. Page table check, 2025. https://docs.kernel.org/mm/page_table_check.html.

[52] LWN. Support for intel's linear address masking, 2022. https://lwn.net/Articles/902094/.

[53] L. Maar, F. Draschbacher, L. Lamster, and S. Mangard. {Defects-in-Depth}: Analyzing the integration of effective defenses against {One-Day} exploits in android kernels. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[54] L. Maar, S. Gast, M. Unterguggenberger, M. Oberhuber, and S. Mangard. {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, Aug. 2024.

[55] L. Maar, J. Juffinger, T. Steinbauer, D. Gruss, and S. Mangard. Kernelsnitch: Side-channel attacks on kernel data structures. In *Proceedings of the 2025 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Apr. 2025.

[56] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow. Preventing kernel hacks with hakc. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Apr. 2022.

[57] L. W. McVoy, C. Staelin, et al. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, San Francisco, CA, Jan. 1996.

[58] P. Media. Phoronix Test Suite, 2024. https://www.phoronix-test-suite.com/.

[59] Microsoft. Data execution prevention, 2004. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[60] Microsoft. Control flow guard for platform security, 2024. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[61] M. Y. Mo. Corrupting memory without memory corruption, 2022. https://github.blog/security/vulnerability-research/corrupting-memory-without-memory-corruption/.

[62] M. Y. Mo. Pwning the all google phone with a non-google bug, 2022. https://github.blog/security/vulnerability-research/pwning-the-all-google-phone-with-a-non-google-bug/.

[63] M. Y. Mo. Pwning pixel 6 with a leftover patch, 2023. https://github.blog/security/vulnerability-research/pwning-pixel-6-with-a-leftover-patch/.

[64] M. Y. Mo. Gaining kernel code execution on an mte-enabled pixel 8, 2024. https://github.blog/2024-03-18-gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/.

[65] M. Y. Mo. Bypassing mte with cve-2025-0072, 2025. https://github.blog/security/vulnerability-research/bypassing-mte-with-cve-2025-0072/.

[66] M. Momeu, S. Schnückel, K. Angnis, M. Polychronakis, and V. P. Kemerlis. Safeslab: Mitigating use-after-free vulnerabilities via memory protection keys. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, Oct. 2024.

[67] M. Momeu, A. J. Gaidis, J. vd Heidt, and V. P. Kemerlis. Iubik: Isolating user bytes in commodity operating system kernels via memory tagging extensions. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (Oakland)*, San Fransisco, CA, May 2025.

[68] NVIDIA. Zero-copy gpu memory, 2022. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#zero-copy-memory.

[69] NVIDIA. NVIDIA BlueField-3 DPU Data Processing Unit, 2023. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[70] NVIDIA. Gpu computing, 2024. https://developer.nvidia.com/gpu-computing.

[71] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[72] L. T. Project. Linux test project, 2024. https://linux-test-project.github.io/.

[73] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xmp: selective memory protection for kernel and user space. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, USA, May 2020.

[74] Z. Qian, J. Hu, J. Zhou, Q. Tang, and W. Shen. Pagejack: A powerful exploit technique with page-level uaf, 2024. https://i.blackhat.com/BH-US-24/Presentations/US24-Qian-PageJack-A-Powerful-Exploit-Technique-With-Page-Level-UAF-Thursday.pdf.

[75] Qualcomm. Pointer authentication on armv8.3, 2017. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[76] H. Ragab, A. Mambretti, A. Kurmus, and C. Giuffrida. {GhostRace}: Exploiting and mitigating speculative race conditions. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[77] A. Ryabinin. Kasan: kernel address sanitizer, 2015. https://lwn.net/Articles/618180/.

[78] Samsung. Samsung knox security platform, 2024. https://www.samsung.com/us/business/solutions/services/mobility-services/security/.

[79] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-sanitizer: a fast address sanity checker, 2012. https://research.google.com/pubs/archive/37752.pdf.

[80] B. Sevens and J. Horn. Cve-2023-33107: Qualcomm adreno gpu kgsl_ioctl_gpuobj_import integer overflow, 2023. https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2023/CVE-2023-33107.html.

[81] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[82] E. Stepanov and K. Serebryany. Hardware-assisted addresssanitizer, 2018. https://clang.llvm.org/docs/HardwareAssistedAddr

essSanitizerDesign.html.

[83] A. Stern. Explanation of the Linux-Kernel Memory Consistency Model, 2017. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/memory-model/Documentation/explanation.txt.

[84] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, USA, May 2020.

[85] Z. Wang, Y. Guang, Y. Chen, Z. Lin, M. Le, D. K. Le, D. Williams, X. Xing, Z. Gu, and H. Jamjoom. {SeaK}: Rethinking the design of a secure allocator for {OS} kernel. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[86] N. Wu. Dirty pagetable: A novel exploitation technique to rule linux kernel, 2023. https://web.archive.org/web/20250304082609/https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html#425-catch-a-page-table-again.

[87] E. R. Xiling Gong, Xuan Xing. The way to android root: Exploiting your gpu on smartphone, Aug. 2024.

[88] W. Yong. Ksma: Breaking android kernel isolation and rooting with arm mmu features, 2018. https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf.

[89] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim. {In-Kernel}{Control-Flow} integrity on commodity {OSes} using {ARM} pointer authentication. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[90] J. You, J. Seo, K. Lee, Y. Cho, and Y. Paek. bastag: Byte-level access control on shared memory using arm memory tagging extension. In *Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS)*, Taipei, Taiwan, Oct. 2025.

[91] G. P. Zero. Arm mali csf: page freed while still mapped into host userspace due to vma split mishandling, 2022. https://bugs.chromium.org/p/project-zero/issues/detail?id=2325.

[92] G. P. Zero. Arm mali non-csf: Imported_user_buf is released without flushing host-side vmas, leading to page uaf, 2022. https://bugs.chromium.org/p/project-zero/issues/detail?id=2327.

[93] G. P. Zero. Qualcomm adreno/kgsl: pages can be freed to page pool while having gpu references [on !config_qcom_kgsl_use_shmem], 2023. https://project-zero.issues.chromium.org/issues/42451566.

[94] G. P. Zero. Cve-2024-0671, 2024. https://bugs.chromium.org/p/project-zero/issues/detail?id=2522.

[95] G. P. Zero. Project zero issue tracker, 2025. https://project-zero.issues.chromium.org/.

A Appendix

A.1 Page Management Interfaces

This appendix enumerates the key functions that DMGUARD instruments for page management operations. We detail the interfaces for page allocation, deallocation (free), mapping, and unmapping across three components: the Linux kernel (§A.1.1), the Arm Mali GPU driver (§A.1.2), and the Qualcomm Adreno GPU driver (§A.1.3). The following lists provide a comprehensive reference for these instrumentation points.

A.1.1 Linux Kernel

Map `set_pte_at`
Unmap `ptep_get_and_clear`
`ptep_get_and_clear_full`
Alloc `post_alloc_hook`
Free `free_pages_prepare`

A.1.2 Mali GPU Driver

Map `entry_set_ate`
`entry_set_pte`
Unmap `entries_invalidate`
Alloc `kbase_mem_pool_alloc`
`kbase_mem_pool_alloc_locked`
`kbase_mem_pool_alloc_pages`
`kbase_mem_pool_alloc_pages_locked`
Free `kbase_mem_pool_free`
`kbase_mem_pool_free_locked`
`kbase_mem_pool_free_pages`
`kbase_mem_pool_free_pages_locked`
`free_partial_locked`

A.1.3 Adreno GPU Driver

Map `__arm_lpae_set_pte`
`arm_lpae_init_pte`
Unmap `__arm_lpae_set_pte`
`arm_lpae_init_pte`
`__arm_lpae_free_pgtable`
`__arm_lpae_unmap`
Alloc `kgsl_pool_alloc_page`
Free `kgsl_pool_free_page`