PhantomMap: GPU-Assisted Kernel Exploitation

Jiayi Hu*, Qi Tang‡, Xingkai Wang*, Jinmeng Zhou*, Rui Chang* and Wenbo Shen*†
*Zhejiang University
‡Jilin University
{lotuhu,bittervan,11921110,crix1021,shenwenbo}@zju.edu.cn, tangqi5522@mails.jlu.edu.cn

***Abstract*—Graphics Processing Units (GPUs) have become essential components in modern computing, driving high performance rendering and parallel processing. Among them, Arm's Mali GPU is the most widely deployed in mobile devices. In contrast to the mature and robust defenses on the CPU side, the GPU remains poorly protected. Consequently, GPUs have become a preferred target for attackers seeking to bypass CPU defenses. Notable incidents, such as Operation Triangulation, have demonstrated how GPU-side vulnerabilities can be exploited to compromise system security. Despite the rising threat, the comprehensive and in-depth security analysis of the Mali GPU is still missing.**

To address this gap, we conduct the first in-depth security analysis of Mali GPU's memory mapping mechanism and uncover two new security weaknesses: allocation–mapping decoupling and missing physical address validation. Exploiting these weaknesses, we introduce PhantomMap, a novel GPU-assisted exploitation technique that transforms limited heap vulnerabilities into powerful physical memory read/write primitives—bypassing mainstream kernel defenses without requiring privileged capabilities or information leaks. To assess its security impact, we develop a static analyzer that systematically identifies all vulnerable mapping paths, uncovering 15 exploit chains across two Mali driver architectures. We further demonstrate PhantomMap 's practicality by developing 15 end-to-end exploits based on real-world CVEs, including the first public exploit for CVE-2025-21836. Finally, we design and implement a lightweight in-driver mitigation that eliminates the root cause with minimal performance overhead on Pixel 6 and Pixel 7 devices.

I. Introduction

Graphics Processing Units (GPUs) have become critical to modern computing, driving high-performance rendering and massively parallel workloads. Among them, Arm's Mali GPU has emerged as the most widely deployed GPU in mobile devices, consistently dominating the global smartphone market with an average 46% market share over the past three years[1]. Beyond smartphones, Mali GPUs are also extensively integrated into tablets, embedded platforms, and other end-point devices—making them a pervasive and high-impact target in today's computing landscape.

†Wenbo Shen is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240201
www.ndss-symposium.org

As GPUs become ubiquitous in modern computing systems, they are increasingly targeted by attackers. While CPU-side kernel mitigations—such as W⊕X, KASLR, and CFI—have significantly raised the bar for traditional exploitation, they also force adversaries to pivot toward under-protected components. *The GPU, with its distinct architecture and weaker security design, offers a compelling path to circumvent CPU-side defenses.* A notable case is Operation Triangulation, which exploits GPU vulnerabilities to bypass CPU defenses and execute arbitrary code on iPhone[2]. The risk is even more severe on Mali-based devices, where the GPU remains accessible from highly restricted contexts, making it a viable and attractive target for privilege escalation.

Targeting the Mali GPU, recent efforts—including those by Google Project Zero—have uncovered several exploitable page-level use-after-free (UAF) vulnerabilities in the GPU driver[3], [4], [5], [6]. These flaws allow adversaries to tamper with GPU page tables and escalate privileges from unprivileged contexts. However, these prior works focus almost entirely on manual bug discovery, lacking systematic and automatic analysis. To our best knowledge, there is no comprehensive and in-depth analysis of the Mali GPU's memory management. As a result, the security of the Mali GPU remains largely unexplored, and its broader impact on system security is still unknown.

To fill this gap, this paper presents the first systematic analysis of the Mali GPU's memory mapping mechanism. Our study reveals two previously overlooked but critical security weaknesses in the GPU memory mapping workflow. First, the decoupling of physical memory allocation from page table updates in the Mali GPU driver introduces a time-of-check to time-of-use (TOCTOU) window, which can be exploited by attackers. Second, the Mali GPU driver doesn't validate the physical addresses before mapping, allowing arbitrary kernel memory to be remapped into user space without any security checks.

Leveraging these two security weaknesses, we propose a novel exploitation technique named *PhantomMap*, a GPU-assisted exploitation technique that abuses Mali's physical memory remapping to achieve kernel code injection and execution. Instead of tampering with page tables directly, PhantomMap hijacks the driver's legitimate mapping workflow to create malicious mappings to the user space. As a result, PhantomMap converts modest heap-corruption bugs into powerful arbitrary physical-memory read/write primitives and

bypasses mainstream kernel defenses through GPU-initiated writes.

PhantomMap offers three advantages over existing techniques: (i) it adapts to virtually any heap vulnerability—whether in kmalloc or vmalloc; (ii) it can bypass mainstream kernel defenses and provide a direct and powerful kernel-code-injection primitive; and (iii) it requires neither privileged capabilities nor information leaks, making the attack feasible even in the most restricted environments.

To assess the security impact of PhantomMap, we develop an LLVM-based static analyzer that systematically examines the Mali GPU driver to identify all instances of the decoupled allocation–mapping pattern. Our analyzer combines bottom-up control-flow analysis with targeted data-flow tracking to correlate memory allocation and mapping operations via their associated key structures. To improve precision, we introduce a memory-origin–based correlation analysis that leverages the driver's explicit distinction between physical memory sources, enabling accurate matching of allocation–mapping paths. Applying this analyzer to Mali GPU drivers across both Job Manager (JM) and Command Stream Frontend (CSF) driver architectures, we identify 15 distinct exploit chains involving 6 key structures.

To further validate the exploitability of these exploit chains and practicality PhantomMap, we conduct an extensive evaluation using real-world vulnerabilities. Specifically, we select 13 up-to-date and representative CVEs spanning all major types of kernel heap vulnerabilities. Based on these, we developed 15 end-to-end exploits that reliably achieve kernel code injection and privilege escalation, demonstrating significantly higher success rates and stability compared to existing techniques. Notably, we successfully exploited CVE-2025-21836, a vulnerability with no previously known working exploit, thereby showcasing PhantomMap's capability to unlock unexploitable bugs.

To fully mitigate PhantomMap, we propose a lightweight, software-based defense implemented directly within the Mali GPU driver. Our mitigation prevents the driver from mapping any non-GPU physical pages into user space, thereby eliminating the core attack surface. We integrates this mitigation into the Mali GPU drivers and upstream Linux kernel, and deployed it on real Android devices, including the Pixel 6 and Pixel 7. The experimental evaluation across multiple devices and driver architectures shows that our mitigation incurs negligible performance overhead—averaging 0.56% on the Pixel 6 and 0.34% on the Pixel 7.

We have responsibly disclosed the PhantomMap attack surface to Arm's Product Security Incident Response Team, including full technical details, working exploits, a reproducible environment, and proposed mitigation patches. Arm has acknowledged our disclosure and assigned a tracking identifier for their investigation process. All vulnerabilities leveraged in our study have been addressed and patched. Our primary goal is to strengthen the security of the Arm Mali GPU and to improve the overall security of GPU-enabled systems and devices.

In summary, this paper makes the following contributions:

- We conduct the first systematic analysis of the Mali GPU's memory mapping mechanism and discover two security weaknesses.
- We propose a novel GPU-assisted exploitation technique named PhantomMap.
- We develop a static analyzer to systematically examine the Mali GPU driver.
- We demonstrate PhantomMap's practicality using 13 CVEs.
- We design and implement a lightweight mitigation to defend against PhantomMap.

We make all artifacts publicly available at https://github.com/Lotuhu/PhantomMap, including our static analysis tool, our light-weight mitigation patch and the performance test setups and LTP test results.

II. Background

In this section, we give preliminary knowledge of architectural design and memory management mechanism of Mali GPU driver and introduce diverse categories kernel mitigations on the CPU-side.

A. Mali GPU Memory Management

The Arm Mali GPU driver architecture has evolved to handle user-space tasks through two primary command-processing models: the traditional Job Manager (JM)[7] and the more recent Command Stream Frontend (CSF)[8]. This subsection provides an overview of the key components relevant to our work, focusing on memory management and omitting details of other mechanisms like job scheduling or hardware abstraction layers.

To support these operations and ensure process-level isolation, the driver is built around several core abstractions. At the highest level, `struct kbase_device` serves as the global abstraction for an individual Mali GPU hardware unit, directly interfacing with the physical device via MMIO and orchestrating system-wide resources. For each user process, `struct kbase_context` encapsulates a dedicated GPU execution environment. It manages the process's virtual address space, job queues, and event notifications. The lifecycle of a `kbase_context` is strictly bound to its process's `/dev/maliX` file descriptor, ensuring synchronized resource cleanup upon process termination.

Notably, in most production devices, the system-level IOMMU (SMMU) is not enabled for the Mali GPU by default. Instead, Arm Mali GPUs implement their own built-in MMU to handle address translation internally. In this condition, when accessing the system memory, addresses issued directly by the Mali GPU are treated as physical addresses. This behavior has been leveraged by several publicly disclosed CVE exploits[4], [5], [6], confirming its prevalence in real-world deployments.

Furthermore, it implements a semi-custom memory management subsystem. While leveraging standard kernel APIs like `alloc_pages()` for page allocation, the driver uses its own two-tiered pooling mechanism, `struct kbase_mem_pool`, for efficient management. A

process first attempts to allocate memory from its own `kbase_context->mem_pools`. If empty, it requests from the global `kbase_device->mem_pools`. Only when both are exhausted is `alloc_pages()` called to refill the device's pools. When memory is freed, it is returned first to the context's pools, then to the device's pools if the context's pools are full, and finally back to the kernel's buddy system if both are full.

B. Existing Kernel Mitigations

Modern kernels adopt a wide range of security mitigations to defend against exploitation. These mitigations raise the bar for attackers by restricting code execution, defeating code reuse, protecting critical data structures, and limiting the attack surface.

1) Code injection mitigations: Modern kernels enforce write-xor-execute policies to stop classic shellcode injection. On Arm architectures, this is enforced through the XN (eXecute-Never) and PXN (Privileged XN) page table attributes, which prevent code execution from user- or kernel-writable pages. These protections are typically implemented through kernel page table configurations.

To further protect the page tables themselves, additional defenses have been proposed, such as SecVisor, TZ-RKP, and SKEE[9], [10], [11]. These systems aim to enforce kernel code integrity and page table protection at the hypervisor or TrustZone level. Together, these mitigations make classical code injection attacks infeasible in modern kernel environments.

2) Code reuse mitigations: To combat code reuse attacks such as return-oriented programming (ROP), modern kernels deploy Address Space Layout Randomization (ASLR). In particular, Kernel Address Space Layout Randomization (KASLR) randomizes the base address of the kernel image at boot time, making it more difficult for attackers to locate usable code gadgets[12]. Finer-grained variants (FG-KASLR) shuffle individual function sections to remove large contiguous blocks of gadgets.

Complementing KASLR, LLVM's Kernel Control-Flow Integrity (KCFI) attaches a compile-time type identifier to every indirect call and checks it at run time, blocking forward-edge hijacks[13]. Furthermore, on Arm64 platforms, Pointer Authentication (PAC) has been introduced to sign and verify return addresses and function pointers, providing strong protection against backward-edge attacks such as return address overwrites[14], [15].

3) Data-only attack mitigations: As control-flow hijacking becomes harder, attackers increasingly shift toward data-only attacks that target critical kernel data structures. To defend against such threats, modern processors provide memory access control features—Arm PAN (Privileged Access Never) and x86 SMAP (Supervisor Mode Access Prevention)—which prevent the kernel from accessing user-space memory unless explicitly allowed, mitigating user-to-kernel data corruption vectors[16], [17]. Moreover, researchers proposed additional defenses to protect sensitive kernel data structures such as `modprobe_path` and page table from being corrupted[18], [19]. Additionally, Google's SLAB_VIRTUAL goes a step further, giving every cache its own virtual address range in vmalloc, blocking cross-cache attacks that underpin many heap exploits[20].

4) Attack surface reduction and isolation: Beyond code, control-flow, and data protections, modern kernels also reduce the attack surface exposed to unprivileged users. Mandatory Access Control (MAC) frameworks such as SELinux/SEAndroid—enabled and enforced by default on Android since 2015—limit what even compromised processes can access, based on strict policy definitions[21].

III. PhantomMap Exploitation Technique

While existing kernel mitigations significantly raise the bar for traditional exploitation techniques—blocking code injection, disrupting control-flow hijacking, and safeguarding critical data structures—they are all designed and enforced on the CPU side. None of them account for threats originating from heterogeneous computing components like GPUs.

In this section, we present the PhantomMap exploitation technique in detail. We first define the threat model, then analyze two critical security flaws in the Mali GPU driver: decoupled memory operations and missing physical address validation. Next, we show how these implementation flaws enable arbitrary physical memory remapping, demonstrate PhantomMap 's effectiveness, highlight its advantages over prior techniques, and explain how it bypasses modern CPU-side mitigations.

A. Threat Model

We assume that the attacker is an unprivileged user operating on the Android system. Specifically, the user has no extra linux kernel capabilities. The attacker's objective is to exploit kernel vulnerabilities to bypass existing kernel-level protections and achieve privilege escalation.

While we assume the presence of heap corruption vulnerabilities in the kernel, the attacker's capabilities are limited to conventional memory corruption primitives, such as use-after-free (UAF), out-of-bounds (OOB) access, and double-free (DF). We explicitly exclude more advanced capabilities, including information leakage vulnerabilities and side-channel attacks.

For deployed defenses, we assume that all major Android kernel mitigations commonly found on production devices are enabled. As discussed in §II-B, these defenses can be categorized as follows: *1) code injection mitigations*, such as XN, PXN, and page table protection schemes, including SecVisor, TZ-RKP, and SKEE; *2) code reuse mitigations*, including KASLR, forward-edge and backward-edge protection schemes; *3) data-only attack mitigations*, including PAN, sensitive data structure protection, and SLAB protection such as SLAB_VIRTUAL; *4) attack surface reduction and isolation*, including SELinux/SEAndroid; Finally, we assume that the hardware—including the CPU, memory, and GPU—is trusted and reliable. There are no vulnerabilities, malicious modifications, side-channels, or backdoors at the hardware or bus level.

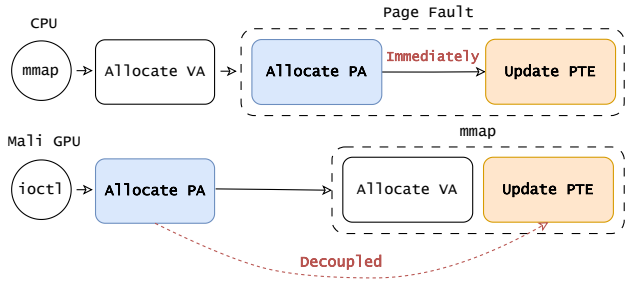

Fig. 1: Mapping patterns between CPU and Mali GPU.

B. Attack Insights

Building on the threat model described above, we conducted a thorough examination of the Mali GPU memory-management stack. To support this effort, we developed a dedicated static-analysis tool (detailed in §IV) for comprehensive code analysis. Our methodology began with contrasting user-space memory mapping patterns between CPU and GPU subsystems. Our analysis revealed two previously unknown security weaknesses in the Mali GPU driver—decoupled memory operations and unchecked GPU memory pages.

Decoupled memory operations. On CPUs, memory management typically follows a demand-paging model, where a user process first receives a virtual address through system calls such as `mmap()`. The actual allocation of physical memory and the corresponding page table updates are deferred until a page fault occurs. At that point, the kernel atomically allocates a physical page and maps it to the virtual address, which causes the page fault. Such immediate mapping operations leave no opportunity for attacker intervention.

In contrast, the Mali GPU driver adopts a fundamentally different memory allocation flow, as shown in Fig. 1. User processes must first explicitly allocate physical memory in advance, often through driver-specific interfaces. Only after the physical pages are allocated does the user invoke a separate system call, `mmap()`, which prompts the driver to update the GPU page tables using functions such as `kbase_mmu_insert_pages()`.

This decoupling between physical memory allocation and page table update on the Mali GPU introduces a large, attacker-controllable time window. During this time window, an attacker may exploit a kernel vulnerability to tamper with the allocated physical pages or their metadata. When the mapping operation is eventually triggered, the driver—unaware of any manipulation—blindly inserts the corrupted data into the page tables, leading to the remapping of arbitrary physical memory into user space.

Security Weakness 1: The Mali GPU driver decouples physical memory allocation from page table updates, creating a user-controllable time window that can be exploited to remap physical memory.

```
1  int insert_page(struct vm_area_struct *vma,...struct
   ↪  page *page,...) {
2      ...
3      retval = validate_page_before_insert(vma, page);
4      if (retval)
5          goto out;
6      ...
7      retval = insert_page_into_pte_locked(page, ...);
8      ...
9  }
10
11 int validate_page_before_insert(...,struct page*page){
12     struct folio *folio = page_folio(page);
13     if (!folio_ref_count(folio))
14         return -EINVAL;
15     ...
16     if (folio_test_anon(folio) ||
       ↪  folio_test_slab(folio) || page_has_type(page))
17         return -EINVAL;
18     ...
19 }
```

(a) Mapping check in CPU mapping interface

```
1  int kbase_mmu_insert_pages(struct tagged_addr *phys,
   ↪  ...) {
2      ...
3      err = mmu_insert_pages_no_flush(..., phys, ...);
4      ...
5  }
```

(b) Unchecked GPU mapping root interface

Fig. 2: Comparison of memory mapping check between CPU and Mali GPU.

Missing GPU physical address validation. In CPU memory management, the kernel performs a series of strict security checks before mapping a physical page into user space. As illustrated in Fig. 2a, the kernel first ensures that the target physical page holds a valid reference count, preventing freed pages from being remapped (Line 13). It then verifies that the page is neither an anonymous page nor a slab page and that it is not marked with any special flags (Lines 16). These safeguards are essential to prevent sensitive kernel pages—such as those containing critical data structures—from being exposed to user space.

In contrast, the Arm Mali GPU driver omits such checks entirely. It implements custom memory mapping and page fault handling interfaces for GPU-specific Virtual Memory Areas (VMAs), but these interfaces lack any form of validation for the physical pages being mapped. As shown in Fig. 2b, the driver directly inserts the physical page into the GPU page tables without verifying its legitimacy. This critical oversight allows arbitrary and potentially sensitive physical memory to be mapped into user space, representing a severe security vulnerability.

Security Weakness 2: The Mali GPU driver doesn't validate physical pages before mapping, allowing arbitrary or sensitive kernel memory to be remapped into user space without any security checks.

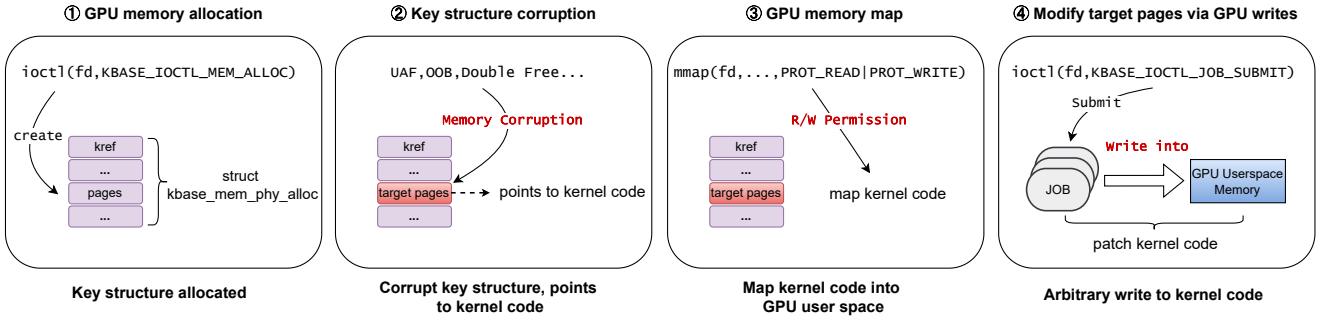

Fig. 3: PhantomMap exploit steps.

C. PhantomMap Attack

While the above two weaknesses together form a classical time-of-check to time-of-use (TOCTOU) vulnerability, they expose key data structures to attacker manipulation. Specifically, the decoupling between memory allocation and mapping introduces an exploitable time window during which critical structures can be tampered with. Simultaneously, the lack of validation in the mapping interfaces allows unverified physical pages to be directly inserted into the page tables.

Building on these newly identified flaws, we propose a novel attack technique called *PhantomMap*, which enables the remapping of arbitrary physical memory into user space. Notably, the Mali GPU driver does not use an external IOMMU, allowing direct access to the entire system's physical memory. This enables attackers to write to any physical address without triggering address translation errors. Unlike traditional GPU page table attacks, PhantomMap targets intermediate data structures involved in the Mali GPU memory mapping process. It exploits kernel vulnerabilities to replace GPU-allocated physical memory with attacker-targeted physical pages. These substituted pages are then legitimately mapped into user space via the driver's custom interfaces. The full PhantomMap exploitation consists of four steps, as illustrated in Fig. 3.

① **GPU memory allocation.** The attack begins with a standard GPU memory allocation request, during which the driver creates an internal structure to manage the allocated physical pages. This structure becomes the primary target of our exploitation technique. Specifically, the attacker first invokes the `ioctl(fd, ..., KBASE_IOCTL_MEM_ALLOC)` syscall, prompting the Mali GPU driver to allocate physical memory and construct a `kbase_mem_phys_alloc` key structure. This structure contains metadata for memory management, including a variable-length array named `pages` (as shown in Fig. 3 ①) that stores the physical addresses of the allocated memory. These addresses are intended for future use during the mapping phase.

② **Key structure corruption.** Leveraging *Security Weakness 1*, the attacker exploits the time window between memory allocation and mapping to corrupt the key structure created in Step ①. During this window, a kernel memory corruption vulnerability—such as a UAF, OOB write, or a Double Free combined with heap spraying—is used to tamper with the `kbase_mem_phys_alloc` structure, whose user-controllable size makes it an ideal target across various heap vulnerability capabilities. The attacker modifies the `pages` array to replace the original physical addresses with targeted physical addresses, such as read-only kernel code memory.

③ **GPU memory map.** Once the internal structure is corrupted, the attacker triggers a GPU memory mapping operation that maps the manipulated physical addresses into page tables without validation. This is done by invoking the `mmap()` syscall, using the handle returned by the earlier `ioctl()` call in Step ①. As described in *Security Weakness 2*, the Mali GPU driver does not verify the contents of the pages array before mapping, allowing the attacker-corrupted physical addresses (e.g., kernel code memory) to be mapped directly into GPU user space with read/write permissions.

④ **Modify target pages via GPU writes.** With the attacker-controlled physical memory now mapped into user space, arbitrary memory write operations can be carried out.

To achieve this, the attacker can define a GPU write job and submit it using the `ioctl(fd, ..., KBASE_IOCTL_JOB_SUBMIT)` syscall. When the GPU processes the job later, it executes the GPU write operation, writing to the GPU userspace address provided by the attacker. In this case, the attacker gains the ability to modify arbitrary physical memory, such as kernel code. By patching kernel code or critical variables (e.g., `selinux_state`, `selinux_enforcing`), the attacker can escalate privileges and disable key security mechanisms, such as SELinux.

PhantomMap attack leverages two subtle but critical design flaws in the Mali GPU driver—decoupled memory operations and the absence of physical page checks—to achieve reliable remapping of arbitrary physical memory into user space. By targeting intermediate data structures rather than directly manipulating GPU page tables, PhantomMap provides a stealthy and powerful exploitation path that bypasses traditional kernel protections. In the following section, we discuss the unique advantages of PhantomMap compared to existing GPU and kernel exploitation techniques.

```
1 static inline struct kbase_mem_phy_alloc
  ↪ *kbase_alloc_create(...
2 {
3     struct kbase_mem_phy_alloc *alloc;
4     size_t alloc_size = sizeof(*alloc) +
      ↪ sizeof(*alloc->pages) * nr_pages;
5     ...
6     if (alloc_size >
      ↪ KBASE_MEM_PHY_ALLOC_LARGE_THRESHOLD)
7         alloc = vmalloc(alloc_size);
8     else
9         alloc = kmalloc(alloc_size, GFP_KERNEL);
10    ...
11 }
```

Fig. 4: Allocation strategy of `kbase_mem_phy_alloc`.

D. PhantomMap Advantages

PhantomMap presents a general, practical, and powerful exploitation primitive that outperforms existing kernel exploitation techniques across several dimensions. Below, we summarize its three key advantages.

1) Applicable to almost all heap vulnerabilities: PhantomMap is broadly applicable to almost all heap-based kernel vulnerabilities, regardless of whether the vulnerable object resides in the kmalloc or vmalloc region. This makes it a highly versatile and general-purpose exploitation technique.

Although previous work[22] explored heap objects for kmalloc-based exploitation, their techniques remain almost entirely limited to the kmalloc heap. In contrast, vulnerabilities in vmalloc-allocated objects remain difficult to exploit due to the lack of flexible primitives. Recent work[23] demonstrates vmalloc exploitation using eBPF JIT spraying; this approach depends on privileged capabilities like `CAP_BPF`, which are typically unavailable to unprivileged users—especially on Android systems with strict SELinux enforcement.

PhantomMap introduces a unified and elastic exploitation primitive that works across both kmalloc and vmalloc heaps. This flexibility is achieved by exploiting the allocation behavior of a critical internal structure. As detailed in Fig. 4, the kernel dynamically allocates the `kbase_mem_phy_alloc` structure using either `kmalloc()` or `vmalloc()` depending on the requested size (Lines 6-9). Importantly, the size of this structure can be controlled by an unprivileged user via a standard `ioctl()` call. By carefully adjusting this size, an attacker can deterministically steer the structure into the *desired heap region* with the *desired object size*, aligning it with the location of a vulnerable object to facilitate reliable exploitation. This heap placement and object size controllability allows PhantomMap to adapt to almost any heap corruption vulnerability, making it a one-size-fits-most solution for modern kernel heap exploitation.

Under the constraint of limited vulnerability capabilities—such as when the exploit primitive is restricted to writing kernel heap pointers into freed objects—pivoting is an essential step. Most exploit techniques, such as DirtyCred[24], require this pivot, and PhantomMap is no exception. However, the adaptability of PhantomMap makes it more flexible in choosing among various pivoting methods.

2) Powerful code injection primitives: PhantomMap provides an arbitrary physical memory write primitive that can bypass all existing code injection mitigation and enable direct kernel code injection. Existing techniques require complex exploitation chains to gain arbitrary code execution. For example, DirtyCred and similar approaches rely on privileged file overwrites to abuse kernel modules, often requiring capability escalation or kernel module loading. These chains are tightly coupled and limited to specific environments.

In contrast, PhantomMap enables direct kernel code patching by remapping arbitrary physical memory into user space. This allows overwriting kernel code or critical variables (e.g., `selinux_enforcing`) without requiring chained primitives or capability escalation. The primitive is reusable across diverse environments and devices, significantly reducing the overhead of adapting exploits to new platforms.

3) Does not require capabilities or information leaks: PhantomMap operates entirely without privileged capabilities or information leaks, making it viable even in the most restricted environments. Existing techniques such as Interpreter-Flow Hijacking[25] and USMA[26] depend on capabilities like `CAP_BPF` or `CAP_NET_RAW`, which are typically unavailable to unprivileged users now. Additionally, four techniques: Interpreter-Flow Hijacking, USMA, RetSpill[27], and Page Spray[28] require leaking kernel text or heap addresses, which is an increasingly difficult and unstable step in modern systems with robust mitigations.

PhantomMap is fully capability- and infoleak-free. PhantomMap completes the exploitation process without requiring any capabilities. Moreover, precise heap spraying and size control allow PhantomMap to place vulnerable and target structures adjacently. This flexibility eliminates the need for any information leak during the exploitation.

E. Mitigations Bypass

PhantomMap can effectively bypass state-of-the-art mitigations, including code injection mitigations, code reuse mitigations, and data-only attack mitigations. Building on both the advantages discussed above and its ability to bypass these mitigations, we present a comparative analysis of popular exploit techniques in TABLE I, which compares PhantomMap's performance against popular exploit techniques.

1) Bypass code injection mitigations: The GPU-assisted physical memory write primitive provided by PhantomMap allows attackers to directly patch kernel code, thereby achieving code injection and execution. This method effectively bypasses traditional mitigations, such as XN and PXN.

Additionally, PhantomMap can bypass hypervisor-based protection such as Samsung RKP[19] and SecVisor[9], which typically leverage Stage-2 page tables to enforce read-only permissions on critical kernel regions (e.g., the `.text` section). However, these protections remain incomplete, as certain kernel read-only data structures are not fully safeguarded. PhantomMap specifically targets this gap: despite RKP's page table protections, PhantomMap manipulates intermediate-state data structures and leverages legitimate kernel workflows to

TABLE I: A comparative analysis of exploit techniques against the latest security mitigations. ✔means that the technique is capable of bypassing the corresponding mitigation. ✗means that the technique can not bypass the mitigation. ✔✗ means that the bypass is only achievable under certain constraints or assumptions.

		Code Reuse Mitigations		Data-only Attack Mitigations	
Technique	**Capability Required**	**No Need Infoleak**	**Bypass CFI**	**Bypass Data Protection**	**Bypass SLAB_VIRTUAL**
DirtyCred [24]	NULL	✔	✔	✗	✗
USMA [26]	CAP_NET_RAW	✗	✔	✔	✔
DirtyPageTable [29]	NULL	✔	✔	✗	✗
Retspill [27]	NULL	✗	✔✗	✔	✔
Interp-flow Hijacking [25]	CAP_BPF	✗	✔	✔	✔
SLUBStick [30]	NULL	✔	✔	✗	✗
Page Spray [28]	NULL	✗	✔	✔	✗
PhantomMap	NULL	✔	✔	✔	✔

modify the GPU page table. Then, by corrupting critical read-only data structures, such as `security_hook_heads`, using the GPU, PhantomMap can successfully bypass these protections and ultimately achieve kernel privilege escalation.

2) Bypass code reuse mitigations: Nowadays, popular exploitation techniques primarily target non-control data and can bypass CFI effectively, with RetSpill[27] as an exception, still relying on control-flow hijacking. RetSpill assumes attackers can hijack backward-edge branches to gain PC control, but converting a heap bug into a stack overwrite is complex and multi-staged. Thus, we argue that RetSpill can only bypass CFI protection under specific constraints. PhantomMap diverges from control-flow hijacking by exclusively targeting non-control data structures, thereby evading CFI/KCFI protections entirely.

3) Bypass data-only attack mitigations: (1) Critical data protection: Critical data protections monitor and restrict modifications to protect certain non-control data regions, thereby securing critical data such as `struct cred` and page tables. Consequently, they successfully block exploit techniques like DirtyCred, which achieve privilege escalation by swapping protected `struct cred`. Similarly, methodologies such as DirtyPageTable[29] and SLUBStick[30], which leverage object-level heap vulnerabilities to tamper with page tables, are also mitigated.

Rather than directly manipulating page tables, PhantomMap targets and corrupts unprotected mapping-related key structures. Then it leverages the kernel's standard page-table update process, causing the kernel to unknowingly use this tampered data to inject malicious physical addresses. As the entire update procedure adheres to a legitimate kernel workflow, PhantomMap elegantly bypasses critical data protections enforced by the Linux kernel.

(2) SLAB_VIRTUAL: Many popular exploit techniques frequently rely on either corrupting critical kernel data structures[24] (e.g., cred, file) within dedicated caches or pivoting object-level vulnerabilities to page-level capabilities (e.g., modifying page tables)[28], [30], [29]. Both strategies inherently require cross-cache attacks. However, SLAB_VIRTUAL effectively mitigates traditional cross-cache attacks. In this condition, PhantomMap operates by corrupting kernel objects residing in general caches, aligning with mainstream kernel heap vulnerabilities. This generality allows PhantomMap to achieve exploitation without relying on cross-cache techniques, thereby circumventing this latest mitigation entirely.

F. Generality Discussion

We further evaluated PhantomMap on Qualcomm's Adreno GPU. The memory management model of the Qualcomm GPU tightly couples physical page allocation with page table updates, which precludes any opportunity for an attacker to control and manipulate intermediate states prior to memory mapping. As a result, PhantomMap is ineffective against Qualcomm's GPU.

IV. Identification Of Exploit Chains And Targets

To fully understand the impact of PhantomMap on the Mali GPU driver and uncover all potential exploit chains, a systematic analysis is essential. The Mali GPU driver features an extensive and complex codebase, where different `ioctl` commands may trigger diverse types of memory allocations. These allocated regions can subsequently be mapped through multiple, often disjoint code paths—such as direct `mmap` calls or custom page fault handlers. This complexity renders manual auditing both infeasible and error-prone, increasing the risk of missing critical exploit paths.

To address this challenge, we developed a dedicated static analysis tool designed to systematically scan the Mali GPU driver's codebase and identify exploit chains in which the allocation of memory resources is decoupled from their eventual mapping. Through this process, we also identify key data structures that act as bridges between allocation and mapping operations—critical components that serve as prime targets for exploitation. Importantly, our analysis spans the distinct logic and codebases of both the JM and CSF architectures of

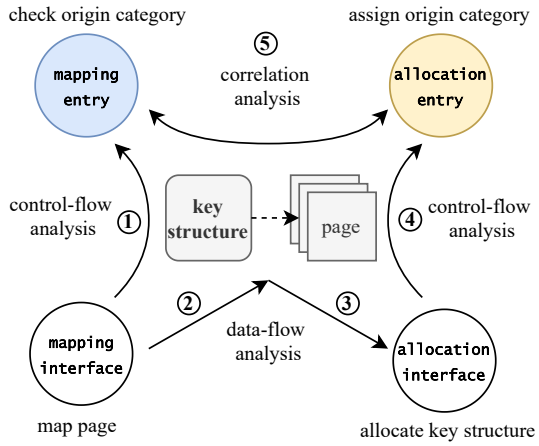

Fig. 5: PhantomMap analyzer workflow.

the Mali GPU. In the following, we present the design and workflow of the PhantomMap static analyzer and demonstrate how it enables the analysis and discovery of potential exploit chains of PhantomMap.

A. Identification Methodology

To systematically discover potential exploit chains, our tool implements a multi-phase static analyzer. As illustrated in Fig. 5, the workflow of PhantomMap analyzer combines bottom-up control-flow analysis with targeted data-flow analysis. Additionally, to enhance analysis accuracy in complex cases, we introduce a dedicated correlation analysis phase. In the following, we will describe the design and workflow of PhantomMap analyzer in detail.

Backward analysis of mapping operations. We begin with the identification of mapping interfaces, which are the final processing points responsible for handling memory mapping requests within the Mali GPU driver. Crucially, we focus exclusively on kernel-to-user mapping operations, excluding kernel-to-kernel mappings(such as `vmap()`), as the former presents a significantly more direct and potent attack surface. Mali GPU driver does not reuse the Linux kernel's standard mapping interface like `remap_pfn_range()`. Instead, it implements its own custom mapping interfaces for mapping GPU memory to both CPU and GPU user space.

For GPU-side mapping, we identify functions responsible for updating the GPU's page tables, which are represented by the `struct kbase_mmu_table`. We collect all functions that either take this structure as a parameter or internally reference it to perform page table modifications. For CPU-side mapping, Mali GPU utilizes a unified and driver-specific interface `mgm_vmf_insert_pfn_prot`, provided via its device operations, to handle all CPU mapping operations. Finally, we define these functions as the *mapping interface* of PhantomMap static analyzer.

TABLE II: Key structures identified by the PhantomMap analyzer. ‡ indicates that the structure has a flexible allocation size.

Structure	Property	Related Field
kbase_mem_phy_alloc‡	k/vmalloc	pages
kbase_alloc_import_user_buf‡	k/vmalloc	pages
kbase_queue	kmalloc	phys
kbase_device	vmalloc	reg_start
kbase_context	kmalloc	aliasing_sink_page
kbase_csf_device	vmalloc	dummy_db_page

Based on the set of mapping interfaces, we perform a bottom-up control-flow analysis to construct a comprehensive call graph of mapping operations (Step ① of Fig. 5). By traversing this call graph, we identify *mapping entry* as high-level functions that serve as mandatory convergence points in the call graph, through which all paths reaching towards the mapping interfaces must pass. These functions typically correspond to user-accessible entry points, such as handlers for `mmap` or specific `ioctl` syscalls. These map entries can directly serve as the controllable triggers for an attacker to execute the mapping phase of PhantomMap.

During the bottom-up traversal of the call graph constructed in Step ①, we perform targeted data-flow analysis in Step ② to trace the origin of the physical memory involved in mapping operations. Specifically, we track how physical memory is propagated to the mapping interface, aiming to identify its source. During this analysis, if we observe that the physical memory originates from a particular kernel data structure, we designate that structure as a *key structure*, a specific data structure responsible for carrying physical memory information throughout the mapping process. These key structures also serve as exploitation targets for PhantomMap.

Analysis of allocation operations. With the mapping paths and their associated key structures established, we proceed to identify the corresponding allocation call sites for these key structures. As part of Step ③, we conduct a data-flow analysis to locate these allocation sites. We refer to the functions responsible for allocating the key structures as *allocation interface*. Starting from each identified allocation interface, we perform a bottom-up control-flow analysis in Step ④ to trace back to the user-invokable functions (e.g., syscall handlers) capable of triggering the allocation interface. We then define these interfaces as *allocation entry*, analogous to the mapping entry identified earlier.

Finally, our goal is to match the allocation entry and mapping entry identified in the preceding steps to form end-to-end exploit chains. Based on the steps described, we have now identified allocation entries (and their corresponding key structures) and mapping entries. Establishing a direct correspondence between an allocation entry and a mapping entry that both operate on the same key structure would define a potential exploit chain.

However, this methodology encounters an additional challenge in practice. We observed that certain key structures are

TABLE III: Exploit chains discovered by the PhantomMap static analysis tool. ● indicates that the exploit chain can map arbitrary physical memory to user space with read/write permissions. ◐ indicates that the exploit chain can map arbitrary physical memory to user space with read-only permissions. † means that the exploit chain exists exclusively on CSF architecture.

Allocation Entry	Syscall	Mapping Entry	Syscall	Related Key Structure	Read/Write
kbase_mem_alloc	ioctl	kbase_gpu_mmap	mmap	kbase_mem_phy_alloc	●
kbase_mem_alloc	ioctl	kbase_mem_commit	ioctl	kbase_mem_phy_alloc	●
kbase_mem_alloc	ioctl	kbase_cpu_mmap	ioctl	kbase_mem_phy_alloc	●
kbase_mem_import	ioctl	kbase_gpu_mmap	mmap	kbase_mem_phy_alloc	●
kbase_mem_import	ioctl	kbase_map_external_resource	ioctl	kbase_alloc_import_user_buf	●
kbase_mem_import	ioctl	kbase_mem_flags_change	ioctl	kbase_mem_phy_alloc	●
kbase_mem_import	ioctl	kbase_map_external_resource	ioctl	kbase_mem_phy_alloc	●
kbase_mem_alias	ioctl	kbase_gpu_mmap	mmap	kbase_mem_phy_alloc	●
kbase_mem_alias	ioctl	kbase_cpu_mmap	mmap	kbase_mem_phy_alloc	◐
kbase_create_context	open	kbase_gpu_mmap	mmap	kbase_context	◐
kbase_create_context	ioctl	kbase_map_external_resource	mmap	kbase_mem_phy_alloc	●
kbase_platform_device_probe†	—	kbase_csf_cpu_mmap_user_reg_page	mmap	kbase_device	◐
kbase_platform_device_probe†	—	kbase_csf_cpu_mmap_user_io_pages	mmap	kbase_device	●
kbase_csf_doorbell_mapping_init†	—	kbase_csf_cpu_mmap_user_io_pages	mmap	kbase_csf_device	●
kbase_csf_alloc_command_stream_user_pages†	ioctl	kbase_csf_cpu_mmap_user_io_pages	mmap	kbase_queue	●

heavily reused across the driver, serving as a shared structure for numerous memory operations. As a result, a single key structure may be associated with multiple allocation entries as well as multiple mapping entries, creating a many-to-many ambiguity. This ambiguity makes it difficult to identify a clear and valid allocation–mapping chain.

Memory-origin–based correlation analysis. To resolve this challenge, we introduce a more granular analysis, leveraging the driver's explicit distinction between physical memory sources. Mali GPU driver categorizes managed memory regions using the `kbase_memory_type` enumeration. We call this property the *origin category*. The origin category allows the driver to distinguish between memory with different origins and properties (such as memory allocated natively by the driver vs. memory imported from userspace) and to apply specific handling logic and mapping rules accordingly. Crucially, this origin category is stored as a field within the same key structure that holds the physical page addresses.

In Step ⑤, we leverage this origin category for precise correlation analysis. For each identified allocation path targeting a key structure, our analyzer performs forward dataflow analysis to determine the specific origin category value assigned to the structure's related field during the allocation routine. Conversely, for each identified mapping path operating on a key structure, our analyzer scans for conditional logic (e.g., if statements, switch cases) that checks the type field within the mapping routine and records which origin category values are permitted to proceed down that specific mapping path.

Finally, we establish our correlation rule that an allocation entry is paired with a mapping entry to form a correct and viable exploit chain only when two conditions are met: (1) they operate on the shared key structure. (2) the origin category assigned by the allocation routine is permitted by the origin category checked by the mapping routine.

B. Identification Results

We applied the PhantomMap static analyzer to the Mali GPU driver across the two distinct kernel branches corresponding to our target architectures: the *gs-android-gs-raviole-mainline* branch (JM architecture) and the *android-gs-pantah-5.10-android14-qpr3* branch (CSF architecture). As shown in TABLE III, our analysis identified 15 distinct end-to-end exploit chains where memory allocation and mapping operations are dangerously decoupled. Moreover, we confirm that 12 of these 15 paths provide the capability to map arbitrary physical memory into a user-space process with full read and write permissions, representing a systemic and high-impact security threat.

These exploit chains are all mediated by a set of 6 unique key structures that serve as the bridge between the allocation and mapping phases. As summarized in TABLE II, these structures encompass diverse allocation types, including both `kmalloc` and `vmalloc`. Notably, some of these key structures behave as elastic objects, whose allocation size can be controlled by attackers. This property allows them to be adapted to the specific capabilities of various kernel heap vulnerabilities, demonstrating the versatility of the PhantomMap.

Furthermore, these chains exploit various time windows that exist between the initial allocation of physical memory and the final update of the user-space page tables. The most common pattern involves separate user-accessible syscalls for memory

allocation and mapping, allowing attackers to exploit the time window between the two operations. Additionally, in certain exploit chains, resources are pre-allocated during device initialization (e.g., in `kbase_platform_device_probe`) and persist throughout the device's entire operational lifetime, creating a permanent attack window. Our analysis reveals that these exploit chains are distributed across both the JM and CSF architectures, with most chains existing in both. This demonstrates that this attack surface is not an isolated legacy issue but a fundamental design flaw that persists in the driver's evolution.

C. Exploit Chain Validation And Analysis

Regarding false positives, PhantomMap analyzer initially identified 16 potential exploit chains. We conducted a thorough manual review of every discovered chain to verify the accuracy of these findings. We found that one of the chains was a false positive. The discrepancy arose because the static analyzer incorrectly resolved a complex conditional branch within a function's control flow, leading it to identify an execution path that is not actually reachable in practice. After discarding this invalid path, we were left with 15 theoretically viable exploit chains.

Furthermore, to confirm their real-world exploitability, we developed 15 end-to-end exploits for each of the exploit chains using CVE-2022-20409 and CVE-2023-48409. For the 12 chains capable of mapping arbitrary physical memory with R/W permissions, each exploit achieves full privilege escalation. For the remaining 3 chains that allow RO mapping, we created a proof-of-concept(POC) that leaks arbitrary kernel memory, successfully demonstrating a bypass of KASLR.

Regarding false negatives, we found that the set of chains discovered by our analyzer fully encompassed a collection of exploit chains that we had previously identified through manual auditing. Our methodology does its best to minimize false negatives. We ensured the completeness of mapping interfaces based on standard driver implementation patterns. From that foundation, our subsequent analyzes were performed comprehensively across the entire codebase of the Mali GPU driver, covering both its JM and CSF architectures.

V. Exploitability Evaluation

To systematically evaluate the generality and effectiveness of PhantomMap in real-world exploitation, we designed and conducted a comprehensive exploitability study. We selected 13 representative and up-to-date real-world vulnerabilities. We choose these CVEs from widely recognized vulnerability databases[31] and exploitation collections[32], focusing on those published within the last three years and confirmed to be triggerable on mobile devices equipped with Mali GPUs. These selected CVEs incorporate all common vulnerability types (UAF, Double Free, OOB). The selection of vulnerabilities includes examples from both the Android kernel and the Mali GPU drivers.

Furthermore, we assume a hardened environment where all mitigations claimed in our threat model are enabled, along with unprivileged user namespace restrictions. Building on this, we confirmed the exploitability of both existing and PhantomMap-based exploits for the 13 CVEs, with detailed results presented in TABLE IV.

Exploitability. The result shows that PhantomMap outperforms existing exploits in its ability to bypass modern defenses. Notably, for CVE-2023-20938 and CVE-2022-20421, the vulnerabilities represent a special case. Because the vulnerable objects reside in dedicated caches and therefore need a cross-cache attack, SLAB_VIRTUAL has prevented the initial capability pivot required to begin exploitation, which is a prerequisite step for any subsequent attack. This renders the vulnerability unexploitable by both existing exploits and the PhantomMap attack.

To demonstrate practical exploitability, we developed 15 full end-to-end exploits that achieve kernel privilege escalation on contemporary devices, including Google Pixel 6/7 and Samsung Galaxy A71, thereby confirming the attack's real-world feasibility. Our exploits target all three common vulnerability types: UAF (CVE-2025-21836, CVE-2022-38181) and Double Free (CVE-2022-20409) in the Android kernel, and OOB (CVE-2023-48409) in the Arm Mali GPU driver. Among these, CVE-2025-21836 is a limited vulnerability. To the best of our knowledge, we have not found any public exploits for it. Despite this, we successfully developed an exploit for it using PhantomMap, proving the power and effectiveness in handling challenging and previously unexploited vulnerabilities. Moreover, we successfully achieved privilege escalation on a Samsung Galaxy A71 with both RKP and DEFEX enabled, by exploiting CVE-2022-38181 with PhantomMap. This case further validates PhantomMap's practical effectiveness against heavily fortified devices.

For the remaining CVEs, we confirmed that they can be similarly exploited based on their comparable vulnerability primitives. However, developing a full exploit for each one requires significant manual effort, including frequently flashing device kernels and porting the relevant vulnerability patches for each test environment. Therefore, to maintain a feasible scope, we focused on the representative cases for full exploitation development.

Simplicity & Stability. PhantomMap offers substantial improvements in simplicity and stability compared to traditional kernel exploitation techniques, which are often encumbered by two major challenges. The first challenge is the reliance on information leaks. Traditional exploitation methods often require leveraging the vulnerability itself to pivot to an information leak primitive before the main exploit proceeds. This means that the attacker should trigger the vulnerability once more, which not only adds a layer of complexity but also impairs reliability. For race condition vulnerabilities, such as CVE-2025-21836, this extra trigger significantly lowers the probability of success and leads to system instability and crashes. The second challenge is the dependency on precise memory layouts. Many existing exploits are contingent on complex heap or page layouts that are difficult to reproduce reliably in dynamic, real-world systems, further degrading

TABLE IV: Exploitability demonstrated on 13 Real-World CVEs using PhantomMap. ⋆ means that the vulnerable kernel heap resides in a dedicated cache and therefore requires a cross-cache attack. CVE-2025-21836 does not have public exploits.

		Existing Exploits		PhantomMap Exploit	
CVE ID	**Type**	**No Leak Required**	**Bypass Mitigations**	**No Leak Required**	**Bypass Mitigations**
CVE-2025-21836	UAF	—	—	✔	✔
CVE-2024-46740	UAF	✔	✔	✔	✔
CVE-2024-26582	Double Free	✗	✔	✔	✔
CVE-2023-32882	OOB	✗	✔	✔	✔
CVE-2023-6560	OOB	✔	✗	✔	✔
CVE-2023-32837	OOB	✗	✗	✔	✔
CVE-2023-32832	UAF	✗	✔	✔	✔
CVE-2023-48409	OOB	✗	✔	✔	✔
CVE-2023-20938⋆	UAF	✗	✗	✔	✗
CVE-2023-0266	UAF	✗	✔	✔	✔
CVE-2022-38181	UAF	✔	✔	✔	✔
CVE-2022-20421⋆	UAF	✗	✗	✔	✗
CVE-2022-20409	Double Free	✗	✗	✔	✔

their success rates.

PhantomMap directly addresses these limitations. By design, it operates without requiring any information leak and is not dependent on complex memory layouts, thereby simplifying the exploitation process and enhancing its stability. In our evaluation, when targeting CVE-2023-48409, a vulnerability previously exploited with a complex kernel page layout, we compared our PhantomMap-based exploit with the existing one. In 20 trials for each, PhantomMap elevated the success rate from a mere 45% to 90%. Our exploits for CVE-2022-20409 also demonstrate high reliability, with a consistent success rate of over 95%. Finally, in our exploit for CVE-2025-21836, PhantomMap requires only a single trigger of the race condition vulnerability to achieve privilege escalation, bypassing the step for a separate info-leak and thus ensuring higher system stability.

Moreover, under busy-system conditions, noise in kernel SLUB-allocator destabilizes complex heap layouts, thereby reducing exploit reliability. As a GPU-assisted kernel exploitation technique, PhantomMap offloads some exploitation steps to the GPU, making it resilient to such noise and enhancing exploit reliability even under high system load. In our experiments simulating a busy system by running stress-ng, the success rate of existing exploits of CVE-2022-20409 dropped to 20%, while our PhantomMap exploit achieved a 50% success rate under the same conditions.

VI. Defense Against PhantomMap

The efficacy of the PhantomMap attack stems from its exploitation of two fundamental design flaws within the Arm Mali GPU driver ecosystem: (1) the decoupled allocation-mapping workflow that creates an attacker-controllable time window, and (2) the absence of physical page validation before mapping operations. Effective mitigation strategies must address these root causes. While an ideal theoretical solution would establish robust hardware-level isolation (e.g., via IOMMU/SMMU or GPU TEEs) to compartmentalize CPU and GPU memory domains, thereby fundamentally preventing cross-domain remapping. However, we examined open-source device trees and on-device configurations for popular Mali-based phones, including the Google Pixel 6/7 and Samsung Galaxy A71, and found that their Mali GPUs are not connected to the system's IOMMU/SMMU. This appears to be a result of System-on-Chip (SoC) integration choices rather than a simple software configuration issue. Furthermore, even if IOMMU is enabled in the future, PhantomMap would still remain effective. Our exploit chains include paths that exploit the legitimate driver workflow to update both the system IOMMU page tables and the Mali GPU's own MMU page tables, allowing for arbitrary physical memory remapping into the GPU with read/write permissions.

Alternatively, eliminating the attack window through extensive refactoring of the Mali driver to enforce tight coupling of allocation and mapping operations proves suboptimal. This approach would incur substantial engineering effort and risk destabilizing the driver's core functionalities, as the decoupled design is intrinsically tied to performance and architectural requirements. Consequently, we argue that practical defenses should prioritize enforcing physical page validation at critical mapping interfaces. In the following, we first analyze the limitations of existing mitigations against PhantomMap, then introduce our lightweight in-driver solution designed to enforce page validation with minimal performance impact.

A. Limitations of Existing Potential Mitigations

Samsung RKP. A prominent defense is Samsung's Knox with Real-time Kernel Protection (RKP), which leverages the hypervisor to mark critical kernel data and page tables as read-only. However, this approach is ineffective against PhantomMap. The fundamental flaw is that RKP is designed to validate the privilege of the caller, not the integrity of the data used in the request. Our attack exploits this by first corrupting unprotected non-control data (key structures). Subsequently, a legitimate kernel workflow uses this tainted data and makes a valid request to the secure world, for instance, to remap a malicious physical address. Since the call originates from a trusted kernel function, RKP approves the operation, unknowingly compromising the page table. Moreover, RKP's Stage-2 protection for critical regions fails to defeat PhantomMap. As discussed in §III-E, PhantomMap can achieve privilege escalation by modifying unprotected read-only critical data structures outside the guarded set.

MTE. Memory Tagging Extension (MTE) is a hardware feature designed to mitigate runtime memory corruption vulnerabilities by enforcing memory access correctness through tags[33]. Its defensive scope is primarily confined to the vulnerability triggering phase. The focus of this work, in contrast, is the post-exploitation phase, and our threat model explicitly assumes that an attacker has already triggered a vulnerability. Therefore, the challenge of bypassing initial corruption mitigations like MTE is orthogonal to our scope.

In the post-exploitation stage, PhantomMap is not thwarted by MTE. The core of its operation involves hijacking valid kernel workflows to perform legitimate GPU page table updates and DMA write operations. Since these operations are inherently compliant with memory access policies, they naturally satisfy MTE's tag verification. Thus, PhantomMap sustains its ability to achieve privilege escalation on MTE-enabled systems.

B. Lightweight Mitigation

Our lightweight mitigation is designed to target this fundamental design flaw by verifying the mapping operations in the Mali GPU driver.

Design & Implementation. Our mitigation introduces a type-checking mechanism during memory mapping operations to prevent unauthorized access to non-GPU memory. The core idea is to ensure that only memory explicitly allocated for the GPU can be mapped by the Mali GPU driver.

The Mali kernel driver allocates and frees memory at the page level through wrapper functions based on top of the standard `alloc_page()` and `free_page()` calls. And we leverage a reused field `page_type` within the `struct page`, which is part of a union that also includes `_mapcount`. Pages allocated via the standard `alloc_page` from the freelist should not have a `page_type` assigned, and since the Mali GPU driver maintains its own separate map count, the `_mapcount field` is also unused by the GPU. This provides an opportunity to reuse this field to introduce a custom page type in the Mali GPU driver. We define a new type, `TYPE_GPU`, and hook the root physical memory allocation and deallocation interfaces within the Mali driver, `mgm_alloc_page` and `mgm_free_page`. When a physical page is allocated to the Mali GPU Driver, we mark it with `TYPE_GPU`. Conversely, when the page is freed back to the buddy system, we reset its type to an invalid value. Crucially, at all mapping interfaces on the Mali GPU driver side, which we identified in §IV-A, we introduce a verification step. Before any mapping operation can proceed, we check whether the physical page intended for mapping is marked with `TYPE_GPU`. This check ensures that only pages explicitly allocated for the GPU can be mapped, effectively preventing the illegal mapping of the non-GPU pages.

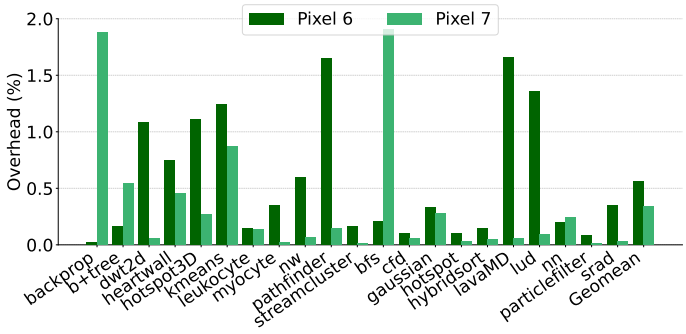

Fig. 6: Overhead of our mitigation on Rodinia benchmarks.

Overhead Evaluation. To evaluate the performance overhead of our proposed lightweight mitigation, we employ the Rodinia[34] GPU benchmark suite, which is widely used for performance evaluation on Arm devices[35], [36], [37]. Our overhead evaluations were conducted on two distinct Google Pixel devices to cover different hardware and software architectures: (1) a Pixel 6, equipped with a Mali-G78 GPU, running *gs-android-gs-raviole-mainline* to test the JM architecture, and (2) a Pixel 7, equipped with a Mali-G710 GPU, running *android-gs-pantah-5.10-android14-qpr3* to test the CSF architecture.

We ported the entire Rodinia benchmark suite to mobile platforms, comprising 21 applications that represent a wide spectrum of computational workloads. To ensure statistical reliability, each benchmark was executed 30 times, and the average execution time is reported as the final performance metric. As illustrated in Fig. 6, our mitigation introduces minimal performance overhead, averaging only 0.56% on the Pixel 6 and 0.34% on the Pixel 7. Detailed per-bench breakdown results, along with standard deviations, are provided in TABLE V, which lists the mean execution time (mean) and standard deviation (std) for each benchmark, both with the mitigation applied (patch) and without (pre), as well as the precise performance overhead percentage (Overhead %) calculated from this data.

Furthermore, to verify that our lightweight mitigation does not introduce functional regressions or compromise system stability, we executed the Android Linux Test Project (LTP) test suite on both Pixel 6 and Pixel 7 devices. To ensure

TABLE V: Overhead details of our mitigation on rodinia benchmarks. The final geometric mean overhead was 0.56% on the Pixel 6 (JM) and 0.34% on the Pixel 7 (CSF).

Benchmark	Total exec time (ms)								Overhead (%)	
	Pixel 6 (JM)				Pixel 7 (CSF)				Pixel 6	Pixel 7
	pre		patch		pre		patch			
	mean	std	mean	std	mean	std	mean	std		
backprop	3622.05	111.49	3622.80	121.71	678.43	20.41	691.19	4.53	0.02	1.88
b+tree	55.13	18.96	55.23	19.44	39.07	7.08	39.29	6.74	0.16	0.54
dwt2d	438.45	10.24	443.20	9.12	486.57	15.59	486.86	10.49	1.08	0.06
heartwall	11744.08	199.27	11831.88	57.98	14588.54	204.19	14655.04	217.07	0.75	0.46
hotspot3D	1142.37	61.89	1155.02	62.26	3042.95	189.36	3051.31	129.69	1.11	0.27
kmeans	2699.82	70.33	2733.56	68.87	1664.30	43.01	1678.72	40.78	1.24	0.87
leukocyte	78.62	22.24	78.74	23.44	100.73	36.46	100.87	36.69	0.15	0.14
myocyte	41468.61	6237.74	41617.88	7832.05	59939.92	549.13	59954.58	463.74	0.35	0.02
nw	4621.69	124.27	4649.58	169.10	746.89	15.25	747.45	14.83	0.60	0.07
pathfinder	224.34	13.43	228.04	4.82	166.15	7.42	166.39	8.06	1.65	0.15
streamcluster	57267.00	1794.47	57362.06	2056.53	61137.52	569.19	61145.60	731.20	0.16	0.01
bfs	158.47	5.08	158.80	6.55	106.35	10.19	108.38	11.88	0.21	1.91
cfd	93323.41	8085.78	93423.81	9408.65	96634.35	460.21	96695.17	1168.48	0.10	0.06
gaussian	22407.41	754.30	22481.76	1201.25	22566.46	331.70	22630.75	304.88	0.33	0.28
hotspot	436.49	9.06	436.94	6.54	495.49	9.47	495.63	11.17	0.10	0.03
hybridsort	248.89	11.19	249.27	16.29	195.20	8.57	195.30	6.78	0.15	0.05
lavaMD	9335.476	475.58	9490.16	385.991	7833.29	5.12	7838.15	20.26	1.66	0.06
lud	13465.81	451.67	13649.24	600.35	18149.16	136.22	18165.14	102.64	1.36	0.09
nn	39.13	7.79	39.21	7.93	38.70	3.89	38.80	4.27	0.20	0.24
particlefilter	45659.71	659.20	45696.52	438.89	33031.40	46.64	33035.27	45.90	0.08	0.01
srad	8976.59	77.74	9008.61	90.09	9627.56	241.67	9630.47	137.78	0.35	0.03
Geo-mean	—								**0.56**	**0.34**

the GPU was under active load, we first ran the Rodinia benchmark and immediately followed it with the complete LTP suite. The test outcomes were identical both before and after enabling our mitigation, confirming that our solution maintains full system correctness.

VII. Related Work

We categorize related work into three areas: general kernel exploitation techniques, kernel code injection and execution techniques, and prior exploitation of the Mali GPU driver. In this section, we provide an overview of each category and highlight the distinctions between our work and prior works.

A. Kernel Exploitation Techniques

DirtyCred[24] pioneered privilege escalation by swapping a low-privileged credential with a high-privileged one in memory. Page Spray[28] introduced a reliable memory-corruption primitive by spraying pages with user-controlled data to target kernel objects. ExpRace[38] transforms inherently hard-to-exploit kernel data races into dependable vulnerabilities through precise interrupt manipulation. PSPRAY[39] leverages a timing side-channel to vastly improve heap-based exploit success rates, while SCAVY[40] automates the discovery of kernel memory-corruption targets for privilege escalation using scalable fuzzing and differential analysis. These techniques achieve privilege escalation and stabilize exploits, but none enable arbitrary kernel code execution. In contrast, Code Injection & Execution Techniques enable more powerful and flexible post-exploitation capabilities.

B. Code Injection & Execution Techniques

Previous kernel exploits often rely on control-flow hijacking to achieve code execution. For instance, ret2usr[41] redirects kernel execution to user-space payloads by forging return addresses outside the kernel. ret2dir[42] bypasses simpler protections by forcing the kernel to return to its direct-mapped memory region, which executes a payload that an attacker has placed in a physical page via user-space. RetSpill[27] leverages register spill side effects to atomically place a controlled ROP chain on the stack for hijacking.

However, CFI and FineIBT[43] defenses now block such hijacking methods, driving the development of data-only kernel code injection and execution techniques. Interp-flow Hijacking[25] hijacks the eBPF bytecode interpreter via tail-call to bypass kernel CFI and execute malicious bytecode within the kernel. USMA[26] leverages a user space mapping attack to map and patch kernel code in user space. Dirtypagetable[29] is a data-only exploit that uses heap vulnerabilities to corrupt user pagetable entries, yielding arbitrary physical read/write primitives. SLUBStick[30] exploits a timing side-channel in the SLUB allocator to perform a reliable cross-cache attack, then converts heap vulnerabilities into a page-table manipulation for full arbitrary memory R/W.

C. Mali GPU Exploitation

Google Project Zero and security researchers uncovered and exploited some vulnerabilities in the Mali GPU driver[5], [6], [3], [44], [4], [45]. These exploits rely on the traditional technique of reclaiming a use-after-free (UAF) page as a GPU page table and corrupting it. As we mentioned earlier, this approach heavily relies on high-impact vulnerabilities to tamper with the page table directly. Starlab[46] demonstrated how to pivot certain heap vulnerability capabilities into page-level UAF. It requires a field-precise overwrite capability that updates one designated structure member, which only some bugs can provide, thereby limiting adaptability across different vulnerabilities.

In contrast to prior techniques, PhantomMap neither depends on high-impact vulnerabilities nor requires extra capabilities and complex pivot stage, delivering true arbitrary code execution in kernel space, unlocking far more powerful post-exploitation capabilities. Existing kernel code injection techniques are routinely thwarted by modern mitigations such as restricted access control and hypervisor-based page table protections. PhantomMap introduces a fundamentally novel GPU-Assisted exploit technique that exploits a previously unaddressed design flaw in the Mali GPU architecture, bypassing both software and hardware defenses to achieve full kernel control.

VIII. Conclusion

This paper conducts a systematic analysis of the memory mapping mechanism of the Mali GPU driver, revealing design flaws that stem from both the decoupling of memory allocation and mapping operations and the missing physical address validation. Based on these attack surfaces, we propose a novel exploitation technique, PhantomMap, which can easily pivot limited kernel heap vulnerability capabilities into powerful arbitrary physical memory read/write primitives, enabling stable kernel privilege escalation with multiple mainstream security mitigations enabled. To systematically identify such attack surfaces, we designed and implemented a static analyzer that successfully detected 15 distinct exploit chains. Through the evaluation of 13 representative and up-to-date real-world CVEs and the development of 15 end-to-end exploits, we validated the universality and efficiency of PhantomMap. Finally, we discuss the limitations of existing mitigation strategies and propose a lightweight and efficient approach to minimize the impact of PhantomMap.

IX. Ethics Considerations

We have contacted the Arm product security incident response team and responsibly disclosed the exploitation technique detailed in this paper. To facilitate a swift resolution, we provided Arm with a comprehensive attack surface analysis report, the corresponding exploits, system images for reproduction, and patches of our proposed mitigation. Additionally, all bugs that we used and exploited in the evaluation have already been fixed. All experiments were conducted exclusively on hardware under our control, within a secure, isolated laboratory environment, thereby posing no risk to external systems or users. The primary objective of our work is to enhance the security design of Arm Mali GPU drivers.

Acknowledgment

The authors would like to thank our shepherd and reviewers for their insightful comments. Those comments helped to reshape this paper. This work is partially supported by the National Natural Science Foundation of China (Grants No. 62572432 and No. 62532012).

References

[1] "Global smartphone chipsets market share," https://www.counterpointresearch.com/insight/global-smartphone-apsoc-market-share-quarterly.

[2] "Operation triangulation: The last (hardware) mystery," https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/.

[3] "Security researcher at github security lab," https://github.blog/author/mymo/.

[4] "Make ksma great again: The art of rooting android devices by gpu mmu features," https://i.blackhat.com/BH-US-23/Presentations/US-23-WANG-The-Art-of-Rooting-Android-devices-by-GPU-MMU-features.pdf, 2023.

[5] "Mind the gap," https://googleprojectzero.blogspot.com/2022/11/mind-the-gap.html.

[6] "Arm mali csf: refcount-overflow-leading-to-physical-uaf bugfix in r43p0," https://project-zero.issues.chromium.org/issues/42451624, 2023.

[7] "Reverse-engineering the mali g78," https://www.collabora.com/news-and-blog/news-and-events/reverse-engineering-the-mali-g78.html.

[8] "Mali for all occasions: New gpus for all graphics workloads, use cases and consumer devices," https://community.arm.com/arm-community-blogs/b/mobile-graphics-and-gaming-blog/posts/new-suite-of-arm-mali-gpus.

[9] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 335–350.

[10] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 90–102.

[11] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm." in *NDSS*, vol. 16, 2016, pp. 21–24.

[12] "Kernel address space layout randomization [lwn.net]," https://lwn.net/Articles/569635/.

[13] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 941–955.

[14] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "{PAC} it up: Towards pointer integrity using {ARM} pointer authentication," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 177–194.

[15] Y. Yang, J. Tu, W. Shen, S. Zhu, R. Chang, and Y. Zhou, "kcpa: Towards sensitive pointer full life cycle authentication for os kernels," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 3768–3784, 2023.

[16] "Pan, privileged access never," https://developer.arm.com/documentation/ddi0601/2024-12/AArch64-Registers/PAN--Privileged-Access-Never.

[17] "Supervisor mode access prevention [lwn.net]," https://lwn.net/Articles/517475/.

[18] "Introduce static_usermodehelper to mediate call_usermodehelper()," https://patchwork.kernel.org/project/linux-hardening/patch/20170116165044.GC29693@kroah.com/.

[19] P. Ning, "Samsung knox and enterprise mobile security," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014, pp. 1–1.

[20] "mm/slub: allocate slabs from virtual memory," https://patchwork.kernel.org/project/linux-mm/patch/20230915105933.495735-12-matteorizzo@google.com/.
[21] "Security-enhanced linux," https://en.wikipedia.org/wiki/Security-Enhanced_Linux.
[22] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1165–1184.
[23] "How i use a novel approach to exploit a limited oob on ubuntu at pwn2own vancouver 2024," https://u1f383.github.io/slides/talks/2024_POC-How_I_use_a_novel_approach_to_exploit_a_limited_OOB_on_Ubuntu_at_Pwn2Own_Vancouver_2024.pdf.
[24] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, 2022, pp. 1963–1976.
[25] Q. Liu, W. Shen, J. Zhou, Z. Zhang, J. Hu, S. Ni, K. Lu, and R. Chang, "Interp-flow hijacking: Launching non-control data attack via hijacking ebpf interpretation flow," in *European Symposium on Research in Computer Security*. Springer, 2024, pp. 194–214.
[26] "Usma: Share kernel code with me," https://i.blackhat.com/Asia-22/Thursday-Materials/AS-22-YongLiu-USMA-Share-Kernel-Code.pdf, 2022.
[27] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Retspill: Igniting user-controlled data to burn away linux kernel protections," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3093–3107.
[28] 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 *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1189–1206.
[29] "Dirty pagetable: A novel exploitation technique to rule linux kernel," https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html, 2023.
[30] 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 *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4051–4068.
[31] "A collection of links related to linux kernel security and exploitation," https://github.com/xairy/linux-kernel-exploitation?tab=readme-ov-file.
[32] "kernelctf," https://github.com/google/security-research/tree/master/pocs/linux/kernelctf.
[33] "Arm memory tagging extension," https://source.android.com/docs/security/test/memory-safety/arm-mte.
[34] "gpu-rodinia," https://github.com/yuhc/gpu-rodinia.
[35] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "Strongbox: A gpu tee on arm endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 769–783.
[36] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim, "Securing gpu via region-based bounds checking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 27–41.
[37] H. Lee, H. Kim, C. Kim, H. Han, and E. Seo, "Idempotence-based preemptive gpu kernel scheduling for embedded systems," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 332–346, 2020.
[38] Y. Lee, C. Min, and B. Lee, "{ExpRace}: Exploiting kernel races through raising interrupts," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2363–2380.
[39] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee, "Pspray: Timing {Side-Channel} based linux kernel heap exploitation technique," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6825–6842.
[40] E. Avllazagaj, Y. Kwon, and T. Dumitraș, "{SCAVY}: Automated discovery of memory corruption targets in linux kernel for privilege escalation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7141–7158.
[41] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "{kGuard}: Lightweight kernel protection against {Return-to-User} attacks," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 459–474.
[42] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 957–972.
[43] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "Fineibt: Fine-grain control-flow enforcement with indirect branch tracking," 2023. [Online]. Available: https://arxiv.org/abs/2303.16353
[44] "Simple bug but not easy exploit: Roo4ng android devices in one shot," https://powerofcommunity.net/poc2023/YongWang.pdf.
[45] "Arm mali (mostly ¿=r34p0): page tables freed before pte removal," https://project-zero.issues.chromium.org/issues/42451466.
[46] "Gpuaf - using a general gpu exploit tech to attack pixel8," https://www.scribd.com/document/798244931/GPUAF-Using-a-general-GPU-exploit-tech-to-attack-Pixel8.