Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels

Lukas Maar
Graz University of Technology

Florian Draschbacher
Graz University of Technology and A-SIT Austria

Lukas Lamster
Graz University of Technology

Stefan Mangard
Graz University of Technology

Abstract

With the mobile phone market exceeding one billion units sold in 2023, ensuring the security of these devices is critical. However, recent research has revealed worrying delays in the deployment of security-critical kernel patches, leaving devices vulnerable to publicly known one-day exploits. While the mainline Android kernel has seen an increase in defense mechanisms, their integration and effectiveness in vendor-supplied kernels are unknown at a large scale.

In this paper, we systematically analyze publicly available one-day exploits targeting the Android kernel over the past three years. We identify multiple exploitation flows representing vulnerability-agnostic strategies to gain high privileges. We then demonstrate that integrating defense-in-depth mechanisms from the mainline Android kernel could mitigate 84.6 % of these exploitation flows. In a subsequent analysis of 994 devices, we reveal a widespread absence of effective defenses across vendors. Depending on the vendor, only 28.8 % to 54.6 % of exploitation flows are mitigated, indicating a 4.62 to 2.95[1] times worse scenario than the mainline kernel.

Further delving into defense mechanisms, we reveal weaknesses in vendor-specific defenses and advanced exploitation techniques bypassing defense implementations. As these developments pose additional threats, we discuss potential solutions. Lastly, we discuss factors contributing to the absence of effective defenses and offer improvement recommendations. We envision that our findings will guide the inclusion of effective defenses, ultimately enhancing Android security.

1 Introduction

Over the past decade, the mobile phone market has reached an all-time high, with more than one billion units sold in 2023. Given our daily reliance on mobile phones for communication, financial transactions, and personal data storage, this surge in device adoption underscores the critical need for robust security measures protecting these devices.

[1]Factors of $\frac{1-0.288}{1-0.846}$ and $\frac{1-0.546}{1-0.846}$, respectively.

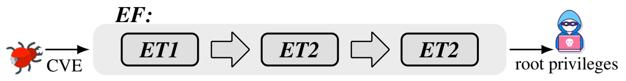


Figure 1: The exploitation flow ***EF*** is a vulnerability-agnostic chain of exploitation techniques ***ET***, with one ***ET*** elevating a primitive to a more powerful form [8]. ***EF*** leverages the capabilities of a vulnerability to gain root privileges ultimately.

Despite the importance of mobile security, recent studies [13, 27, 43, 61, 67] have revealed that Android's security-critical kernel patches often lag significantly behind the mainstream Linux kernel. In over 20 % of cases, delays exceeding one year occur [61], mainly due to the downstream approach of most Android vendors. This delayed deployment of security-critical patches creates opportunities for malicious actors to attack the Android Linux kernel. While these attacks would be classified as one-day exploits due to the known nature of their vulnerabilities, they effectively function as zero-day exploits during the extensive unpatched period. The severity of this situation is underscored by findings from Google Project Zero [9, 49] and Threat Analysis Group [50], which highlight a prevalence of exploits in the wild targeting these unpatched vulnerabilities in the Android kernel.

On the defensive side, the mainline Android kernel has seen an increase in vulnerability-agnostic defenses preventing one-day exploits. While these defense-in-depth mechanisms may be readily available, *their integration and effectiveness in vendor-supplied kernels are unknown*. Consider, for example, the case of the Pegasus spyware. Using BadBinder, an exploit [46] known since 2019, malicious actors can infect target devices with their payload. While an effective defense has been available for over 10 years [47], its rollout status in vendor-provided kernels is entirely opaque. The questionable deployment or absence of such defenses leaves devices vulnerable to one-day exploitation flows (see Figure 1), thus creating a significant security gap in the Android ecosystem. Malicious actors can exploit this and mount attacks against insufficiently protected devices based on public exploits.

In this paper, we address the inadequate protection of Android devices against one-day exploitation flows through comprehensive analysis. We systematically analyze all publicly available one-day exploits targeting memory safety vulnerabilities in the Android Linux kernel over the past three years, comprising 26 exploits. In doing so, we unveil the diversity of these one-day exploits and classify 10 distinct exploitation techniques. In a subsequent analysis, we examine 8 defense-in-depth mechanisms present in the mainline Android kernel and find that they effectively prevent 84.6 % of the previously identified one-day exploitation flows. This percentage serves as the ground truth for how secure mobile devices could be if their kernels were up to date with the defenses enabled. Given the maximum achievable security, we can *quantify the level of security that is actually reached in Android devices*.

For this, we conduct the first large-scale analysis on kernel-level defense-in-depth mechanisms for Android devices via a mostly automated approach. We demonstrate a widespread absence across vendors and uncover flaws in vendor-specific defenses. In our analysis, we cover Android devices from all top 7 vendors (e.g., Samsung, Xiaomi, and Huawei), along with three recognized vendors (i.e., Google, OnePlus, and Fairphone), covering more than 84 % of the global Android device share [6]. We analyze devices from 2018 to 2023 using Android versions 9 to 14 and kernels ranging from v3.10 to v6.1. In total, we analyze 994 device firmwares and 1533 Android kernel source codes. Our results suggest that *the level of security that is actually reached is severely lacking* compared to the mainline Android kernel.

Our work presents four novel findings. First, we provide in-depth insights into the absence of effective defenses in vendor-provided kernels. On average, only 41.5 % of our analyzed one-day exploitation flows can be mitigated. This varies across vendors, from 28.8 % for the least (i.e., Fairphone) to 54.6 % for the most secure (i.e., Google) vendor, indicating a 4.62 to 2.95 times worse scenario than the ground truth.

Second, we unveil advancements in two exploitation techniques, enabling malicious actors to bypass the defense intended against the base technique. These advancements are applicable in all one-day exploitation flows that use the base technique. While these advancements pose additional threats to Android devices, we discuss potential mitigations.

Third, we uncover 4 and 2 distinct weaknesses in Samsung's and Huawei's vendor-specific defenses, respectively. These issues impact Samsung devices ranging from Galaxy A04/A14 to Galaxy S23 5G/Ultra, and, thus, the entire range of low-end to high-end devices, as well as the entire range of Huawei devices. We demonstrate that these defenses do not fully prevent the targeted exploitation technique, or we demonstrate modified exploits that bypass the defense.

Fourth, we discuss factors that may contribute to the lack of effective defenses. While we observe a correlation between older kernel versions and higher one-day susceptibility, we reveal that susceptibility extends beyond mere version correlation. We present factors such as a lack of importance of security features and vulnerable configurations, as well as performance costs (confirmed by Google, Samsung, and Huawei), which are particularly relevant for low-end devices. We also make recommendations to Google and downstream vendors to improve Android security.

We open source[2] our tools that detect the widespread lack of included and effective defenses.

Contributions. The main contributions of our work are

(1) One-Day Exploitation Insights: We analyze 26 one-day exploits and classify 10 different exploitation techniques.

(2) Defense Insights: Based on these insights, we demonstrate defenses for the identified techniques.

(3) Defense Inclusion and Effectiveness Analysis: We unveil a significant gap between the maximum available security and that reached by vendor-supplied kernels.

(4) Novel Findings: We present in-depth insights into the absence of effective defenses in vendor-supplied kernels, exploitation advancements, weaknesses, and factors likely contributing to the absence of defense.

Disclosure. We disclosed our findings to all 10 vendors. While some did not respond (e.g., Oppo and Xiaomi), others (i.e., Google, Fairphone, Motorola, Huawei, and Samsung) acknowledged our findings (fully or partially), and some of these patched unsecured phones to enhance Android security.

Outline. Section 2 provides background. Section 3 shows the high-level workflow. Section 4 presents the one-day analysis and defense identification. Section 5 presents the large-scale defense analysis. Sections 6 and 7 discuss potential solutions and related work. Section 8 concludes our work.

2 Background

2.1 Android Ecosystem and Android Kernels

Android is primarily designed for mobile devices and undergoes active development led by Google. The Android kernel is based on the Linux kernel. For major platform releases, Google specifies compatible launch kernels for new devices and upgrades kernels for existing device updates.

Historically, vendors maintained separate Linux kernel trees for each product model, hindering upstream bug fixes due to vendor-specific code and hardware drivers. Despite the introduction of monthly Android Security Bulletins in 2015, prior research [24, 67] indicates continued delay in patch integration. In response, Google introduced the Generic Kernel Image (GKI) project in Android 11 on kernel versions above or equal to v5.4, aiming to overcome slow patch adoption. This initiative separates the Android kernel into a hardware-agnostic core maintained by Google and vendor-specific modules loaded dynamically. Moreover, it restricts the Android kernel to some constraints, such as ABI compatibility.

[2] https://github.com/IAIK/DefectsInDepth.

2.2 Kernel Exploitation

Fundamental Kernel Defenses. The Linux kernel employs defense-in-depth mechanisms to make vulnerability exploitation more difficult. These are included via the configuration file `.config`. One fundamental defense is the W^X policy, which dictates that sections may never be writeable and executable. Consequently, an attacker cannot simply inject instructions for privilege escalation. Kernel Address Space Layout Randomization (KASLR) randomizes the location of binary sections at boot time. Thus, an exploit typically breaks KASLR through a read primitive or a side channel [21]. Lastly, Privilege Access Never (PAN) prevents access to user-accessible memory while in kernel space, mitigating the control-flow redirection to userspace.

Kernel Exploitation on Android. The exploitation flow (see Figure 1) of most Android kernel exploits consists of three stages: First, an adversary *breaks KASLR* to identify the locations of critical structures. Second, the adversary obtains an *arbitrary read-and-write primitive* that allows them to perform the third step, which is gaining *full root privileges*.

To *break KASLR*, an adversary typically triggers a memory safety vulnerability, e.g., Use-After-Free (UAF) or Out-Of-Bounds (OOB) access, to leak a kernel address. By knowing the Android kernel binary under attack, the adversary then computes the kernel base address. Depending on how powerful the underlying vulnerability is, the adversary either continues or re-triggers this (or another) vulnerability to obtain an *arbitrary read-and-write primitive*. They then typically manipulate credentials to elevate their privileges. Furthermore, they tamper with kernel memory to disable SELinux's Mandatory Access Control (MAC), obtaining *full root privileges*.

Kernel Heap Attacks. Since most memory-safety vulnerabilities concern heap-allocated memory [65], dynamically allocated during runtime, it is a popular attack target.

Use-After-Free. UAF vulnerabilities occur when a resource that is still referenced is freed. A typical UAF exploit works as follows: First, an adversary causes the memory slot of a *vulnerable object* that is still in use to be freed. Freeing the memory slot causes the allocator to reuse the slot for future allocations. Second, they allocate a *reallocated object* such that the vulnerable and reallocated objects simultaneously use the previously freed slot. Third, they use either the vulnerable or the reallocated object to obtain a read or write primitive for the slot. Exploiting a Double-Free (DF) or Invalid-Free (IF) vulnerability (which are special cases of a UAF, where the slot is either freed twice or with an offset) works similarly.

In practice, several challenges render such attacks more difficult to execute. Most vulnerabilities grant only weak write capabilities, such as zeroing out memory at a particular offset. Additionally, to successfully exploit a UAF, the adversary requires knowledge of how the kernel's allocator (i.e., slab allocator) reuses memory slots.

There are generally two ways to exploit this reuse: With *in-cache reuse*, the adversary reuses the freed memory slot for another object that lives in the same slab cache. This only works in caches that contain the vulnerable and reallocated objects, e.g., `kmalloc-*` caches. Hence, the adversary is limited to objects that have the same (or similar) size and the same allocation properties as the vulnerable object.

The other way is to use a *cross-cache reuse* [33, 38, 60] attack. Here, the adversary frees all slots of a slab page, prompting the slab allocator to return the slab page that contains the vulnerable object to the page allocator. The page is then allocated either as a different type of page or to another slab cache. This allows them to reuse a memory slot between slab caches of different types, allocation sizes, and allocation properties.

Out-Of-Bounds. Exploiting an OOB vulnerability [12, 65] with write capabilities follows a similar process. An adversary triggers the OOB write, often in the form of a linear overflow, to manipulate sensitive data in an adjacent memory slot (i.e., victim object). This sensitive data typically references a vulnerable object, e.g., through a reference counter or data pointer [38, 42]. The adversary then forces the memory slot of the vulnerable object into a state where it is referenced twice. This upgrades the OOB write to be exploited analogously to the UAF three-stage exploitation flow typically.

3 High-Level Workflow

This section presents the high-level workflow of our study, depicted in Figure 2. It consists of three main components: the *One-Day Exploitation Analysis* and *Defense Inclusion and Effectiveness Analysis*, both of which yield *Novel Findings*.

In the *One-Day Exploitation Analysis* (see Section 4), we manually analyze all publicly available one-day exploits targeting memory safety vulnerabilities in the Android Linux kernel from the last three years. Our goal is to identify the exploitation flows employed in these exploits. In this context, we refer to an exploitation flow (see Figure 1) as a vulnerability-agnostic chain of exploitation techniques that exploit a vulnerability to gain full root privileges. An exploitation technique is a reusable and reasonably generic strategy for transforming an exploit primitive into a more powerful one [8]. In our study, we analyze 26 one-day exploits and uncover a diverse range of exploitation flows, with 10 used exploitation techniques. In a subsequent analysis, we identify 8 defense-in-depth mechanisms present in the mainline Android kernel v6.1, mitigating most exploitation techniques and, hence, 84.6 % of exploitation flows. This percentage serves as the ground truth for the maximum achievable security of mobile devices.

In the *Defense Inclusion and Effectiveness Analysis* (see Section 5), we collect Android kernels released by all top mobile phone vendors (i.e., Samsung, Xiaomi, Oppo, Vivo, Realme, Huawei, Motorola, Google, OnePlus, and Fairphone) between 2018 and 2023. Our goal is to determine the inclusion and effectiveness of defense mechanisms in protecting these mobile devices. For this, we collect 994 device firmwares and

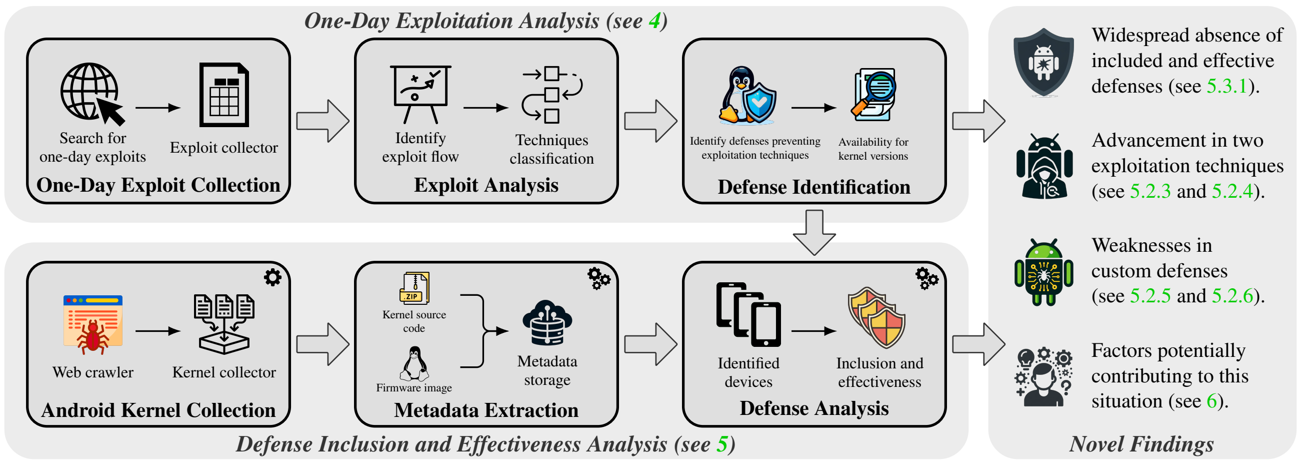

One-Day Exploitation Analysis (see 4)
Search for one-day exploits
Exploit collector
One-Day Exploit Collection
Identify exploit flow
Techniques classification
Exploit Analysis
Identify defenses preventing exploitation techniques
Availability for kernel versions
Defense Identification
Web crawler
Kernel collector
Android Kernel Collection
Kernel source code
Firmware image
Metadata storage
Metadata Extraction
Identified devices
Inclusion and effectiveness
Defense Analysis
Defense Inclusion and Effectiveness Analysis (see 5)
Widespread absence of included and effective defenses (see 5.3.1).
Advancement in two exploitation techniques (see 5.2.3 and 5.2.4).
Weaknesses in custom defenses (see 5.2.5 and 5.2.6).
Factors potentially contributing to this situation (see 6).
Novel Findings

Figure 2: The high-level workflow of our study where ⚙⚙ indicates fully automated and ⚙ indicates mostly automated.

1533 kernel source codes. Our analysis reveals that a significant portion of the analyzed device firmwares lacks multiple defenses, and some of the defenses are flawed, leaving devices vulnerable to multiple of the one-day exploitation flows analyzed in our one-day exploitation analysis.

Our analysis reveals four *Novel Findings*. First, we reveal the widespread absence (see Section 5.3.1) of included and effective defenses against one-day exploitation flows across vendors. Second, we demonstrate advancements in two exploitation techniques (see Sections 5.2.3 and 5.2.4). While these advancements enable bypassing the defense intended against the base techniques, we discuss potential solutions. Third, we uncover 4 and 2 weaknesses (see Sections 5.2.5 and 5.2.6) in Samsung's and Huawei's custom defense, respectively. Lastly, we discuss (see Section 6) factors potentially contributing to the absence of effective defenses and offer improvement recommendations.

4 One-Day Exploitation Analysis

In this section, we elaborate on our systematic analysis of all publicly available one-day exploits targeting memory safety vulnerabilities in the Android Linux kernel over the past three years. We identify and examine 26 exploits, demonstrating that their exploitation flow uses one or more of the 10 exploitation techniques outlined in Section 4.1. These techniques follow a generic strategy for transforming an exploit primitive into a more powerful one. In Section 4.2, we demonstrate that defense-in-depth mechanisms present in the Android kernel v6.1 can mitigate 22 (i.e., 84.6 %) one-day exploitation flows. Lastly, Section 4.3 demonstrates that the remaining 4 one-day exploits either exploit substantially powerful vulnerabilities or can be mitigated by a defense currently in development [44].

One-Day Exploits. We obtained 26 one-day exploits (see Table 1) from public sources, e.g., Google Project Zero [49], Blackhat [35], Github [41], or other websites [58]. Our selection consists of one-days exploiting memory safety vulnerabilities, as the Android kernel has established defenses to prevent their exploitation. By including other vulnerabilities, such as logical (e.g., CVE-2022-22706) and GPU (e.g., CVE-2023-33107) flaws, we expect that the susceptibility to one-day exploits increases as the mainline Android kernel does not yet effectively mitigate them. Our study spans the last 3 years, from 2020 to November 2023. This aligns with Google Project Zero's efforts to track zero-day exploits targeting Android devices. Earlier public exploits are less documented, so we focus on this more recent timeframe [48].

4.1 Identified Exploitation Techniques

We observe that most one-day exploits have distinct exploitation flows to convert one or more memory safety vulnerabilities into either an arbitrary read-and-write primitive or code modification (see Table 1). By examining these exploitation flows, we identify 10 exploitation techniques.

We refer to an exploitation technique as a strategy for turning one exploitation primitive into a more powerful one, with examples of primitives being n-byte OOB write, UAF write, program counter control, or arbitrary read and write. We classify exploitation techniques based on strategies that recur over multiple one-days and are reasonably generic [8]. An example of a technique is control-flow hijacking, which turns program counter control into code execution and is used by multiple one-days. Another example is the unlink operation, which may turn an OOB or UAF write of a double-linked list into a once-triggerable write or read primitive.

ET1: Unlink Operation. By exploiting a vulnerability, an adversary ensures that a victim object resides in the same memory slot as a double-linked list, i.e., `list_head` with `next` and `prev` (see Listing 1). The adversary then initiates the unlinking via `list_del`, resulting in a write to the victim object. The one-days CVE-2019-2215, CVE-2019-2025, and

Table 1: Exploitation flow used by publicly available one-day exploits, where defense-in-depth mechanisms present in the Android Linux kernel v6.1 can ✓ or cannot ✗ prevent the exploitation flow. The exploitation flow is preventable depending on ✶ Samsung's RKP [15] variant. Two one-day exploits ☆ exploit the same CVE with different exploitation flows.

CVE	Vulnerabilities	Exploitation flow	Goal Primitive	Preventable
CVE-2019-2215	UAF	in-cache reuse → unlink operation → KASLR leak, in-cache reuse → unlink operation → `addr_limit` overwrite	arbitrary r/w	✓
CVE-2019-2025 ☆	UAF	in-cache reuse → unlink operation → KASLR leak, in-cache reuse → unlink operation → `file` overwrite	arbitrary r/w	✓
CVE-2020-0030	UAF	in-cache reuse → unlink operation → KASLR leak, in-cache reuse → unlink operation → `addr_limit` overwrite	arbitrary r/w	✓
CVE-2021-1968,-1969,-1940	UAF	leak attacker-controlled data location, KASLR leak, in-cache reuse → CFH → ret2bpf	arbitrary r/w	✓
CVE-2021-0920	UAF	in-cache reuse → unlink operation → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2021-1905	UAF	cross-cache reuse → tamper allocator meta-data → KASLR leak → CFH → ret2bpf	arbitrary r/w	✓
CVE-2022-22265	DF	in-cache reuse → KASLR leak, cross-cache reuse → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2021-25369,-25370	Leak, UAF	KASLR leak, in-cache reuse → `file` overwrite → CFH → `addr_limit` overwrite	arbitrary r/w	✓
CVE-2016-3809,-2021-0399	Leak, UAF	KASLR leak, in-cache reuse → `seq_file` overwrite → CFH → ret2bpf	arbitrary r/w	✓
CVE-2022-20409	UAF	in-cache reuse → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2023-21400	DF	cross-cache reuse → Dirty PageTable	arbitrary r/w	✓
CVE-2022-28350	UAF	cross-cache reuse → Dirty PageTable	arbitrary r/w	✗/✶
CVE-2020-29661	UAF	cross-cache reuse → Dirty PageTable	arbitrary r/w	✗/✶
CVE-2021-22600	DF	in-cache reuse → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2020-0423	UAF	in-cache reuse → KASLR leak → unlink operation → KSMA	code modification	✓
CVE-2022-22057	UAF	in-cache reuse → KASLR leak → slab freelist corruption → KSMA	code modification	✓
CVE-2023-26083,-0266	Leak, UAF	KASLR leak, in-cache reuse → `ctl_file` overwrite → CFH	arbitrary r/w	✗
CVE-2020-0041	UAF	in-cache reuse → KASLR leak, in-cache reuse → unlink operation → tamper `sysctl`	arbitrary r/w	✓
CVE-2019-2205	UAF	in-cache reuse → KASLR leak, in-cache reuse → unlink operation → tamper `binder_proc`	arbitrary r/w	✓
CVE-2019-2025 ☆	UAF	in-cache reuse → KASLR leak, in-cache reuse → unlink operation → KSMA	code modification	✓
CVE-2020-3680	UAF	in-cache reuse → KASLR leak → unlink operation → KSMA	code modification	✓
CVE-2022-20421	UAF	cross-cache overflow → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2022-0847	Uninit Variable	uninitialized `pipe` → DirtyPipe → overwrite the cached file page	arbitrary r/w	✓
CVE-2021-4154	UAF	in-cache reuse → DirtyCred → overwrite shared library	arbitrary r/w	✗
CVE-2021-38001	OOB R/W	OOB write → stack manipulation → KASLR leak → OOB write → stack manipulation → CFH → ret2bpf	arbitrary r/w	✓
NO_NUMBER (∼2021)	OOB W	OOB write → slab freelist corruption → `pipe_buffer` DF → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
				22/26

```
1 struct list_head {
2   struct list_head *next;
3   struct list_head *prev;
4 };
5 /* Unlinks element e */
6 void list_del(list_head *e) {
7   e->next->prev = e->prev;
8   e->prev->next = e->next;
9 }
```

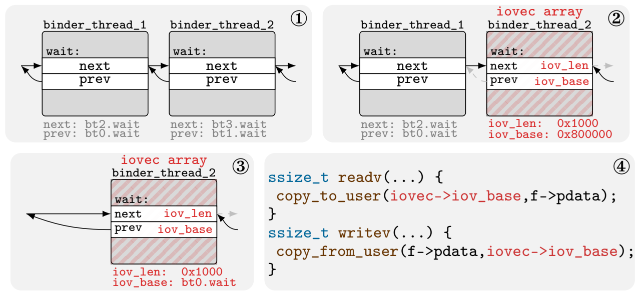
Listing 1: Unlinking operation.

```
struct binder_thread {
  struct list_head wait;
  struct task_struct *task;
};
void remove_wait_queue(
      binder_thread *bt) {
  /* Trigger unlinking */
  list_del(&bt->wait);
}
```

Listing 2: Trigger unlinking.

```
u64 access_ok(const void __user *addr, u64 size) {
  return (u64)((u65)addr + (u65)size <= (u65)current->
        addr_limit + 1);
}
u64 copy_from_user(void *to, const void __user *from, u64 n) {
  u64 res = n;
  if (access_ok(from, n))
    res = raw_copy_from_user(to, from, n);
  return res;
}
```

Listing 3: Userspace data copy function validates with `access_ok` whether `addr` refers to userspace memory.



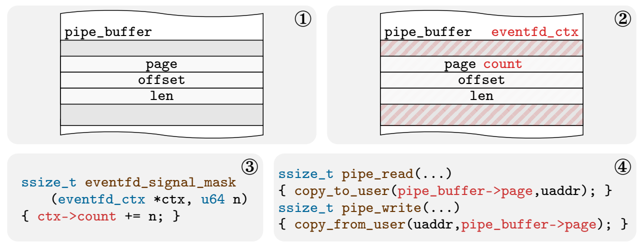
```
ssize_t readv(...) {
 copy_to_user(iovec->iov_base,f->pdata);
}
ssize_t writev(...) {
 copy_from_user(f->pdata,iovec->iov_base);
}
```

Figure 3: Exploitation example of the unlink operation.

CVE-2020-0030, for example, leverage this unlink operation to first leak `binder_thread->task`, whose layout is shown in Listing 2, and then overwrite `task->addr_limit` (**ET2**).

Figure 3 illustrates an exploitation example [46], where initially, an adversary prepares a double-linked list ①. They then exploit a vulnerability to ensure that the second `wait` entry (`binder_thread_2.wait` or short `bt2.wait`) resides in the same memory slot ② as an `iovec` object. This `iovec` stores a user buffer, with `iov_base/len` being the user buffer's pointer/size, commonly used for file reading or writing. Executing `remove_wait_queue` on the first `wait` entry (`bt1.wait`) overwrites the `iov_base` of the second `wait` entry with `bt1.wait->prev` ③ (at Line 7 of `list_del`). Consequently, the buffer `iovec` now points to `binder_thread_0.wait` (short `bt0.wait`). These exploits then use the `iovec` read-/write functionality ④ (e.g., `readv` or `writev`) to write to or read from the `iovec->iov_base` and, hence, `bt0.wait`. This approach is used to leak `binder_thread->task` and overwrite `task->addr_limit`. While this example shows the usage of `iovec` (which has been fixed for v4.13 [2]), other security-critical objects can also be misused, e.g., `msg_msg` [42] or `pipe_buffer` in CVE-2021-0920.

ET2: `addr_limit` Overwrite. This technique turns a `task->addr_limit` overwrite into an arbitrary read and write. AArch64 kernels below v5.11 include `addr_limit` in `task`, which holds the highest address accessible within user-data copy functions, e.g., `copy_*_user`. These functions call `access_ok` to validate that the user address is lower than



```
ssize_t eventfd_signal_mask
    (eventfd_ctx *ctx, u64 n)
{ ctx->count += n; }
```

```
ssize_t pipe_read(...)
{ copy_to_user(pipe_buffer->page,uaddr); }
ssize_t pipe_write(...)
{ copy_from_user(uaddr,pipe_buffer->page); }
```

Figure 4: Exploiting `pipe_buffer` to obtain an arbitrary r/w.

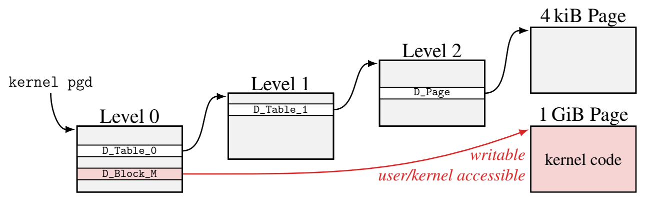


Figure 5: **KSMA:** Due to a write capability to page table level 0, an adversary maliciously overwrites the `D_Block_M` entry to refer to kernel code as writable and user accessible.

`addr_limit` (see Line 6 of Listing 3), aiming to ensure user address access. However, by overwriting `addr_limit` with `KERNEL_DS` (i.e., -1), an adversary can deceive the kernel into legally accessing kernel memory within these copy functions. Hence, syscalls (e.g., `read` and `write`) using these copy functions can be misused as an arbitrary read-and-write primitive.

ET3: `pipe_buffer` Overwrite. Overwriting the `pipe_buffer` yields an arbitrary read and write as follows. Initially, an adversary requires an arbitrarily triggerable overwrite capability for a `pipe_buffer` object that is still in use. One approach is to exploit a UAF vulnerability so that a `pipe_buffer` and a specific object (e.g., `eventfd_ctx` or `signalfd_ctx`) reside in the same memory slot. Since this specific object is writable from userspace, it enables manipulating `pipe_buffer` (e.g., `eventfd_ctx` for CVE-2021-22600). Another approach enforces the coexistence of a `pipe_buffer` and the backed physical page of another `pipe_buffer` in the same slot (cf. CVE-2022-22265).

Figure 4 illustrates the exploitation of CVE-2021-22600. In ①, the memory layout of a `pipe_buffer` is shown with its members `page`, `offset`, and `len`. Step ② exploits the vulnerability where afterward `pipe_buffer` and `eventfd_ctx` reside in the same memory slot, and `page` and `count` coexist on the same address. Calling `eventfd_signal_mask` ③ allows to change `count` and, hence, `pipe_buffer->page`. Consequently, `pipe_read/write` ④ read from or write to this controlled address, granting an arbitrary read and write.

ET4: Control-Flow Hijacking. Various one-day exploits perform a Control-Flow Hijacking (CFH) attack, leveraging an overwrite capability of either a function pointer or a pointer to a function pointer. Compared to x86_64 exploitation, they do not resort to Return-Oriented Programming (ROP) [10]. Instead, they identify an execution path resulting in an arbitrary read-and-write primitive. For instance, CVE-2023-0266 overwrites the `f_ops` pointer (referencing a table of function pointers for file interactions) of `ctl_file`. As a result, the syscalls `read` and `write` confuse the `void *pdata` member of `ctl_file`, leading to a misuse of `copy_*_user` and yielding an arbitrary read-and-write primitive.

ET5: Ret2bpf. Ret2bpf serves as an alternative to ROP, offering a similarly potent capability [7, 28]. Its prerequisites [28] involve hijacking the control flow (**ET4**), partial control of the first argument register, control over the second argument register, and a controllable data region. In ret2bpf, a data region is crafted to contain valid eBPF [7] instructions, performing, for instance, an arbitrary read and write. With the CFH primitive and the crafted eBPF instructions, ret2bpf performs a CFH attack to execute `___bpf_prog_run(regs, inst)`. This function interprets the crafted eBPF instructions as if the eBPF verifier had validated them. Here, `inst` is the data region holding the crafted eBPF instructions, and `regs` represents a writable section used for registers.

ET6: Slab Freelist Corruption. This exploitation technique turns a once-only OOB or UAF write into a memory slot overwrite capability. It requires a memory slot that is currently in the freed state. By exploiting a write capability on this free slot (e.g., zeroing memory due to a UAF or OOB write), an adversary manipulates a freelist pointer stored within the free slot. Then, by allocating an object, the adversary illegally reclaims the memory slot referenced by the corrupted freelist pointer. This allocated object typically grants overwrite capabilities for the reclaimed memory slot.

ET7: KSMA. Yong et al. [63] introduced the Kernel-Space Mirroring Attack (KSMA), which transforms a once-triggerable write primitive into a kernel code manipulation capability. This transformation is done by manipulating a page table level 0, called Page Global Directory (PGD) (e.g., `swapper_pg_dir`), representing the kernel address space.

Specifically, KSMA forges an entry within the kernel's page table level 0, designating its address range as accessible from user and kernel space. This forged entry is marked as a 1 GB huge page and references kernel code. Consequently, the entire kernel code (including kernel data) is readable and writable from userspace. The page-table layout after performing KSMA is shown in Figure 5 with a 3-level page-table translation (i.e., 39 bit Virtual Address Size (`VA_SIZE`) and 4 kiB page size, but it works similarly for other configurations). This kernel code modification is then utilized to disable SELinux and manipulate a syscall to elevate the privileges.

ET8: Dirty PageTable. Dirty PageTable [58] shows how page-table tampering results in an arbitrary read and write on Android (where Maar et al. [38] show generic page-table manipulation). It exploits a UAF (cf. CVE-2022-28350 and CVE-2020-29661) or DF (cf. CVE-2023-21400) for a cross-cache attack [60]. This causes an object with arbitrary overwrite

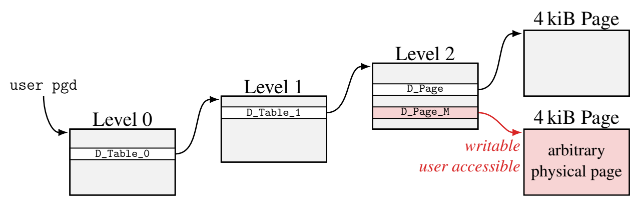


Figure 6: **Dirty PageTable:** With an arbitrary page table level 2 write capability, an attacker tampers the `D_Page_M` entry to refer to an arbitrary page as writable and user accessible.

capabilities (e.g., `signalfd_ctx` for CVE-2023-21400) to reside in the same memory slot as a page table used for user address translation. Figure 6 shows this, where an adversary has an arbitrary overwrite to the page-table entry `D_Page_M` due to the cross-cache attack. By triggering the overwrite, they gain control over the page frame number of this entry. Reading or writing to the user address using this page-table entry gives them arbitrary physical memory access.

ET9: DirtyPipe. The DirtyPipe attack [30] exploits an uninitialized variable to escalate privileges. The CVE-2022-0847 vulnerability allows to use the `pipe_buffer.flags` variable uninitialized. Consequently, this vulnerability allows overwriting of any file contents in the page cache, also in the case of read-only files, which results in privilege escalation.

ET10: DirtyCred. The DirtyCred exploit [34] allows an attacker to escalate privileges. It exploits a file UAF to free a writable file currently in use. Prior to this invalid free, it performs a write operation to the file and stalls this write operation. After the free, it reclaims the file object for a read-only high-privilege file. Continuing the stalled write operation now writes to the read-only high-privilege file. With this file manipulation, DirtyCred can, e.g., overwrite a kernel module with malicious code to construct an arbitrary read and write.

4.2 Defenses to Prevent Exploitation Flows

We identify 10 defense-in-depth mechanisms present in the Android kernel v6.1 or provided by vendors. They prevent exploitation techniques and, hence, exploitation flows, with the findings shown in Table 1 and detailed in Table 2.

DM1: `CONFIG_DEBUG_LIST`. This defense includes checks in `del_list` whether `e->next->prev == e` and `e->prev->next == e`. If these checks fail, the entry will not be unlinked. Thus, it mitigates the unlink operation (**ET1**). In Figure 3, for instance, overwriting from step ② to ③ is prevented as `iovec->iov_base` is not equal to `bt1.wait`.

DM2: `CONFIG_ARM64_UAO`. User-Access Override (UAO) [9] is a hardware-enforced defense that aims to mitigate `addr_limit` overwrite (**ET2**). It introduces new unprivileged load and store instructions that behave like privileged ones when the UAO bit is set. This restricts user-data copy functions, e.g., `copy_*_user`, from being misused to read from or write to kernel addresses directly.

Table 2: Mitigation of one-day exploits, using ✓ to indicate defenses that prevent used exploitation techniques. Conversely, ✗ indicates ineffective defenses (see Sections 5.2.3, 5.2.4 and 5.2.6). Samsungs's defenses ✗/✶ are either ineffective or only effective in certain variants (see Section 5.2.5).

CVE	☰	⇨	▦	⑂	</>	🔗	▪	🏛	📞	✹
CVE-2019-2215	✓	✗	✓							✓
CVE-2019-2025	✓		✓							
CVE-2020-0030	✓	✗	✓							✓
CVE-2021-1968,-1969,-1940				✓	✓				✗	
CVE-2021-0920	✓		✓							
CVE-2021-1905				✓	✓				✗	
CVE-2022-22265			✓							
CVE-2021-25369,-25370		✗	✓	✗					✗	✓
CVE-2016-3809,-2021-0399			✓	✓	✓				✗	
CVE-2022-20409			✓							
CVE-2023-21400			✓						✶	✗
CVE-2022-28350									✶	✗
CVE-2020-29661									✶	✗
CVE-2021-22600			✓							
CVE-2020-0423	✓							✗	✓	✓
CVE-2022-22057						✓		✗	✓	✓
CVE-2023-26083,-0266				✗					✗	
CVE-2020-0041	✓		✓							
CVE-2019-2205	✓		✓							
CVE-2019-2025	✓		✓					✗	✓	✓
CVE-2020-3680	✓		✓					✗	✓	✓
CVE-2022-20421			✓							
CVE-2022-0847							✓			
CVE-2021-4154										
CVE-2021-38001				✓	✓				✗	
NO_NUMBER (∼2021)			✓			✓				

☰ **DM1:** `CONFIG_DEBUG_LIST` ⇨ **DM2:** `CONFIG_ARM64_UAO` ▦ **DM3:** `kmalloc-cg-*`
⑂ **DM4:** `CONFIG_CFI_CLANG` </> **DM5:** `CONFIG_BPF_JIT_ALWAYS_ON`
🔗 **DM6:** `CONFIG_SLAB_FREELIST_HARDENED` ▪ **DM7:** `CONFIG_INIT_ON_ALLOC_DEFAULT_ON`
🏛 **DM8:** KSMA protection 📞 **DM9:** Samsung RKP ✹ **DM10:** Huawei HKIP

DM3: `kmalloc-cg-*`. Linux kernels above v5.13 support heap segregation at the allocator cache level. It separates caches to provide a designated cache for security-critical data marked as accounted, such as for `msg_msg`, `pipe_buffer`, `file`, and `task_struct`. For generic caches, a cache for non-security-critical data (i.e., `kmalloc-*`) and a cache for security-critical data (i.e., `kmalloc-cg-*`) are created. Free and available cached objects will never share the same memory slots within these caches. Hence, this mitigates the `pipe_buffer` overwrite (**ET3**) and unlink operation (**ET1** with security-critical objects), as these techniques rely on security-critical and non-security-critical data sharing the same memory slot. While adversaries might consider cross-cache attacks, three challenges arise with this approach, making the transition infeasible. First, for generic caches, the success rate significantly decreases to 40 % [60], with failure scenarios potentially resulting in a crash. The small success rate makes this approach impractical since the cross-cache reuse only pertains to a small part of the exploit and may need multiple repetitions. Second, exploits that engage in cross-cache attacks typically rely on prior in-cache reuse attacks to stabilize the exploit. For instance, CVE-2022-22265 stabilizes by in-cache reallocating the double-freed slot of the `pipe_buffer` as an `iovec` multiple times. Separating the `pipe_buffer` from objects intended for stabilizing, such as `iovec`, makes the exploit unstable, mostly resulting in a crash, successfully preventing exploitation. A similar applies to CVE-2023-21400, where `seq_operations` (accounted) are prevented from being in-cache reallocated as `signalfd_ctx` (not accounted). Third, various UAF exploits (e.g., CVE-2021-0399) offer a tight time window in which an in-cache attack is exploitable. In contrast, cross-cache attacks require more time due to the recycling/reclaiming of the slab page to/from the page allocator [60], making small windows not exploitable.

DM4: `CONFIG_CFI_CLANG`. Control-Flow Integrity (CFI) [1,3] restricts the control flow to an approximate Control-Flow Graph (CFG), limiting the targets for CFH attacks (**ET4**). The Android kernel uses Clang's implementation [3], providing function-signature-grained CFI. It prevents CFH attacks that overwrite function pointers with arbitrary functions, e.g., CVE-2021-1905 and CVE-2021-0399.

DM5: `CONFIG_BPF_JIT_ALWAYS_ON`. To mitigate ret2bpf (**ET5**), this defense mechanism forces BPF to always use the JIT engine instead of the interpreter. Consequently, the `___bpf_prog_run` function used by ret2bpf is not compiled and, therefore, cannot be called, preventing ret2bpf.

DM6: `CONFIG_SLAB_FREELIST_HARDENED`. To mitigate the manipulation of slab allocator metadata, this defense hardens the slab allocator by adding sanity checks. This includes XORing the freelist pointer with a pseudo-random number, preventing slab freelist corruption (**ET6**).

DM7: `CONFIG_INIT_ON_ALLOC_DEFAULT_ON`. It zeroes out the memory slot for both allocations by the page and slab allocator. Consequently, it greatly minimizes the exploitability of uninitialized values. It prevents the exploitation of DirtyPipe (**ET7**) as the uninitialized `pipe_buffer.flags` cannot be misused to overwrite file content in the page cache.

DM8: KSMA Protection. In response to KSMA (**ET8**), researchers proposed to move all kernel level 0 global page tables (e.g., `swapper_pg_dir` and `tramp_pg_dir`) to a read-only section [62]. As a result, these page tables cannot be manipulated for a huge kernel memory mapping (e.g., 1 GiB) that is writable from userspace, thus preventing KSMA.

DM9: Samsung RKP. Samsung's Real-time Kernel Protection (RKP) [15] employs hypervisor-based protection designed to mitigate code modification, data modification, and control-flow hijacking in the kernel. To address kernel code modification, RKP ensures the integrity of page tables (**ET8** and **ET9**) and code by mapping them as read-only, protected by the hypervisor. Hypervisor calls permit legitimate writes to these protected pages. RKP also limits CFH attacks (**ET4**) by including checks before indirect branches that restrict control-flow transfers to a function-grained CFG.

DM10: Huawei HKIP. Huawei Kernel Integrity Protection (HKIP) [25] employs hypervisor-based protection that protects kernel code and critical kernel data. It also limits privilege escalation and protects additional control-flow-related data. To achieve this, HKIP ensures the integrity of certain page tables (**ET8** and **ET9**), `addr_limit` (**ET2**), CFI metadata, and eBPF interpreted code by protecting them via the hypervisor. Hypervisor calls or exceptions to the hypervisor permit legitimate writes to these protected pages. Protecting CFI metadata only provides additional protection for modules and not against the CFH technique. Similarly, protecting the eBPF-interpreted code does not prevent against ret2bpf, as it only safeguards the already interpreted instructions.

Further defenses. The ongoing research in improving kernel security yielded results with various kernel defenses. For instance, `CONFIG_HARDENED_USERCOPY` restricts `copy_*_user` from reading and writing out of bounds [59]. Other examples include `CONFIG_INIT_STACK_ALL_ZERO` mitigating uninitialized stack variable exploitation [37] and `CONFIG_STACKPROTECTOR_STRONG` providing stack protection [45]. While these defenses cover a broad range of vulnerability mitigation, our focus is specifically on defenses against one-day exploitation flows on the Android kernel (**DM1-10**).

4.3 Unpreventable Exploitation Flows

We identify 4 one-day exploitation flows targeting the Android kernel that remains unpreventable by mainline defenses. Among these, DirtyCred (cf. CVE-2021-4154) presents a powerful technique that falls beyond the defense prevention scope. A similar applies to the CFH one-day (cf. CVE-2023-26083,-0266), redirecting the control flow to perform an arbitrary read and write without violating signature-based CFI.

While our identified defenses do not effectively prevent two other one-days (Dirty PageTable, cf. CVE-2022-28350 and CVE-2020-29661), researchers are actively developing a new defense mechanism specifically designed to counter cross-cache reuse attacks [44]. This mitigation strategy involves switching the allocation of memory slots cached by the slab allocator from physical to virtual pages, thereby preventing the reuse of slab pages returned by the page allocator.

5 Defense Inclusion & Effectiveness Analysis

In this section, we outline our systematic analysis demonstrating a *widespread deficiency of included defense mechanisms across vendors* as well as *shortcomings of certain defenses*. Our approach (see Figure 2) consists of three mostly automated stages: Initially, we collect kernel source codes and firmwares (see Section 5.1) for Android devices from 10 vendors. We then analyze kernel codes (see Section 5.2) to assess the effectiveness of defenses provided by the mainline kernel or vendors. Lastly, we analyze firmwares (see Section 5.3) to detect included effective defenses in devices.

Android Devices. For our analysis (done in November 2023), we cover Android devices from vendors that constitute more than 84 % [6] of the global market. These include the top 7 vendors [6], i.e., Samsung, Xiaomi, Oppo, Vivo, Huawei, Realme, and Motorola, along with Google, OnePlus, and Fairphone. We assess devices released between 2018 and 2023, utilizing Android versions 9 through 14 and kernels ranging from v3.10 to v6.1. These Android versions account for a share of more than 86 % [5], with specific percentages for Android 13, 12, 11, 10 and 9, being 25.7 %, 21.3 %, 17.3 %, 13.9 %, and 8.7 %, respectively. Android 14, while at a negligible market share at the moment, is also considered.

We decided to start with phones released in November 2018 (5 years from the start of this work), as the lifespan of Android phones is 4-6 years (4y for Huawei, 5y for Google, and 6y for Samsung) [17, 54]. Hence, our selection ensures a comprehensive overview of the current device landscape.

5.1 Collection of Firmwares and Kernel Codes

This step automatically collects firmwares (not protected by captchas) and kernel code. To achieve this, we implement a Python script using Selenium that crawls web pages to collect firmwares and kernel source code from our 10 vendors. We manually collected firmware protected by captchas or other automation detections (i.e., $\approx 45.3\,\%$).

Firmwares. Our 10 vendors produced 1698 devices between November 2018 and November 2023 (see Table 4). For 1109 of them, firmwares were provided, where we only considered the most recent release either officially (e.g., Google) or via an intermediate supplier (e.g., Oppo).

Kernel Codes. We collected 1533 kernel codes (see Table 4) with different releases for the same device (e.g., Samsung and Huawei) where available. Other vendors (e.g., Google and Vivo) use the same kernel code for multiple devices, resulting in less collected code than firmwares.

5.2 Analysis of Kernel Source Codes

We examine kernel source codes for efficacy against exploitation techniques. Initially, we provide evidence that our identified defenses (see Sections 5.2.1 and 5.2.2) reflect the real world of mitigating exploitation techniques. However, we also identify shortcomings in these defenses. We show that they can be bypassed, indicating that their efficacy is less than intended due to advanced techniques (see Sections 5.2.3 and 5.2.4) or weaknesses (see Sections 5.2.5 and 5.2.6).

5.2.1 Mainline Defenses in Downstreamed Kernels

7 of the 8 mainline defense mechanisms are intrinsically tied to the core functionality of the Android kernel:

- Associating specific defenses with versions (i.e., **DM3**).
- Not compiling dangerous functions (i.e., **DM5**).
- Replacing non-hardened with hardened functions (i.e., **DM1/4/6/7/8**).

For example, `CONFIG_DEBUG_LIST` (**DM1**) uses the hardened function `__list_add_valid` to validate metadata in double-linked lists. Another example is `kmalloc-cg-*` (**DM3**), which utilizes a segregated set of allocator caches for kernel versions 5.13 and above. The exception is

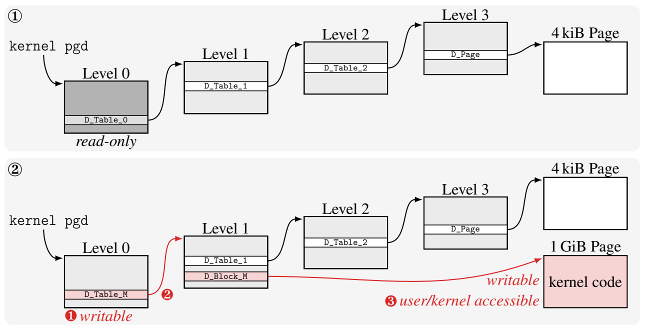


Figure 7: **Advanced KSMA:** ① Initial 4-level page table translation with the level 0 table mapped as read-only. ② Modifies the level 0 mapping to mark it as writable ❶, overwrites `D_Table_M` ❷, and appends `D_Block_M` ❸, to have a 1 GiB mapping accessible from userspace.

`CONFIG_ARM64_UAO` (**DM2**), which is hardware-dependent. Although our analysis might identify this defense as present, this defense can be bypassed, as we demonstrate in Section 5.2.4. Hence, regardless of whether it is enabled, it does not protect `addr_limit` overwrite (**ET2**).

5.2.2 Identified Downstream Defenses

Some vendors provide custom defenses to improve kernel security. We semi-automatically analyze the 1533 collected kernels and provide evidence of 3 vendor-specific downstream defenses against the identified exploitation techniques. The analysis works as follows: First, we automatically collect the configuration flags in the `./security` subdirectory and in the files that require changes to mitigate exploitation techniques, e.g., `vmlinux.ld.S` and `mmu.c` to prevent KSMA (**ET8**). Second, we manually analyze those flags collected from downstream kernels that are not present in the mainline kernel (i.e., Google). Our analysis results in Samsung RKP (**DM9**), Huawei HKIP (**DM10**), and `CONFIG_PG_DIR_RO` from Vivo (which we consider in the firmware analysis as **DM8**). We also received confirmation from Fairphone and Motorola that they do not include vendor-specific defenses.

5.2.3 Advanced Kernel-Space Mirroring Attack

Despite the KSMA mitigation patch, we present an advancement in reenabling KSMA. Its prerequisite is the same constraint write capability as the base KSMA (**ET8**), but it uses it twice: First, it changes the mapping of the level 0 PGD to writable, and second, it maliciously inserts the page-table entry into the PGD. For a 48 bit `VA_SIZE` system with 4-level translation, our technique triggers the write three times, as depicted before ① and after ② the attack in Figure 7. The first write changes the PGD mapping to writable ❶, while the second changes the PGD entry to user accessible ❷. The third write inserts then the entry ❸ in the page-table level 1.

For this technique, the locations of three page-table pages (*level 0*, *level 1*, and *level 3'* corresponding to the mapping *level 0* as read-only) are crucial. Depending on whether `CONFIG_UNMAP_KERNEL_AT_EL0` is active, `swapper_pg_dir` or `tramp_pg_dir` is used as a *level 0* page. A KASLR code leak, combined with knowledge of the kernel binary under attack, is sufficient to obtain these locations since both locations are mapped to a fixed offset to the kernel base address. This step is also needed for the base KSMA.

Both *level 1* and *level 3'* are allocated via the page allocator during the early initialization stages and accessed via the Direct-Physical Mapping (DPM) [39], which is a virtual memory mapping to the entire physical memory. Since the page allocator returns the same physical page during different boots, their locations can be determined. The DPM may be randomized on newer Android kernels (e.g., v6.1). To overcome randomization, a heap address leak is typically sufficient to derandomize the DPM, as the kernel heap uses the DPM directly. Typically, leaking a heap address requires no additional effort beyond leaking the kernel base address.

Armed with the three page-table locations, our advanced KSMA uses the write capability three times to obtain kernel code modification, which no mainline defense can prevent. The level of difficulty of our advancement is similar to the base KSMA, but the write capability is triggered three times, and for recent Android versions, a heap leak is required.

Experiments. Our setup involves a buildroot filesystem with an Android kernel (aarch64 with a 48 bit `VA_SIZE`), specifically v5.15 and v6.1. We run it inside a virtual machine with 4 cores and 4 GB RAM via QEMU 6.2.0. We introduce a write primitive and run our exploits as an unprivileged user, giving them the same capabilities as the base KSMA version. As a result, we successfully execute our advanced KSMA and obtained an arbitrary code modification primitive.

Mitigation. Our advanced KSMA requires the locations of all mapping page tables that are not randomized during the early initialization process. Hence, an adversary can still deduce their location by knowing the kernel binary under attack (and a heap leak for v6.1). To counteract the advanced KSMA, we propose randomizing the locations of the page tables during this early initialization stage, ensuring that adversaries cannot obtain information about the page's location.

5.2.4 Shortcoming of User Access Override

The UAO feature is believed to prevent the `addr_limit` overwrite (**ET2**) [36] effectively. This technique manipulates `addr_limit` with `KERNEL_DS` to facilitate, for example, pipes for arbitrary kernel reads and writes. Specifically, it first writes the pointer to a userspace buffer to one pipe end. It then performs a `read` syscall with a kernel address as an argument, prompting the userspace copy function to write the userspace buffer's content to this kernel address. With UAO enabled, setting `addr_limit` to `KERNEL_DS` prevents the first write operation. Moreover, setting `addr_limit` to `USER_DS` prevents the second write to kernel memory.

However, since `addr_limit` operates at thread granularity, we spawn two threads, *T1* and *T2*, where we only illegally overwrite the `addr_limit` of *T2* with `KERNEL_DS`. We leverage *T1* to perform the first write and *T2* for the second write. As a result, we can bypass UAO without further restrictions. Prior work [9] has presented similar bypasses.

Mitigation. A mitigation would be to remove the `addr_limit` functionality or use kernels above v5.11, which do not support `addr_limit` anymore.

5.2.5 Samsung RKP Weaknesses

We inspect Samsung RKP [15], designed to prevent page-table manipulation and limit CFH attacks. However, we demonstrate that various RKP variants only protect certain page tables and, thus, do not mitigate page-table manipulation. They also provide less CFH protection than the mainline defense.

Analysis. For each of these findings, we provide statistical data on their occurrence, collected using the following approach. We first perform automated source code analysis, followed by manual verification. We then conduct experiments to demonstrate the severity of these identified problems.

Findings. First, some kernels have RKP disabled and do not map `tramp/swapper_pg_dir` or `tramp_pg_dir` as read-only. Compared to the mainline defense, this results in less security as an adversary can directly perform KSMA. We found this weakness mostly in low-end devices such as Galaxy A04/A14 (i.e., released 2022/2023), missing both pages and Galaxy M10 (i.e., released 2019), missing `tramp_pg_dir`, representing 25.4 % and 1.7 % of kernels, respectively.

Second, while some variants protect page tables used for userspace address translation, we observe a strong tendency to exclude this protection towards new high-end devices such as Galaxy S23 5G. Specifically, we observe that less than 53 % of devices include this protection, indicating that more than 47 % are vulnerable to Dirty PageTable [58].

Third, `CONFIG_FASTUH_RKP` is a performance-optimized RKP variant included in over 60 % of all v5.4 kernels, providing a maximum number of read-only pages protected by the hypervisor. If the system demands more, RKP resorts to allocating unprotected pages. An adversary can exhaust these read-only protected pages and, subsequently, perform Dirty PageTable. This performance-optimized RKP variant is available for lower-end devices, e.g., Samsung Galaxy J6, and for high-end devices, e.g., Samsung Galaxy S20 FE and S21+ 5G. Similarly, `CONFIG_TIMA_RKP` provides similar weak protection for page tables, mainly used by older devices.

Fourth, `CONFIG_RKP_CFP_JOPP/_ROPP` aim to mitigate CFH attacks [15] by providing function-granular CFI (`JOPP`) and return address protection (`ROPP`). However, our analysis of exploitation flows reveals that all 6 CFH attacks redirect the control flow with at least function granularity. Hence, both defenses are ineffective in mitigating any of the CFH attacks.

Experiments. For the first weakness, we use the same setup as for our advanced KSMA technique and overwrite unprotected page-table pages to perform KSMA. We then implement POCs for the other weaknesses on a Samsung Galaxy S20 FE, where we modify the kernel code to obtain the corresponding primitive. For the second weakness, we use the introduced write primitive for a page table used for a userspace address translation to successfully perform Dirty PageTable. For the third weakness, we demonstrate that we can drain the protected pages with memory exhaustion. We then prompt the kernel to allocate a page that should be protected, but this is not due to memory exhaustion. For the fourth weakness, we demonstrate that RKP does not mitigate control-flow hijacking to arbitrary functions. As a result, control-flow protection does not prevent the 6 CFH attacks we analyzed.

5.2.6 Huawei HKIP Weaknesses

We examine Huawei's HKIP [25], particularly regarding the protection against KSMA and Dirty PageTable.

Analysis. We observe that HKIP is only included in certain devices and enabled in about 62 %. In the following, we analyze HKIP and experimentally demonstrate the absence of protection for crucial page-table pages.

Findings. First, HKIP protects page-table pages that are allocated for kernel address translations (e.g., via `pte_alloc_one`) in a specific virtual address range. As a result, HKIP does not protect page tables for userspace address translations, leaving devices vulnerable to Dirty PageTable.

Second, while HKIP protects the ttbr (hardware register that stores the current PGD for address translation) switch, it may not be compatible with frequent ttbr switching defenses, i.e., software PAN (`CONFIG_ARM64_SW_TTBR0_PAN`) switches the ttbr for each `copy_*_user` and Meltdown protection (`CONFIG_UNMAP_KERNEL_AT_EL0`) for each user kernel switch. No device has HKIP with either one of these two enabled, leaving these devices vulnerable to KSMA.

Experiments. We could not run experiments with the Huawei kernel source codes as they either had compilation errors, no `defconfig` (e.g., `ranchu64`) viable for virtual environments or failed to boot in QEMU. Therefore, we adapted a Google kernel v4.14 to tag pages that HKIP would have protected. For our page-table manipulation attacks, we experimentally observed that HKIP does not protect page-table pages that KSMA and Dirty PageTable manipulate.

5.3 Analysis of Firmwares

This work refers to the firmware as the stock ROM, the original software loaded onto the device by the vendor. It consists of multiple images [4], such as the system and boot image. Figure 8 shows the automated workflow of our implemented Python script, extracting the necessary metadata for defense

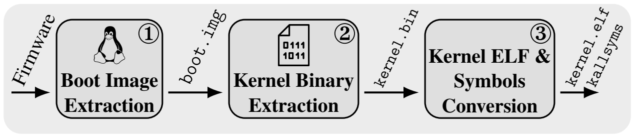


Figure 8: Workflow of extracting `kernel.elf` and `kallsyms` from the firmware, required for the defense detection.

detection. It first extracts the boot image ① using open-source tools, which requires different tools [22, 31, 32, 51, 55] as vendors encode the boot image differently. It then extracts the kernel binary ② using `unpack_bootimg` [35]. Lastly, it uses `kallsyms_finder` and `vmlinux_to_elf` to reconstruct the symbols (i.e., `kallsyms`) and convert the kernel binary to an analyzable ELF (i.e., `kernel.elf`) ③ [40].

The `kallsyms` and `kernel.elf` components form the basis of defense detection. Our Python script uses `kallsyms` to identify global functions within the kernel binary, allowing us to deduce the active defense mechanisms. The presence of `__list_add_valid` in `kallsyms`, for instance, indicates the status of `CONFIG_DEBUG_LIST` (**DM1**). Our script does similar assessments for other defenses (see Table 3). It uses the `kernel.elf` to determine the status of KSMA protection (**DM8**) and `CONFIG_SLAB_FREELIST_HARDENED` (**DM6**). For KSMA protection, all PGDs (e.g., `swapper_pg_dir`) must be mapped in a read-only section. The presence of calling `get_random_long` within `__kmem_cache_create` indicates the status of `CONFIG_SLAB_FREELIST_HARDENED`.

Our evaluation also includes five features for system security; KASLR (`CONFIG_RANDOMIZE_BASE`), code write protection (`CONFIG_STRICT_KERNEL_RWX`), freelist randomization (`CONFIG_SLAB_FREELIST_RANDOM`), restricting user access in kernel (`CONFIG_ARM64_(SW_TTBR0_)PAN`), and Meltdown protection (`CONFIG_UNMAP_KERNEL_AT_EL0`).

Evaluated Firmwares. Out of the 1698 released and 1109 collected devices, our analysis extracted 994 firmwares, resulting in a collection rate of 58.5 %, which aligns with prior work on reverse engineering firmwares [11, 14, 64].

Due to the unavailability of certain firmwares, our analysis could not cover all released devices. However, we observed that the missing firmwares are distributed either normally regarding device age, such as those from Huawei and Vivo, or tailored to older devices, as seen with Xiaomi and Realme. Given our finding that older devices tend to include fewer defenses, our analysis provides conservative results. Thus, we anticipate the real-world scenario to be even more concerning.

5.3.1 Analysis Results

We fully automate the detection of included defenses. Table 5 presents the defenses included for each vendor's firmwares. Our results indicate a lack of basic defenses (e.g., PAN and KASLR) and a significant lack of defenses against one-day ex-

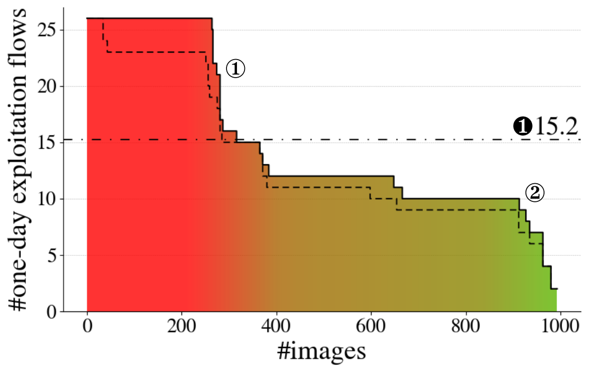


Figure 9: Susceptible one-day exploitation flows of all device images. While ① indicates that 281 images are susceptible to 21 or more one-day exploitation flows and ② indicates 913 images to 10 or more, the ❶ line represents the average susceptibility of 15.2 exploitation flows of all 994 images.

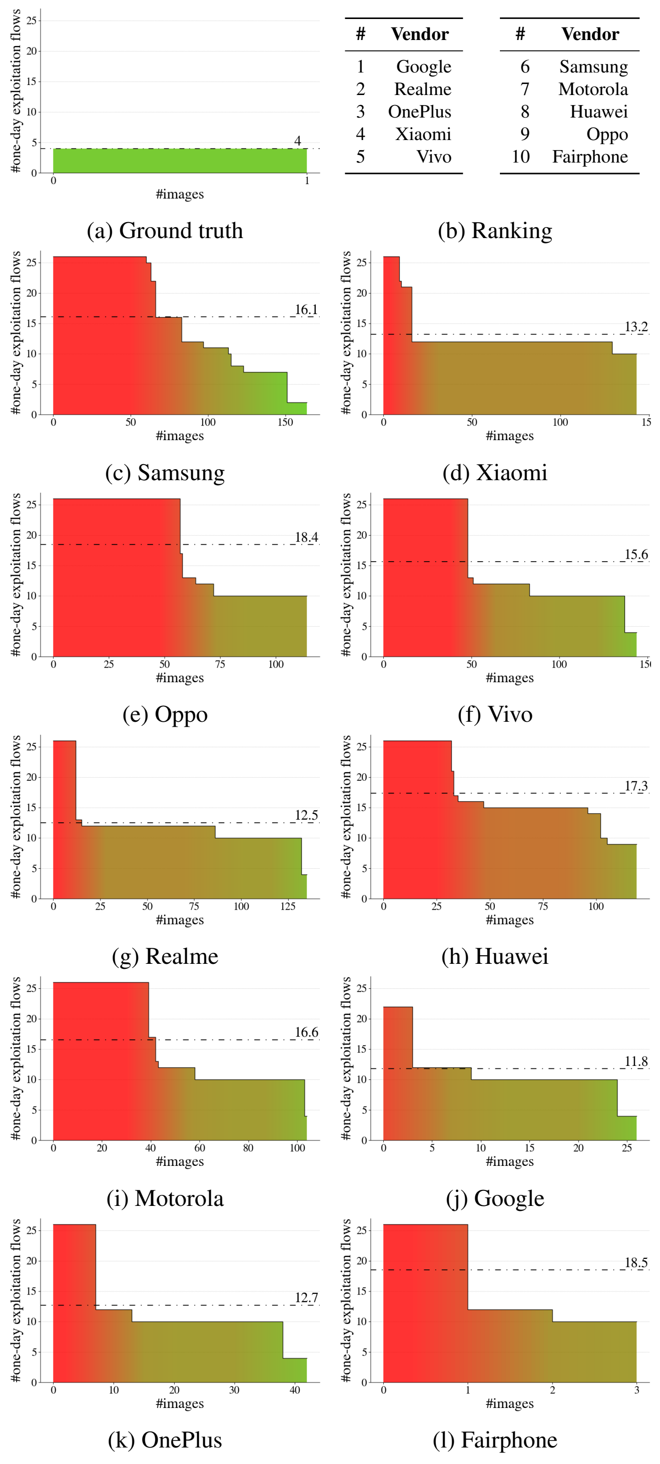
ploit flows. In particular, significant portions of the firmwares do not include defenses such as `CONFIG_DEBUG_LIST`, which is critical to mitigate BadBinder [46].

Susceptibility. Using data from Section 5.2 and Table 2, we evaluate the effectiveness and assess the susceptibility of firmwares to one-day exploitation flows. We consider a firmware to be susceptible to a one-day exploitation flow if it does not include a defense that can prevent the vulnerability-agnostic exploitation flow. Figure 9 illustrates the overall susceptible one-day exploitation flows per firmware with two curves. The dashed line depicts the impact of the widespread defense lack, while the outer line incorporates both the lack and efficacy shortcomings (see Section 5.2), providing a more comprehensive view. Without these shortcomings, on average, nearly two one-day exploitation flows could have been prevented. Both findings highlight the worrying situation and lack of effective defenses to prevent exploitation flows.

Takeaway 1

Even though effective defenses (see Table 2) for a large share of the one-day exploitation flows are available, they are rarely activated in vendor-provided kernels.

Susceptibility per Vendor. We further organize the results by vendor, presenting each in Figure 10. Figure 10a depicts the ground truth, showcasing the maximum achievable security with all available mainline defenses. Figure 10c-10l show each vendor's susceptible one-day exploitation flows, including the lack of defenses and efficacy shortcomings. We specifically highlight Google, Fairphone, and Samsung, representing the most and least secure, and with the highest market share. Their susceptibility is 11.8, 18.5, and 16.1, respectively, while the ground truth has 4. We compute the factor by which they are worse than the ground truth, resulting in 2.95 ($\approx \frac{11.8}{4}$), 4.62, and 4. Figure 10b presents the ranking of vendors according to this deterioration factor.



(a) Ground truth

#	Vendor	#	Vendor
1	Google	6	Samsung
2	Realme	7	Motorola
3	OnePlus	8	Huawei
4	Xiaomi	9	Oppo
5	Vivo	10	Fairphone

(b) Ranking

(c) Samsung (d) Xiaomi

(e) Oppo (f) Vivo

(g) Realme (h) Huawei

(i) Motorola (j) Google

(k) OnePlus (l) Fairphone

Figure 10: Analysis results per devices for each vendors.

Takeaway 2

Protection against one-day exploitation flows is highly vendor dependent, varying between a 4.62 to 2.95 worse scenario than applying all available mainline defenses.

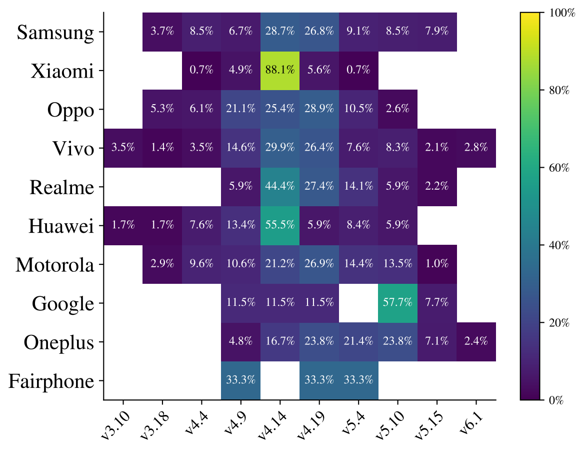


Figure 11: Applied Android kernel versions for each vendor.

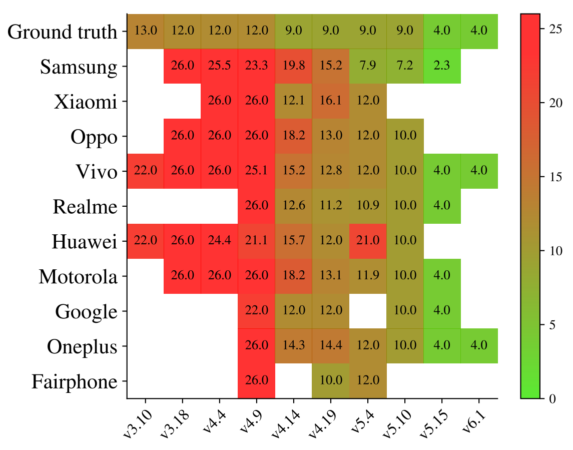


Figure 12: Susceptible exploitation flows per version/vendor.

Susceptibility per Kernel Version. To illustrate a version dependency, we initially obtain the used kernel versions. Figure 11 shows the results covering v3.10 to v6.1. For context, v4.19 was released in 2018, while v3.10 was released in 2014. We then analyze the susceptibility to one-day exploitation flows, organized by kernel version and vendor (see Figure 12). The figure includes a ground truth, representing how many exploitation flows remain susceptible for a given kernel version with all available defenses integrated (see Table 6).

Three findings emerge from this analysis: First, almost no device kernel prior to v4.14 includes any defenses. Since ret2bpf (**ET5**) is not exploitable on v3.10, it may be less susceptible than v3.18. Second, newer kernels tend to have more active protection against exploitation flows, observed across almost all vendors. This is particularly true for those obeying the GKI constraints ($\geq$v5.4 for GKI-1.0 or $\geq$v5.10 for GKI-2.0). Third, although newer kernels provide more defenses, a v3.10 kernel with all available defenses enabled would protect more flows than 38.1 % of our analyzed kernels.

Takeaway 3

While newer kernels provide more defenses, a v3.10 kernel with all available defenses enabled would mitigate more exploitation flows than 38.1 % of vendor-supplied kernels.

Susceptibility per Low/High-End Device. We differentiate the susceptibility according to whether it is a low-end or a high-end device: We initially compute the average one-day susceptibility of the latest low-end and high-end devices from vendors offering both classes, i.e., all except Google and Fairphone (see Table 7). We then compute the susceptibility reduction of high-end compared to low-end devices. For instance, with a susceptibility score of 4.5 and 5.5 for high-end and low-end Samsung devices, respectively, the reduction is 18.2 %. Overall, the reduction is between 0 % to 63.6 %, with an average value of 23.8 %, which indicates a significant reduction of high-end to low-end devices.

Takeaway 4

There is a significant gap of 23.8 % between the one-day susceptibility of high-end and low-end devices.

6 Discussion

Factors Potentially Contributing to the Absence of Effective Defenses. Our analysis, highlighted in Takeaway 1, reveals a concerning reality: vendors lack the inclusion and effectiveness of defenses against one-day exploitation flows. Here we discuss potential factors contributing to this situation.

First, as indicated by Takeaway 2, there is variability in susceptibility to one-day exploitation flows across vendors. While Google and OnePlus demonstrate lower susceptibility, others like Huawei show higher ones. As these vendors utilize different kernel versions, we observe a correlation between higher susceptibility and the use of older versions. Hence, a potential contributing factor is the *use of older kernel versions*.

Second, as emphasized in Takeaway 3, susceptibility extends beyond mere kernel version correlation. Even the deprecated kernel v3.10 (released about ten years ago) would mitigate more one-day exploitation flows, if properly configured, than 38.1 % of vendor firmwares. Huawei underscores this statement with their v5.4.86 kernels, nearly twice as bad as the properly configured v3.10. This lack of proper configuration appears prevalent across multiple vendors. Hence, the second potential contributor is *a lack of importance regarding security-relevant features for the Android kernel*.

Third, as shown in Takeaway 4, we observe that low-end are more susceptible to one-day exploitation flows than high-end devices, as observed by most vendors. On the one hand, low-end devices tend to be less powerful than high-end devices, and on the other hand, enabling defenses increases the performance overhead. To compensate for this performance cost, vendors may deliberately not enable defenses for performance gains. Therefore, the third potential factor is *performance cost, especially for less powerful low-end devices*.

Recommendation to Improve Android Security. With these insights, we propose that Google updates the Android Compatibility Definition Document (CDD), which outlines the requirements for devices to be compatible with Android. While for Android 14 some fundamental defenses are recommended (e.g., `CONFIG_CFI_CLANG`) or required (e.g., `CONFIG_STRICT_KERNEL_RWX`), other critical ones are absent (e.g., `CONFIG_DEBUG_LIST`). By including our findings, we anticipate a substantial improvement in Android security.

Responses. Google responded that they are aware of this problem and are gradually enforcing kernel defenses that will be integrated. However, as defenses can come at a performance cost, enforcing them across all vendors is difficult, especially for low-end devices. They pointed out that `CONFIG_DEBUG_LIST` has been enforced in the past, but vendors complained about the performance hit. This resulted in critical defenses not being integrated. Samsung and Huawei responded similarly, as integration comes at a performance cost, i.e., Samsung for not activating RKP on all (especially low-end) devices and Huawei for not protecting all page tables. These responses highlight our third potential contribution factor. Fairphone and Motorola acknowledged our findings and integrated defenses, while the others did not respond.

Automation and Standardization. Fully automating the analysis process would enhance the demonstration of the effectiveness of defenses. We have already automated several steps, such as parts of the firmware and kernel code acquisition, metadata extraction, and defense analysis, all of which are scalable. Challenges remain in the acquisition and analysis of zero-days and the acquisition of all firmware. Our work addresses these challenges manually and encourages standardization, drastically reducing manual effort. Therefore, our work addresses current technical challenges and encourages progress for future identification of effectively integrated defenses, ultimately improving Android security.

False Negatives/Positives. A false negative occurs when we interpret a device as being susceptible to an exploitation flow when it is not. This could have happened if we have overlooked defenses. To ensure we identified all mainline defenses, we executed each exploitation technique (**ET1-10**) with security measures enabled, resulting in the defenses (**DM1-8**) preventing these exploits. To ensure that we have identified all downstream defenses, we performed a semi-automated analysis of the 1533 downstream kernels in Section 5.2.2, which yielded 3 vendor-specific defenses. While misinterpreted firmware analysis could also lead to false negatives, most defenses are intrinsically tied to the kernel's core functionalities. As described in Section 5.2.1, those defenses that are not intrinsically tied can only lead to false positives, i.e., we interpret a device as mitigating an exploitation flow when it does not. This means our results can be interpreted as conservative, and the real world may be even more worrying.

7 Related Work

Large-scale Firmware Analysis. Possemato et al. [43] investigated compliance with Android's compatibility guidelines and found customizations as security drawbacks. Subsequent work [24] has highlighted delays in adopting critical patches. Other studies scanned ROMs for insecure access policies [16, 23] or privacy-intruding apps [20, 24, 52]. For embedded systems, researchers have uncovered vulnerabilities at a large scale [14, 19] and revealed a reluctance to activate attack mitigations in Linux-based IoT devices [64].

Android Security Patch Ecosystem. Prior works studied the deployment of security updates to Android systems. Wu et al. [57] noted that most Android Security Bulletin (ASB) issues stem from native code. Farhang et al. [18] found that CVEs in the kernel took the longest to propagate to vendor ASBs, while other researchers [29, 67] reported weeks to months of delay in deploying Android security updates.

Patch Detection. Researchers proposed strategies to detect patches in kernel binaries. Zhang et al. [66] presented a detection approach by deriving a signature from the mainstream version, which is then compared with target kernels. PDiff [27] statically extracts the semantics of source-level patches and uses a similarity-based measure to detect patches in compiled kernels. Dynamic approaches [26, 68] automatically adapt existing PoC exploits to different kernel variants.

Vulnerability Patching. Researchers have proposed solutions to address the long delays in kernel patch deployment. Wang et al. [56] prevented bugs discovered by a sanitizer from being triggered and, hence, exploited till a patch is available. Talebi et al. [53] instrumented vulnerable syscall implementations to undo harmful side-effects. Other researchers focused on downstream Android kernels. Chen et al. [13] proposed hot-patching with Lua code to filter vulnerable function arguments. Xu et al. [61] extended this by suggesting automated binary hot patches from source-level upstream fixes.

Zero-Day Analysis. Google Project Zero [9, 45, 49] and Threat Analysis Group [50] hunt for zero-days in the wild. They release public findings covering various entities, e.g., Android phones, significantly enhancing system security.

8 Conclusion

This work conducted a one-day analysis of Android devices, combined with an analysis of defense inclusion and effectiveness. Our findings unveiled a significant gap between the current state of Android security and its maximum potential. We discussed potential contributing factors and offered recommendations for improvement, enhancing Android security.

Acknowledgements

We thank Mathias Oberhuber, Andreas Kogler, the anonymous reviewers, and our shepherd for their valuable feedback.

This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087). Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *CCS*, 2005.

[2] Al Viro. iov_iter: saner checks on copyin/copyout, 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=09fc68dc66f7597bdc8898c991609a48f061bed5.

[3] Android. Kernel Control Flow Integrity, 2022. URL: https://source.android.com/docs/security/test/kcfi.

[4] Android. Overview, 2024. URL: https://source.android.com/docs/core/architecture/partitions.

[5] AppBrain. Top Android OS versions, 2023. accessed: 28.11.2023. URL: https://web.archive.org/web/20231128122419/https://www.appbrain.com/stats/top-android-sdk-versions.

[6] AppBrain. Top manufacturers, 2023. accessed: 14.09.2023. URL: https://web.archive.org/web/20230915054021/https://www.appbrain.com/stats/top-manufacturers.

[7] Brandon Azad. An iOS hacker tries Android, 2020. URL: https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html.

[8] Brandon Azad. A survey of recent ios kernel exploits, 2020. URL: https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html.

[9] Ian Beer. Mind the Gap, 2022. URL: https://googleprojectzero.blogspot.com/2022/11/.

[10] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[11] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[12] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *USENIX Security*, 2020.

[13] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android Kernel Live Patching. In *USENIX Security*, 2017.

[14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security*, 2014.

[15] Samsung Knox Documentation. Real-time Kernel Protection (RKP), 2023. URL: https://docs.samsungknox.com/admin/fundamentals/whitepaper/core-platform-security/real-time-kernel-protection/.

[16] Zeinab El-Rewini and Yousra Aafer. Dissecting residual apis in custom android roms. In *CCS*, 2021.

[17] Everphone. What is the average smartphone lifespan?, 2023. URL: https://web.archive.org/web/20231123081219/https://everphone.com/en/blog/smartphone-lifespan/.

[18] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *WWW*, 2020.

[19] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *CCS*, 2016.

[20] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. In *S&P*, 2020.

[21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[22] Hemanth. Extractor of SpreadTrum firmware files with extension pac, 2023. URL: https://github.com/HemanthJabalpuri/pacextractor.

[23] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R. B. Butler. Bigmac: fine-grained policy analysis of android firmware. In *USENIX Security*, 2020.

[24] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. Large-scale security measurements on the android firmware ecosystem. In *International Conference on Software Engineering (ICSE)*, 2022.

[25] Huawei. Emui 11.0 security technical white paper, 2020. URL: https://consumer.huawei.com/content/dam/huawei-cbg-site/common/campaign/privacy/whitepaper/emui_11.0_security_technical_white_paper_v1.0.pdf.

[26] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. Aem: Facilitating cross-version exploitability assessment of linux kernel vulnerabilities. In *S&P*, 2023.

[27] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. In *CCS*, 2020.

[28] Xingyu Jin and Richard Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel, 2021. URL: https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf.

[29] Kailani R. Jones, Ting-Fang Yen, Sathya Chandran Sundaramurthy, and Alexandru G. Bardas. Deploying android security updates: an extensive study involving manufacturers, carriers, and end users. In *CCS*, 2020.

[30] Max Kellermann. The Dirty Pipe Vulnerability, 2022. URL: https://dirtypipe.cm4all.com/.

[31] Bjoern Kerler. oppo_decrypt_ozip, 2022. URL: https://github.com/bkerler/oppo_ozip_decrypt.

[32] Bjoern Kerler. oppo_decrypt, 2023. URL: https://github.com/bkerler/oppo_decrypt.

[33] Zhenpeng Lin. How AUTOSLAB Changes the Memory Unsafety Game, 2021. URL: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game.

[34] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In *ACM*, 2022.

[35] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android, 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf.

[36] Linux Kernel Driver DataBase. CONFIG_ARM64_UAO: Enable support for User Access Override (UAO), 2024. URL: https://cateee.net/lkddb/web-lkddb/ARM64_UAO.html.

[37] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *NDSS*, 2017.

[38] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *USENIX Security*, 2024.

[39] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In *ACSAC*, 2023.

[40] Marin. vmlinux-to-elf, 2023. URL: https://github.com/marin-m/vmlinux-to-elf.

[41] Man Yue Mo. One day short of a full chain: Part 1 - Android Kernel arbitrary code execution, 2021. URL: https://securitylab.github.com/research/one_day_short_of_a_fullchain_android/.

[42] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel, 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.

[43] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In *S&P*, 2021.

[44] Matteo Rizzo and Jann Horn. Prevent cross-cache attacks in the SLUB allocator, 2023. URL: https://lore.kernel.org/linux-mm/202309151425.2BE59091@keescook/T/.

[45] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack, 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html.

[46] Maddie Stone. Bad Binder: Android In-The-Wild Exploit, 2019. URL: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[47] Maddie Stone. CONFIG_DEBUG_LIST=y, 2020. URL: https://twitter.com/maddiestone/status/1245834936629616640?lang=de.

[48] Maddie Stone. Detection Deficit: A Year in Review of 0-days Used In-The-Wild in 2019, 2020. URL: https://googleprojectzero.blogspot.com/2020/07/detection-deficit-year-in-review-of-0.html.

[49] Maddie Stone. 2022 0-day In-the-Wild Exploitation...so far, 2023. URL: https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html.

[50] Maddie Stone. The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022, 2023. URL: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html.

[51] Superr. splituapp, 2019. URL: https://github.com/superr/splituapp.

[52] Thomas Sutter and Bernhard Tellenbach. Firmwaredroid: Towards automated static analysis of pre-installed android apps. In *MOBILESoft*, 2023.

[53] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo workarounds for kernel bugs. In *USENIX Security*, 2021.

[54] USA Today. How long before a phone is outdated? Here's how to find your smartphone's expiration date, 2023. URL: https://web.archive.org/web/20231022153016/https://eu.usatoday.com/story/tech/columnist/komando/2023/10/22/how-to-find-smartphone-expiration-date/71255625007/.

[55] Vasya. payload dumper, 2023. URL: https://github.com/vm03/payload_dumper.

[56] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel. In *USENIX Security*, 2023.

[57] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards understanding android system vulnerabilities: Techniques and insights. In *AsiaCCS*, 2019.

[58] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel, 2023. URL: https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.

[59] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *USENIX Security*, 2019.

[60] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *CCS*, 2015.

[61] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic Hot Patch Generation for Android Kernels. In *USENIX Security*, 2020.

[62] Jun Yao. arm64/mm: move {idmap_pg_dir,tramp_pg_dir,swapper_pg_dir} to .rodata section, 2018. URL: https://patchwork.kernel.org/project/linux-hardening/patch/20180620085755.20045-2-yaojun8558363@gmail.com/.

[63] Wang Yong. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features, 2018. URL: https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf.

[64] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building embedded systems like it's 1996. In *NDSS*, 2022.

[65] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *USENIX Security*, 2022.

[66] Hang Zhang and Zhiyun Qian. Precise and Accurate Patch Presence Test for Binaries. In *USENIX Security*, 2018.

[67] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *USENIX Security*, 2021.

[68] Xiaochen Zou, Yu Hao, Zheng Zhang, Juefei Pu, Weiteng Chen, and Zhiyun Qian. Syzbridge: Bridging the gap in exploitability assessment of linux kernel bugs in the linux ecosystem. In *NDSS*, 2024.

A Detailed Statistics

A.1 Detailed Defense Detection of Kernels

Table 3 illustrates the comprehensive list of how we assess the state of our identified defense mechanisms. We follow the procedure to identify symbols of globally reachable functions within the `kallsyms` file. This file contains all globally reachable functions and variable symbols used in the kernel binary, e.g., marked with `EXPORT_SYMBOL`. For instance, the presence of `__list_add_valid` in `kallsyms` indicates the status of the `CONFIG_DEBUG_LIST`. As another example, the symbol `cache_random_seq_create` indicates the presence of `CONFIG_SLAB_FREELIST_RANDOM`. A similar assessment stands true for detecting the other defenses. Additionally, to identifying symbols `kallsyms`, our approach also detects defense mechanisms which do not contain globally reachable

Table 3: Symbols and used additional information for our defense detection approach. The defense feature ☆ is enabled if this symbol (e.g., globally visible function or variable) is present within the `kallsyms` containing all kernel symbols.

Defense Feature	Kernel Executable	Present within `kallsyms`☆	Information
`CONFIG_DEBUG_LIST`		`__list_add_valid`, `__list_del_entry_valid`	≥v3.18
		`__list_add`, `__list_del_entry`	<v3.18
`CONFIG_CFI_CLANG`		`cfi_module_add`, `cfi_module_remove`	
`CONFIG_BPF_JIT_ALWAYS_ON`		`___bpf_prog_run`	≥v4.14
		`__bpf_prog_run`	<v4.14
`kmalloc-cg-*`			available for ≥v5.13
`CONFIG_INIT_ON_ALLOC_DEFAULT_ON`		`init_on_alloc`	
`CONFIG_ARM64_UAO`		`uao_thread_switch`, `cpu_enable_uao`	no `addr_limit` for ≥v5.11
`CONFIG_SLAB_FREELIST_HARDENED`	`get_random_long`		called within `__kmem_cache_create`
KSMA Protection	`swapper_pg_dir`, `tramp_pg_dir`	`swapper_pgdir_lock`, `swapper_pg_dir`, `tramp_pg_dir`	`*_pg_dir` mapped as read-only
`Samsung RKP`		`rkp_init`	only for Samsung devices
`CONFIG_RANDOMIZE_BASE`		`module_alloc_base`, `kaslr_early_init`	
`CONFIG_STRICT_KERNEL_RWX`		`set_debug_rodata`, `mark_readonly`, `mark_rodata_ro`	named as `CONFIG_DEBUG_RODATA` for <v4.14; on v3.10 only for 32 bit systems
`CONFIG_ARM64_PAN`		`cpu_enable_uao`	
`CONFIG_ARM64_SW_TTBR0_PAN`		`reserved_ttbr0`	<v5.4; available for ≥v4.10
	`"emulated: Privileged Access Never (PAN) \ using TTBR0_EL1 switching"`		≥v4.19
`CONFIG_SLAB_FREELIST_RANDOM`		`cache_random_seq_create`, `cache_random_seq_destroy`	
`CONFIG_UNMAP_KERNEL_AT_EL0`		`tramp_pg_dir`	

Table 4: Statistical results of firmware extraction and kernel code collection.

Vendors	Firmware Extraction			Kernel Code
	#devices	#available	#extracted	#collected
Samsung	197	190	164	654
Xiaomi	278	151	143	188
Oppo	229	145	114	29
Vivo	307	178	144	30
Realme	307	137	135	135
Huawei	182	121	119	218
Motorola	115	112	104	246
Google	26	26	26	9
OnePlus	54	46	42	21
Fairphone	3	3	3	3
Total	1698	1109	994	1533

symbols. For instance, `CONFIG_SLAB_FREELIST_HARDENED` only includes inline functions and member variables. To detect the presence of this defense, our approach analyzes the kernel binary, more specifically, the function where these inline calls are executed, e.g., `get_random_long` within function `kmem_cache_open`. Executing the call indicates the presence of this defense. To detect the presence of the KSMA protection, `swapper_pg_dir` and `tramp_pg_dir` must also be mapped read-only. For instance, these pages might be mapped between `__start_rodata` and `__init_begin`.

A.2 Statistical Results of Firmware Extraction

Table 4 illustrates the extractable firmwares. Our success rate of 58.5 % (with a collection and extraction rate of 65.3 % and 89.6 %) from produced devices to extractable firmwares aligns with prior work [11, 14, 64]. The two main reasons for extraction failure were that our approach did not recognize the correct format or that part of the firmware was corrupted.

Table 5: Included defenses averaged over all firmwares for each vendor. ✶ inidcates that it is ineffective while ☆ indicates that it is ineffective for kernels <v5.11.

Vendor	☰	</>	⑂	▦	■	➦☆	🔗	🏛✶	♻	⌨	🔨	🔀	🏰	📞	✹
Samsung	60	49	26	8	63	96	25	5	84	100	91	27	16	39	
Xiaomi	89	94	74	0	93	98	10	1	97	100	97	10	83		
Oppo	50	49	19	0	44	95	37	13	91	100	95	49	14		
Vivo	69	65	27	5	67	96	44	22	95	98	88	73	22		
Realme	91	91	34	2	89	100	36	22	100	100	99	47	44		
Huawei	15	18	67	0	20	92	12	13	97	100	79	87	14		62
Motorola	62	58	34	1	59	90	44	29	89	100	79	58	31		
Google	88	88	100	8	88	100	65	65	100	100	100	77	65		
Oneplus	83	83	52	10	83	100	69	55	100	100	100	90	55		
Fairphone	67	67	33	0	67	100	33	33	100	100	100	67	33		

☰ `CONFIG_DEBUG_LIST` </> `CONFIG_BPF_JIT_ALWAYS_ON` ⑂ `CONFIG_CFI_CLANG` ▦ `kmalloc-cg-*`
■ `CONFIG_INIT_ON_ALLOC_DEFAULT_ON` ➦ `CONFIG_ARM64_UAO` 🔗 `CONFIG_SLAB_FREELIST_HARDENED`
🏛 KSMA protection ♻ `CONFIG_RANDOMIZE_BASE` ⌨ `CONFIG_STRICT_KERNEL_RWX`
🔨 `CONFIG_ARM64_(SW_TTBR0_)PAN` 🔀 `CONFIG_SLAB_FREELIST_RANDOM` 🏰 `CONFIG_UNMAP_KERNEL_AT_EL0`
📞 `Samsung RKP` ✹ `Huawei HKIP`

Table 6: ✓ indicates defenses available for mainline Android kernel from v3.10 to v6.1, while ✗ indicates that the defense is not required for the specific version.

Kernel	☰	</>	⑂	▦	■	➦	🔗	🏛	♻	⌨	🔨	🔀	🏰
v3.10	✓	✗								✓			
v3.18	✓	✓				✓			✓	✓	✓		
v4.4	✓	✓				✓			✓	✓	✓		✓
v4.9	✓	✓	✓			✓			✓	✓	✓	✓	✓
v4.14	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓
v4.19	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓
v5.4	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
v5.10	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
v5.15	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
v6.1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓

☰ `CONFIG_DEBUG_LIST` </> `CONFIG_BPF_JIT_ALWAYS_ON` ⑂ `CONFIG_CFI_CLANG` ▦ `kmalloc-cg-*`
■ `CONFIG_INIT_ON_ALLOC_DEFAULT_ON` ➦ `CONFIG_ARM64_UAO` 🔗 `CONFIG_SLAB_FREELIST_HARDENED`
🏛 KSMA protection ♻ `CONFIG_RANDOMIZE_BASE` ⌨ `CONFIG_STRICT_KERNEL_RWX`
🔨 `CONFIG_ARM64_(SW_TTBR0_)PAN` 🔀 `CONFIG_SLAB_FREELIST_RANDOM` 🏰 `CONFIG_UNMAP_KERNEL_AT_EL0`

Table 7: The susceptibility reduction (i.e., **Susc Reduc**) against one-days of high-end to low-end devices.

Vendor	Low-End		High-End		Susc Reduc
	Devices	Susc	Devices	Susc	in %
Samsung	Galaxy A(1,2,3,5)4	5.5	Galaxy S23.*	4.5	18.2
Xiaomi	Redmi 12.*	12.0	13T.*	12.0	0.0
Oppo	A(3,9)8	12.0	Find X2.*	10.0	16.7
Vivo	Y(100,27)	11.0	X100.*	4.0	63.6
Realme	C(33,53,55)	10.7	Neo 5.*	10.0	6.2
Huawei	Nova 11.*	15.5	P60.*	10.0	35.5
Motorola	G(1,5,8)4.*	10.7	Edge 40.*	8.5	20.3
OnePlus	Nord 3.*	10.0	11.*	7.0	30.0
Mean					23.8