IUBIK: Isolating User Bytes in Commodity Operating System Kernels via Memory Tagging Extensions

Marius Momeu
Technical University of Munich
marius.momeu@tum.de

Alexander J. Gaidis
Brown University
agaidis@cs.brown.edu

Jasper v.d. Heidt
Technical University of Munich
jasper.von-der-heidt@tum.de

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

***Abstract*—Hardening OS kernels against memory errors is generally addressed by protecting security-critical data against corruption and disclosure. However, establishing a sound model for identifying sensitive memory objects in need of protection is hard, leading to emergent attack vectors that can be abused by attackers. In this paper, we propose rethinking how OS kernels are hardened by introducing `IUBIK` for compartmentalizing kernel memory. `IUBIK` prevents kernel exploitation by segregating attacker-controlled data—frequently used to manipulate security-critical data—in shadow memory, preventing it from interacting with sensitive kernel objects. To achieve this, `IUBIK` uses MTE: a recent hardware feature, available in ARM CPUs, which allows mitigating exploits based on both spatial and temporal memory-errors, efficiently. We ensure that segregated objects do not contain sensitive fields, such as pointers, by rewriting their `struct` definitions. Moreover, we develop a profiling framework that explores the kernel codebase in-depth and records code sites where attacker-controlled objects are allocated, allowing `IUBIK` to isolate them; our profiler recorded 292 privileged and 212 non-privileged allocation sites for a diverse set of workloads. Finally, we evaluate an implementation of `IUBIK` for the Linux kernel, across a suite of micro- and macro-benchmarks, demonstrating that our prototype incurs no runtime overhead in most tests and negligible additional memory consumption.**

1. Introduction

The past few years have witnessed a surge in publicly-released exploits that demonstrate how memory errors in OS-kernel code allow attackers to take complete control of a system (see Table 7 in Appendix A). These are also complemented by offensive-security research that improves exploit reliability [1], [2], proposes novel exploitation techniques [3], [4], and automates identifying exploitation primitives [5], [6]. In response, researchers have developed hardware-assisted, memory-isolation frameworks to protect *sensitive* kernel memory against corruption, neutralizing kernel exploits [7], [8], [9], [10]. However, identifying data that is *security-sensitive* and should be protected in monolithic OS kernels remains an open problem, with recent work continuing to uncover new objects that attackers may target to circumvent existing defenses [11], [12].

In this work, we propose rethinking how to tackle kernel security by ensuring its integrity without having to identify individual, sensitive objects. Instead, our approach focuses on neutralizing the *primitives* abused in exploits to corrupt security-critical data. This insight is drawn after our systematic analysis of kernel exploits published during 2020–2024, which reveals that most of them abuse a memory error with limited capabilities —i.e., *use-after-free (UAF)* or *object-out-of-bounds (OOB)*—to manipulate a kernel object by overlapping it with a *user-controlled object* containing data supplied by a malicious program (see Table 7 in Appendix A). As such, we call these user-controlled objects *usercopy primitives*. Notably, usercopy primitives are the standard way of *leaking* code pointers to defeat *KASLR* and *overwriting* them to build code-reuse chains in Linux (usually in that order) [13]. They are also the main method leveraged by SLUBStick [1] to manipulate page tables and takeover the OS kernel.

The Linux kernel tries to limit the use of such primitives by allocating user-controlled objects in dedicated memory pools, known as *object caches*, which are separated from other kernel data that may be targeted by attacks. Nevertheless, our exploit analysis reveals that many of them are still allocated from caches containing kernel objects targeted by existing attacks, facilitating *same-cache* attacks [13]. Even if all user-controlled objects were allocated from dedicated caches, the lack of strong memory isolation to enforce their separation from sensitive kernel data still allows attackers to overlap them and mount *cross-cache* attacks [1], [14].

To tackle these problems we introduce `IUBIK`: a novel solution for compartmentalizing memory within monolithic OS kernels. `IUBIK` isolates user-controlled objects in *shadow memory* and prevents them from ever accessing security-critical data. In doing so, `IUBIK` is able to break existing and future kernel exploits that hinge on abusing user-controlled objects to corrupt sensitive data, regardless of their type (e.g., process credentials or page tables) or targeted vulnerability (i.e., UAF or OOB). Achieving this level of isolation requires overcoming several challenges.

Challenge #1: Isolating Memory. First, we require a memory isolation primitive capable of confining user-controlled objects in both spatial and temporal memory-safety scenarios. However, existing solutions focus solely on either spatial [7], [8], [10] or temporal [15] violations, or provide

only probabilistic integrity [9]. In contrast to existing solutions, IUBIK relies on ARM's recently introduced *Memory Tagging Extension (MTE)* feature [16] that provides both spatial and temporal safety guarantees with low overhead. We equip IUBIK with MTE and introduce a novel memory-isolation framework for OS kernels that breaks exploits by enforcing strong isolation between user-controlled objects and the rest of kernel memory. IUBIK only requires two MTE tags to maintain its two *isolation domains*: one for user-controlled objects and the other for everything else. This way, IUBIK does not inherit the limitations that stem from the small number of available MTE tags, which have raised concerns in prior work [17], [18], [19].

Challenge #2: Structure Layout. Second, we identify several exploits that abuse *flexible arrays* stored at the end of C `structs` [20] to transfer data to and from user space and to place a target object closer to a victim object (à la *heap feng shui* [21]). Two of the most widely used object types in kernel exploitation, `struct msg_msg` and `struct user_key_payload`, follow exactly this pattern. Many of these objects mingle sensitive kernel fields (e.g., function pointers) with user-controlled bytes in a single C `struct`, allowing attackers to abuse memory errors to corrupt them even when they are isolated in an MTE domain. To combat this, we rewrite such `struct` definitions to extract user-controlled fields into shadow memory where IUBIK can isolate them securely. In case the extracted fields need to refer back the host object (e.g., in idioms like `container_of`), we maintain a backward pointer in the extracted field that references the original object. To prevent user-controlled objects from corrupting this reference, we encrypt it using ARM's *Pointer Authentication* [22] feature.

Challenge #3: Identifying Allocation Sites. Finally, in order to allocate select heap objects in shadow memory, we need to instrument their *allocation sites* (called *allocsites* henceforth). To achieve this, we first instrument allocsites of user-controlled objects derived from public exploits and prior work. However, we also aim to reveal other potentially dangerous objects that may be user-controlled and instrument them before they become abused in future exploits. Yet doing so through manual code analysis would be a herculean task given the size of an OS kernel. Additionally, existing tools based on static analysis [5] exhibit a large number of false positives, making it hard for security analysts to triage them and identify suitable candidates. Consequently, to address the third challenge, we develop a *memory profiling framework* for OS kernels that is able to *record* objects accessed from user space and recover their allocsites—we call this tool the *usercopy profiler*. We execute a series of comprehensive payloads to explore the kernel's codebase in-depth, which reveals a large number of candidates that require instrumentation to reduce the kernel's attack surface.

Evaluation. We first evaluate the effectiveness of the usercopy profiler, where we record 292 privileged (i.e., they can be triggered by `root`) and 212 non-privileged (i.e., they can be triggered by regular users) allocation sites, respectively, on a diverse set of workloads. Subsequently, we evaluate IUBIK's performance and effectiveness.

For performance, we measure the runtime overhead IUBIK incurs in micro-benchmarks and real-world applications. In micro-benchmarks, IUBIK incurs worst-case slowdowns of 7%–8% in two benchmarks, while the remaining exhibit negligible ($< 5\%$) slowdown or none at all. In macro-benchmarks, IUBIK incurs a negligible slowdown of $< 3\%$ in all scenarios. We also measure the memory overhead of IUBIK at boot time (where a large number of user-controlled allocsites are exercised) and find IUBIK requires 2.17% more memory than SLUB during boot ($\approx$ 7MB). Lastly, we evaluate IUBIK's effectiveness via a suite of security tests and by demonstrating its ability to mitigate real-world exploits.

Contributions. We make the following contributions:

- We propose IUBIK: a novel memory-isolation solution that prevents kernel exploitation by isolating attacker-controlled objects in shadow memory using ARM MTE.
- We develop a set of patches that contain rewritten C `structs`, which allocate their user-controlled fields in shadow memory and use ARM PA to protect the few backward references that they need to store.
- We develop a dynamic profiling framework that identifies allocation sites of user-controlled objects requiring instrumentation, and share a dataset of our findings.
- We evaluate IUBIK on the Linux kernel in terms of performance and security effectiveness.

2. Background

2.1. Memory Errors

Software written in memory- and/or type-unsafe languages—e.g., C, C++, and ASM—is susceptible to memory errors in heap-allocated objects [2]. These errors can be broadly classified into two categories: *spatial memory errors* and *temporal memory errors*. Spatial memory errors occur when a pointer in a victim object is made to reference memory in a target object outside of its intended boundary. Such *out-of-bounds* (OOB) errors allow attackers to corrupt and/or leak data from a target object via a neighboring (i.e., linear OOB) or arbitrary (i.e., non-linear OOB) victim object. Attackers typically (ab)use OOB errors to stretch the bounds of a victim object to corrupt a target object.

In contrast, temporal memory errors occur when *(dangling) pointers* referencing a freed object are *reused* and can be accessed simultaneously in different execution contexts. Examples of such errors include: (i) *use-after-free* (UAF), where a dangling pointer can still be accessed even after its underlying memory object is freed; (ii) *double-free* (DF), where the same pointer is freed multiple times; and (iii) *invalid free* (IF), where an incorrect (potentially attacker-controlled) pointer is freed. Attackers typically (ab)use such temporal memory errors to create *type confusion* conditions, whereby they *overlap* a victim and a target object and leverage the former's dangling pointer to corrupt the latter.

2.2. Memory Allocation in Linux

Allocator Overview. The Linux kernel primarily uses two *dynamic memory allocators* to manage in-kernel memory: a *page allocator* and an *object allocator*. The former, called Buddy, leverages the *buddy system* to satisfy memory requests on a page granularity [23], while the latter leverages the *slab* approach to facilitate efficient memory allocation on a sub-page granularity [24]. The slab allocator, called SLUB, uses the underlying page allocator to reserve one or more physically-contiguous memory pages to form *object slabs* that store objects of the same size (i.e., *type*). To save memory space and increase performance, SLUB caches freed slots and reuses them for subsequent allocations.

In the presence of a temporal memory error, attackers abuse this weakness to create a type confusion condition by overlapping different objects from the same cache (i.e., *same-cache attacks*). Similarly, Buddy reallocates freed pages across caches, which attackers can abuse to overlap a victim object and a target object from different caches (i.e., *cross-cache attacks*). IUBIK aims to protect against both attack types by allocating attacker-controlled objects in isolated caches.

Allocation Profiling. Linux recently introduced a *memory allocation profiling* feature [25] that maintains a *codetag* for every call to SLUB's allocation routines, s.a., kmalloc and kmem_cache_alloc. The codetags are represented by a metadata object added to the kernel's data section at compile-time, storing several fields including an allocation site's source file and line number, as well as the function where it is invoked from. Notably, the framework provides a mechanism to allow retrieving an allocated object's codetag at run-time (i.e., its allocsite) using just its address.

2.3. ARM Hardware Features

Memory Tagging Extension. ARM Memory Tagging Extension (MTE) [16] is a hardware feature introduced in the ARMv8.5 instruction set to detect spatial and temporal memory-safety violations. For that, MTE provides two types of tags that implement a "lock and key" mechanism to mediate memory accesses, namely: address tags and memory tags. Address tags serve as the "key" and are represented by four bits in the top byte of every pointer. The Top Byte Ignore (TBI) feature is used here to instruct the CPU to ignore the top byte when dereferencing pointers, allowing software to avoid expensive masking operations. Memory tags serve as the "lock" and are also represented by four bits that are associated with every 16-byte (aligned) region (known as the tag granule) of memory.

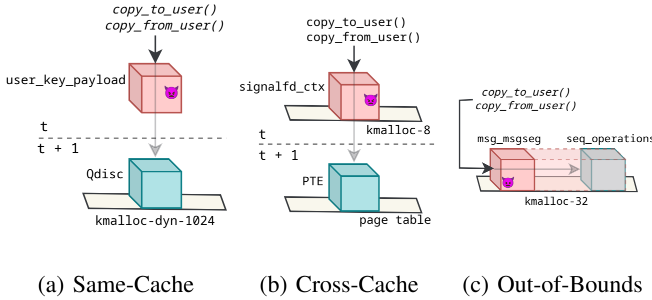
MTE extends the ARM instruction set with several new instructions that allow manipulating memory tags. When accessing memory, both the address tag and the memory tag must match, else a fault occurs according to the reporting mode: (i) *synchronous* raises the fault immediately, (ii) *asynchronous* records faults and raises them on the next context switch, and (iii) *asymmetric* records faults on read instructions and raises them on writes.

Pointer Authentication. ARM Pointer Authentication (PA) [22] was introduced in ARMv8.3-A to protect the integrity of pointers against tampering. At a high level, PA attaches cryptographic signatures to pointers and later verifies them before use. The signature is a Message Authentication Code (MAC)—called a Pointer Authentication Code (PAC) in this context—computed over the original pointer value and a 64-bit context (e.g., the address of the pointer) with a 128-bit key stored in a special CPU register. The PAC is stored in the top unused bits of hardened pointers, and its length varies between 3 and 31 bits, depending on hardware configuration. To prevent attackers from brute-forcing PAC values to guess valid signatures, e.g., via the PACMAN attack [26], ARM extends PA with the FPAC feature, which raises a fault on a failed PAC authentication.

3. Threat Model

Adversarial Capabilities. We assume a non-privileged attacker aiming to escalate privileges and control a system by exploiting memory-safety vulnerabilities in heap-allocated objects (i.e., managed by SLUB). Specifically, we allow an attacker to trigger one or more spatial or temporal memory errors on heap-allocated objects in the kernel—e.g., UAF, DF, IF, or OOB—at arbitrary times and as frequently as necessary to escalate their privileges. With this, an attacker can interact with the kernel via buggy interfaces, such as pseudo-filesystems (e.g., procfs [27] and debugfs [28]), the system call layer, and virtual device files (i.e., devfs [29]) to trigger an arbitrary sequence of (de-)allocations to trick SLUB to overlap a user-controlled object onto a kernel-sensitive one (e.g., one containing function pointers, process credentials, or page tables) either allocated on the same or different page via a *same-cache* or *cross-cache* attack, respectively. We consider memory errors on the stack or on the Buddy allocator out of scope. Regarding adversarial capabilities, our threat model is realistic and on par with the current state-of-the-{art, practice} regarding kernel-based heap exploitation [1], [6], [30].

Hardening Assumptions. We assume an OS that implements the W^X memory policy [31], [32], [33] in kernel space; hence, direct (shell)code injection in kernel memory is not attainable. Moreover, we presume that the kernel is hardened against ret2usr [34] attacks via PXN and PAN [35]. Lastly, the kernel may have support for KASLR [36], XOM [37], stack-smashing protection [38], proper .rodata sections (e.g., constification of critical data structures) [33], pointer (symbol) hiding [39], freelist randomization [40], [41], freelist obfuscation [42], randomized slab caches for kmalloc [43], AUTOSLAB [44], SLAB_VIRTUAL [45], hardened usercopy [46], (freed) memory poisoning [47], and CFI [48], [49], [50], [51], [52]. It may also use memory isolation frameworks that protect certain sensitive objects, such as process credentials or page tables [4], [7], [8], [9], [10], [53], and hardware-based techniques that aim at neutralizing temporal memory errors in Linux [15]. IUBIK does not require nor preclude any of the above, as they are orthogonal to its design. We



(a) Same-Cache (b) Cross-Cache (c) Out-of-Bounds

Figure 1: Scenarios for abusing usercopy primitives.

only require that ARM MTE and PA are supported and cannot be disabled. Finally, we consider side-channel [54], micro-architectural [55], [56], [57], fault-injection [58], [59], [60], and EPF [61] attacks out of scope.

4. Motivating Examples

User-controlled objects are crucial instruments that attackers abuse to build exploitation primitives. We identified 28 public exploits (in Linux) and a recent study on improving exploit reliability [1] that fall under this category. After closely examining them, we noticed that usercopy primitives are leveraged to exploit temporal- and spatial-memory errors, and they are abused to leak and corrupt both code and data pointers. We analyze three of them in the following section, outlining existing exploitation scenarios where user-controlled objects are abused.

Same-Cache Attacks. The exploit that abuses CVE-2023-0461 [62] leverages a UAF in a Linux instance hardened with `KMALLOC_SPLIT_VARSIZE`: a feature proposed to make heap exploitation harder by maintaining heap allocations with fixed sizes, known at compile-time, in a separate set of `kmalloc` caches than those with variable sizes (i.e., *elastic*), known at runtime [45]. The latter are generally more valuable to attackers, as their dynamic nature allows them to *massage* the heap and target objects in arbitrary caches, regardless of the targeted size. Nevertheless, this segregation is incomplete and can be circumvented.

Specifically, the attack first abuses the UAF on a vulnerable `struct tls_context` to overlap it with a `struct fqdir` in `kmalloc-512`, which is used for objects with a static size of 512 bytes. Then, the exploit *pivots* the UAF from `fqdir` to a UAF on its inner field of type `struct bucket_table`, that is allocated in `kmalloc-dyn-1k`, which is used for elastic objects that are 1024 bytes. There, the exploit turns to `user_key_payload` to make use of its *flexible array* field, whose size and contents are entirely controllable from user space; this allows targeting any kernel object with a variable size, regardless which `kmalloc-dyn` cache it is allocated in. The exploit transfers the UAF from `bucket_table` to `user_key_payload`, and crafts its size to overlap a `struct Qdisc` object, leading to a successful *same-cache attack*. Finally, the exploit calls `copy_{to,from}_user` on `user_key_payload` to first leak `Qdisc`'s function pointers to bypass KASLR before corrupting them to build a ROP chain for privilege escalation. This scenario is depicted in Figure 1a.

This exploit highlights several key issues. (1) User-controlled objects with a variable size (i.e., `struct user_key_payload`) facilitate exploiting temporal memory errors by allowing attackers to control their size and overlap them onto targeted kernel objects with sensitive fields (i.e., `struct Qdisc` and its function pointers). In fact, our systematic analysis on existing kernel exploits revealed that most attacks adopt this strategy. (2) Allocating elastic objects that mingle user-controlled flexible arrays, s.a., `user_key_payload`, in the same caches as kernel objects that store sensitive fields, s.a., `Qdisc`'s function and data pointers, enables attackers to leverage the former to corrupt the latter. This also applies to the Linux caches meant to segregate objects known to be user-controlled, called `kmalloc-cg-*`, as well as to a similar mitigation proposed by an existing study on exploitable elastic objects in the kernel [5]. The exploit that abuses CVE-2023-3390 [63] demonstrates how attackers can circumvent these by mounting a same-cache attack solely using objects segregated in the `kmalloc-cg-*` caches. (3) Although objects known to be user-controlled should be allocated in the `kmalloc-cg-*` caches, `user_key_payload` (and many other kernel objects) are *still* allocated in the regular `kmalloc-*` or `kmalloc-dyn-*`, making them a valuable resource for manipulating sensitive memory, as demonstrated by existing exploits. Finally, (4) simply segregating objects with a fixed size from those with a variable size is insufficient, as attackers may still manipulate pointers stored by the former to pivot temporal errors onto the latter.

Cross-Cache Attacks. SLUBStick [1] and several other public exploits [14], [64], [65] demonstrate that kernel attackers can inflict type confusion with user-controlled objects even when they are allocated in a different cache (or on a different page) than the targeted objects. They achieve this by mounting *cross-cache* attacks (§2.1). For example, one of the PoCs published by SLUBStick first obtains a dangling pointer to a `struct signalfd_ctx` by inducing a UAF on it. Then, it massages `Buddy` to reclaim the affected page as a page used for page tables—a boilerplate cross-cache attack. Then, the exploit calls `copy_from_user` on the dangling pointer, allowing the attacker to manipulate the overlapping PTE in the targeted page table. We depict this scenario in Figure 1b. In fact, all of the code patterns that SLUBStick relies upon (to corrupt page tables) leverage user-controlled objects whose pages were reclaimed as page tables after a cross-cache attack. Having access to dangling pointers on such objects, attackers are able to manipulate the contents of the underlying page tables through `copy_from_user`. Note that simply allocating the objects used by SLUBStick in a segregated cache, such as the `kmalloc-cg-*` caches, is insufficient to mitigate the attack, since attackers may also mount cross-cache attacks against them—there is currently no mechanism that addresses cross-cache attacks in Linux.

Additionally, several object types identified by SLUB-Stick as candidates for building memory write primitives, such as `struct joydev`, or `struct mmc_ioc_cmd`, are *not* elastic (i.e., they have a fixed size). Prior work on identifying user-controlled objects that could potentially be useful for exploitation focused primarily on identifying elastic objects [5], thus missing out those that are static, even though they can also be used for exploitation.

Out-of-Bound Attacks. The exploit that abuses CVE-2022-0185 [66] leverages an OOB vulnerability on the execution path of the `fsconfig` system call, depicted in Figure 1c. Concretely, the vulnerability consists of an invalid bounds check in the function `legacy_parse_param` that leads to an *integer underflow* on the length of a string buffer sent from user-space, which then gets copied in `struct legacy_fs_context->legacy_data` via `memcpy`. This grants the exploit a large OOB on the heap. The exploit then leverages `msg_msg`'s flexible array to massage the heap and place it right after the victim object in memory. Then, the attack abuses the OOB to corrupt the `m_ts` field of the targeted `msg_msg`, which induces an OOB in a `msg_msgseg` object placed in `kmalloc-32`. There, the exploit sprays several `seq_operations` objects and calls `copy_to_user` and `copy_from_user` on the user-controlled `msg_msgseg`, bypassing KASLR and inflicting corruption on the `modprobe_path` global variable to achieve privilege escalation.

This scenario demonstrates that attackers can also abuse user-controlled objects to exploit OOB vulnerabilities. Precisely, the victim object itself, i.e., `legacy_fs_context->legacy_data`, had a user-controlled buffer overflow, which the attack pivoted to a stronger memory-write primitive by inducing a buffer overflow into the target object, i.e., `msg_msg`, which is also user-controlled. Sadly, this is a recurring pattern in exploits targeting OOB vulnerabilities on the heap [66], [67], [68], [69]. Moreover, segregating such objects in separate caches, as done by the `kmalloc-cg-*` caches would not block the attack as the exploit may massage the allocator to place the target page right next to the victim page, allowing the overflow to reach it.

5. IUBIK

`IUBIK`'s objective is to break kernel exploits by preventing them from accessing kernel-sensitive data through user-controlled objects. Such exploits can take advantage of temporal and spatial memory errors. `IUBIK` mitigates both scenarios, and it does so in a pragmatic manner, without introducing unrealistic performance or memory overheads. In realizing `IUBIK`, we overcome several challenges.

Challenge #1: Isolating Memory. There exists no mechanism currently in the kernel to enforce the strong separation between user- and kernel-controlled objects—or memory in general. Prior studies leverage hardware primitives to isolate sensitive memory [7], [8], [9], [10], [15], [70], but they either suffer from high performance and/or memory overhead, or are tailored to mitigating either spatial or temporal isolation (not both). To address these shortcomings, `IUBIK` leverages MTE to provide both spatial and temporal isolation at low performance and memory cost. Armed with MTE, we develop a novel in-kernel memory compartmentalization framework, which we describe in Section 5.1.

Challenge #2: Structure Layout. Several kernel objects (e.g., `msg_msg` or `user_key_payload`) mingle user-controlled buffers and sensitive fields, such as function and data pointers, within the same `struct` definition. This gives attackers an opportunity to abuse the former to corrupt the latter if placed in the same compartment. We thwart this risk by rewriting the objects containing user-controlled fields, effectively pulling them out, and isolating them in a separate compartment than the rest of the original object. We describe this process in Section 5.2.

Challenge #3: Identifying Allocation Sites. In order to isolate user-controlled objects, `IUBIK` must instrument their allocation sites and instruct the underlying allocators to confine them in protected regions. However, there is currently no automated way for identifying such allocation sites in the Linux kernel. Although allocation flags exist that instruct the allocator to use separate allocation caches for certain objects (e.g., `msg_msg`), they are only used sparingly, and miss objects used by existing exploits (e.g., `user_key_payload` [13]). To overcome this limitation, we develop a novel memory profiling framework for Linux that hooks several execution points in the codebase and records when user-supplied data is stored in kernel objects. We detail our approach in Section 5.3.

5.1. Isolating Memory with MTE

`IUBIK` groups all of heap memory in two domains, *DomU* and *DomK*, by tagging each physical page used by the heap allocator with one of the two corresponding, predefined MTE tags: *TagU* and *TagK*. Specifically, `IUBIK` modifies the kernel's heap allocator, `SLUB`, to tag newly allocated memory with either *TagU* or *TagK* using MTE's `STG` and `DC GVA` instructions, which tag a memory granule (16B) and a block of granules (on our platform this is 64B), respectively. With a few exceptions (addressed in Section 5.2), in `IUBIK` all objects allocated on a memory page belong to the same domain and they have the same tag. `IUBIK` uses MTE in *synchronous mode*, which provides the strongest guarantees, as it yields a fault as soon as a tag mismatch occurs during unauthorized memory accesses.

MTE's flexibility allows us to design *DomU* and *DomK* on top of the existing virtual memory layout of the kernel without modifying it, which has been a challenge in prior isolation techniques in the kernel. For example, hardware isolation primitives that are configured by bits stored in page table entries (such as `MPK` or `SMAP`) may require breaking *huge pages* (i.e., 2MB) into smaller ones (i.e., 4KB) to enforce fine-grained, page-level isolation [7], [8], [15], which may add performance overhead [71]. `IUBIK` configures `SLUB` to place new memory in *DomK* by default, unless otherwise instructed by the calling allocation site.

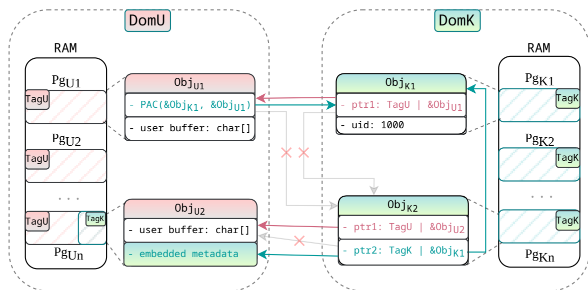


Figure 2: High-level overview of `IUBIK`'s memory layout.

`IUBIK` facilitates this by introducing a new memory allocation flag, `GFP_IUBIK_USER`, which kernel developers can use to request memory in *DomU* by passing the flag as an argument to memory allocation routines. Upon receiving this flag, `IUBIK` serves the allocation request from a new set of caches, called `kmalloc-iubik-*`, which resemble the original `kmalloc-*` caches in terms of functionality, except that they tag the underlying physical pages with *TagU* when configuring a new slab.

Accessing pages from either *DomU* or *DomK* can only go through pointers that have the matching tag, which is mandated by ARM MTE (§2.3). As such, we extend the allocator to *mask* the corresponding tag into a newly allocated address before returning it to a caller during an allocation request. This involves executing a small number of pointer arithmetic instructions that only take a couple of cycles. Upon receiving a tagged address, the caller can use it to access memory from its corresponding domain *without* having to mask-out the tag from the pointer, thanks to ARM TBI. Figure 2 illustrates a simplified view of the kernel's memory layout with `IUBIK` in place—we detail it below.

Armed with these primitives, `IUBIK` places user-controlled objects—i.e., objects used by `copy_{from, to}_user`, *etc.*—in *DomU*, while leaving the rest of kernel memory in *DomK*. Hence, both user-controlled objects and the pointers that reference them get tagged with *TagU*, which forbids access to memory tagged with *TagK*—in cases of spatial and temporal memory errors. Specifically, if attackers manage to corrupt the bounds of a *DomU* object, e.g., the user buffer of $\texttt{Obj}_{\texttt{U1}}$ in Figure 2, they are unable to reach *DomK* objects (e.g., $\texttt{Obj}_{\texttt{K1}}$); and vice versa. Similarly, if attackers gain control of a dangling pointer in *DomK*, e.g., `ptr1` in $\texttt{Obj}_{\texttt{K1}}$, they are unable to access it once its underlying memory page, i.e., $\texttt{Pg}_{\texttt{U1}}$ gets reallocated for an object in *DomK* (e.g., $\texttt{Obj}_{\texttt{K2}}$); and vice versa.

This enforcement is capable of breaking a large class of kernel exploits that rely on user-controlled objects as primitives to corrupt sensitive data. For example, attackers are not able to abuse same-cache attacks through UAF errors to, say, overlap `struct user_key_payload` onto `struct Qdisc`, as shown in Section 4, because `IUBIK` allocates the former in *DomU* and the latter in *DomK* with *TagU* and *TagK*, respectively. Moreover, `IUBIK` mitigates the OOB attack described in Section 4, as it allocates `struct msg_msg` and `msg_msgseg` in *DomU* where it cannot reach `modprobe_path`, which stays in in *DomK*. Finally, `IUBIK` also breaks cross-cache attacks, such as those described in Section 4, as accessing a dangling pointer tagged with *TagU* after its underlying pages are tagged with *TagK* will be detected by MTE. This represents a shift in how we address kernel security compared to prior work, as `IUBIK` essentially assumes that the *entirety* of kernel memory is sensitive by default, and focuses on making it inaccessible to attacker-controlled data.

5.2. Rewriting C `structs`

The MTE-based memory protection primitives described in Section 5.1 allow `IUBIK` to isolate user-controlled objects in *DomU*. However, in their original form, many of them still expose sensitive kernel fields, such as function or data pointers, which may be targeted by attackers through memory errors in *DomU*—this is permitted by our threat model (§3). As the same-cache attack described in Section 4 demonstrates, blending sensitive fields with user-controlled buffers in *DomU* allows attackers to violate `IUBIK`'s security policy and circumvent it. Unfortunately, such a `struct` layout is prevalent in the Linux kernel, as shown by our profiling data (§7.1). Moreover, our data set also reveals several kernel objects that have *embedded buffers* (or fields) that store user-controlled data, e.g., the *name* of a resource—SLUBStick (§4) uses such `structs` to manipulate page tables. Other popular exploits also abuse embedded user-controlled buffers for exploitation [69].

To overcome this issue, we rewrite these `structs`, splitting them into two objects: one that contains its user-controlled field(s) (e.g., the flexible array), and another that keeps the remaining fields stored in the original object. We allocate the former using the `GFP_IUBIK_USER` flag, instructing the allocator to store the segregated buffer on a *DomU* page and tag its address with *TagU*. Upon receiving a tagged address, we store it in the original object in a newly added field. The resulting layout is shown in Figure 2—we tag objects allocated in *DomU* with *TagU* since they only contain user-supplied data and no sensitive kernel data (i.e., function or data pointers). These objects are referenced by objects allocated in *DomK*, which are tagged with *TagK*, and contain everything else that the original kernel object did. Thus, a potential dangling pointer or OOB on objects tagged with *TagU*, e.g., `ptr1` in $\texttt{Obj}_{\texttt{K1}}$, can only access other objects in *DomU*, such as $\texttt{Obj}_{\texttt{U2}}$, which do not contain any useful fields for exploitation. This way, `IUBIK` breaks the OOB exploit in Section 4, as we pull out the user-controlled fields of `msg_msg` and `msg_msgseg`, and allocate them in *DomU*, while their sensitive fields (s.a., `m_ts`) stay with the original objects in *DomK*, where the overflow in `legacy_fs_context->legacy_data` cannot tamper with them since it is also allocated in *DomU*. Also, attackers cannot abuse the variable size of the original objects to place them in arbitrary caches, since `IUBIK` makes them static and allocates them from a single cache.

We rewrote the struct types stemming from the allocation sites recorded by our profiler during benchmarking (§7.1). Additionally, we collected popular struct types used in existing kernel exploits and rewrote them. This led us to rewrite a total of 29 structs, during which we encountered several challenges. First, we introduce a new allocation site for the separated user-controlled struct right after the allocation site of the original object and instrument it with the GFP_IUBIK_USER flag—we do not instrument the original allocation site, as it will allocate memory from *DomK* by default (§5.1). Then, for every rewritten object, we must also free its user-controlled field before freeing the original struct to avoid memory leaks. Similarly, we must also copy the user-controlled struct when the original object is copied (e.g., via memcpy or memmove). To assist us and future developers in identifying these seamlessly, we equip our dynamic tracing framework (§5.3) with the ability to record the call sites where the original object is freed and (mem-)copied.

Moreover, some subsystems rely on being able to obtain the base of an object from an inner field, e.g., by applying the container_of macro on its address. As we may pull out such embedded fields from their original struct, we need a mechanism to preserve this relationship at runtime. In IUBIK, we overcome this by maintaining a *backward reference* to the original object in the user-controlled object. To prevent attackers from corrupting the backward reference in *DomU*, we rely on ARM PA (§2.3) to preserve its integrity. Specifically, after allocating a given object in *DomK* and its extracted user-controlled buffer(s) in *DomU*, we use the PACDA instruction to sign the former and store the result in the latter. Additionally, we prevent replay attacks by using the address where we store the backward reference as context in the sign operation. Then, we replace all occurrences of container_of on the user-controlled field with a routine that authenticates and returns the backward reference using the AUTDA instruction. In case attackers tamper with the reference's PAC, the authentication will fail resulting in program termination, thanks to PA's FPAC feature. For example, the PAC stored in Obj_{U1} in Figure 2 can not be overwritten to point to Obj_{K2} unless the kernel re-signs it with the new value.

IUBIK uses the APDA key register for storing the secret authentication key, which is currently unused in the ARM kernel. Moreover, we save the key in struct task_struct on a context switch to user space and restore it upon reentering the kernel, thus avoiding malicious programs to manipulate it. Note that IUBIK allocates task_struct in *DomK*, thus preventing attackers from abusing user-controlled objects to manipulate it in the presence of a memory error. Our hardware (i.e., the Pixel 8) uses bits $[0 : 38]$ of a pointer to store the virtual addresses, while MTE uses bits $[56 : 59]$ for storing the tag, and bit 55 is used to distinguish between kernel and user addresses. IUBIK therefore stores the PAC in bits $[39 : 54]$ (16 bits). The *FPAC* feature prevents attackers from brute-forcing PAC values to guess valid signatures.

Although pulling out the flexible array field of elastic objects does not lead to major code modifications, doing so for embedded struct buffers requires adjusting some usage scenarios. Specifically, we replace all sizeof macro invocations in the redefined struct field with a modified macro that returns the size of the extracted buffer (instead of the size of a pointer; i.e., 8 bytes). We also adjust all code locations where the address of the original field is taken (i.e., &struct->field) to simply use the field (i.e., struct->field) since it is now a pointer—this frequently occurs when the embedded field is a struct. Finally, when pulling out fields that have a struct type, we replace all locations where they are dereferenced as a static struct (i.e., struct.field) with a pointer dereference (i.e., struct->field). Nevertheless, as rewriting some structs might require substantial adjustments across kernel subsystems, we also propose alternative solutions.

Indeed, our data set revealed one such instance, namely the struct skb_shared_info object, which is embedded in the user-data buffer referenced by socket buffers (i.e., struct sk_buff->data). Pulling it out of the data buffer would require significant adjustments across several networking subsystems in Linux, as many assume it is contained within. Leaving such metadata objects unprotected in *DomU* would pose a risk for IUBIK, especially since they usually contain several data pointers, function pointers, and other sensitive fields that could be targeted via memory errors in *DomU* (similar to the container_of backwards reference described above). We mitigate this risk by using MTE to tag such embedded objects with *TagK*, even though they are stored in *DomU* objects. This prevents any memory corruption on a *DomU* object from accessing it. Obj_{U2} in Figure 2 depicts this scenario. For that, we first ensure that the embedded object lies at a 16-byte aligned offset and has a size that is multiple of 16 (both required by MTE), then we tag it with *TagK* after the hosting object is allocated in *DomU*; note that Obj_{U2} is partly tagged with *TagU* and partly with *TagK* in Figure 2. Finally, before freeing the hosting object, we re-tag its embedded isolated segment with *TagU* (enabling it to be reused in *DomU*).

This way, IUBIK provides two alternative methods for protecting the remaining sensitive data stored in *DomU* when rewriting is not possible. First, developers may want to use MTE to protect chunks that are larger than 8 bytes (s.a., skb_shared_info), are longer-lived, and contain a multitude of pointers, since frequently (re-)tagging on (de)allocation might add performance overhead for short-lived objects. Second, developers may want to use PAC for smaller, short-lived objects (s.a., the 8-byte backward reference in container_of) since the performance penalty is spent at access time instead of (de)allocation time.

5.3. Usercopy Profiling

IUBIK serves objects isolated in *DomU* to allocation sites instrumented with the GFP_IUBIK_USER flag. To systematically identify such sites at scale in the kernel, IUBIK includes a dynamic profiling framework, dubbed the

```c
SYSCALL_DEFINE5(add_key, ..., const void __user *,
    _payload, size_t, plen, ...) {
  void *payload;
  payload = kvmalloc(plen, GFP_KERNEL); ①
  copy_from_user(payload, _payload, plen); ②
  key_create_or_update(..., payload, plen, ...);
}

int user_preparse(
    struct key_preparsed_payload *prep) {
  struct user_key_payload *upayload;
  size_t datalen = prep->datalen;
  upayload = kmalloc(sizeof(*upayload) + datalen, ③
                     GFP_KERNEL);
  memcpy(upayload->data, prep->data, datalen); ④
}

void user_destroy(struct key *key) {
  struct user_key_payload *upayload =
    key->payload.data[0];
  kfree_sensitive(upayload); ⑤
}
```

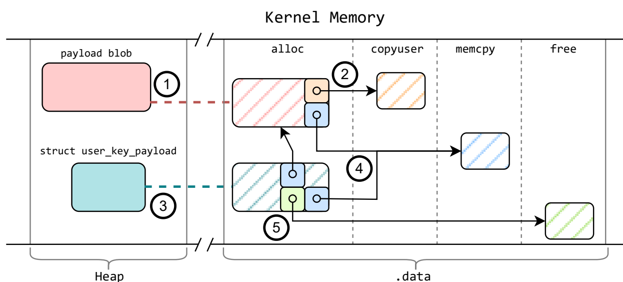


Figure 3: High-level overview of the usercopy profiler.

usercopy profiler, which is able to track memory objects beginning from where the kernel copies data from user space. The usercopy profiler is built atop Linux's recently introduced *memory allocation profiling* feature (§2.2) to mark the allocation sites of objects that are accessed from user space. An overview of how the usercopy profiler is able to record complex allocation patterns is shown in Figure 3.

Initially, at compile-time, metadata tag structures—called *codetags* and associated with allocation sites, user copying sites, `memcpy` sites, and `free` sites—are added to the kernel's `.data` section (hatched boxes in Figure 3). These tags contain the file, function, and line number of the associated allocation site as well as other information populated at run-time when certain events occur; e.g., an object is allocated or data is copied from one object to another. Given the listing in Figure 3—which highlights the routines from Linux's key retention service [72] that facilitate using `struct user_key_payload` as an exploitation primitive (§4)—the usercopy profiler will create a codetag for the allocsite on ln. 4 and ln. 12 (i.e., alloctags), the usercopy site on ln. 5 (i.e., a copyuser tag), the `memcpy` site on ln. 14 (i.e., a `memcpy` tag), and the `free` site on ln. 20 (i.e., a `free` tag). Retrieving a given codetag is possible using the address of an allocated object, shown by the link marked with ① and ③ connecting an object and its corresponding codetag. After a given workload has completed running, the usercopy profiler can be queried via `/proc/allocinfo` for the information collected. To mark object sites, the usercopy profiler hooks the routines used by kernel developers to transfer data to and from user space and adds functionality to retrieve and mark their respective codetags (ln. 5; ②).

Linux facilitates these routines via the *usercopy* subsystem, which exposes an API that other subsystems can use to interact with user programs. At its core lies `copy_{from,to}_user`, which are by far the most widely used routines that the API provides. The usercopy profiler hooks the calls to these routines, as well as to their derivatives, and marks the codetag of the kernel object that is given as an argument during their invocation—for `copy_from_user` this is the destination object where user data is copied *to*, while for `copy_to_user` this is the source object where kernel data is copied *from*. Specifically, we instrument the following usercopy routines: `do_{get,put}_user_call`, `raw_copy_{from,to}_user`, and `strncpy_from_user`. (All the other usercopy derivatives will eventually call one of these.)

Importantly, there are cases where user data *escapes* the original object it was copied into, propagating to other objects allocated on the heap. For example, in our listing from Figure 3, the contents of `struct user_key_payload` are first stored in `struct key_preparsed_payload->data` via `copy_from_user`, and then copied in `user_key_payload` via `memcpy` (ln. 14). The usercopy profiler handles such cases by hooking the functions that perform *buffered data copies* across heap objects: i.e., `memcpy`, `memmove`, `strcpy`, and `strncpy`. When these execute during profiling, we mark the destination pointer's codetag if the source's codetag is already marked as copied *from* user, and the source pointer's codetag if the destination's codetag was copied *to* user (④). The usercopy profiler also keeps track of the *offsets* where user data gets copied in kernel objects, and only marks the `memcpy`'ed objects if the respective addresses and sizes fall within these offsets. (This avoids introducing false positives.)

The usercopy profiler also maintains an alloctag similar to `SLUB`'s allocation routines for several *alloc wrappers* that we identified throughout the kernel codebase. This reduces the number of false positives in our dataset since we can apply finer-grained instrumentation to wrappers' call sites that allocate user-controlled objects, instead of allocation sites within the wrappers, which would serve isolated memory for all invocations. (For a list of hooked alloc wrappers refer to Table 4 in Appendix A.) We also design the profiling framework to assist developers with rewriting `structs` that contain fixed-size buffers (indirectly) accessible from user space and isolating them with `IUBIK`. For that, we extend the usercopy profiler to maintain a list of *free sites* (⑤) and *`memcpy` sites* (④) for each allocsite, allowing developers to efficiently discover locations where calls to `free` and `memcpy` are needed when rewriting C `structs` (§5.2).

In summary, the usercopy profiler records the following information for every allocsite:

- *Flags*: Whether the allocsite is instrumented with a kernel flag that signals a user allocation—i.e., GFP_-KERNEL_ACCOUNT. This flag could be used to identify other sites that may be touched by user data. However, we find that many sites recorded by our profiler were *not* instrumented with this flag.
- *Cache Type*: Whether the allocsite uses a generic or dedicated cache; i.e., kmalloc or kmem_cache. kmalloc sites should have a higher instrumentation priority as they facilitate type confusion—generally more useful to attackers (§4).
- *Privilege*: Whether the usercopy was performed during the execution of a privileged or non-privileged task. Instrumenting non-privileged tasks should have a higher priority; however, there is no guarantee that non-privileged attackers cannot execute the others.
- *Offset*: The offsets within the object where user data was copied. This hints if the struct type needs to be rewritten and which field should be extracted.
- *Copy Information*: Whether data was copied *to* or *from* user space and the allocsites of objects where data was memcpy'ed to and from (if applicable).
- *Object Size Type*: Whether the allocsite used a constant object size or a variable object size.
- *Syscalls*: The system calls that executed the allocsite.
- *Usercopies*: The usercopy call sites where this allocsite was accessed; the full list of usercopy routines that we track is available in Table 5 of Appendix A.

6. Implementation

We implemented IUBIK's usercopy profiler atop Linux kernel v6.10, which supports the recent *memory allocation profiling* feature (§2.2) required by our framework. Then, we ran a wide range of workloads that gave us a base dataset with allocation sites that IUBIK must instrument to harden the OS (§7.1). Our profiler prototype consists of ≈1350 added and ≈250 removed lines of C and ASM code. Next, we implemented IUBIK's hardening atop Linux v5.15, the latest kernel supported by Android for Pixel 8. Note that most allocsites recorded on v6.10 also exist on v5.15. Our IUBIK prototype consists of ≈1050 added and ≈520 removed lines of C and ASM code.

For the case study presented in this paper, we instrumented the allocsites recorded by our profiler during boot as well as during the tests from LMbench [73] and the Phoronix Test Suite [74], which we use to evaluate IUBIK's performance overhead (§7.2). In total we instrumented 79 allocsites. Out of these, 23 involved rewriting the respective struct types that they allocate and pulling out the user-controlled buffer in a new struct definition, which we isolate from the rest of the object. Among the 23 rewritten types, we encountered 1 whose extracted field was being used in container_of calls. Therefore, we had to add a *backward reference* in its new struct and preserve its integrity with ARM PA.

Additionally, we instrumented allocation sites and rewrote the structs that were previously abused in kernel exploitation—some of them quite frequently—but were not triggered by our performance benchmarks. Specifically, we hardened msg_msg, msg_msgseg, nft_set, nft_userdata, tipc_aead_key, and sixpack. The whole rewriting changeset consisted of ≈750 added and ≈360 deleted lines of C code. We list all of the struct types rewritten, along with additional details (e.g., whether they include flexible arrays), in Table 6 of Appendix A. Finally, to avoid producing a large changeset, we tagged sk_buff->data->skb_shared_info with *TagK* instead of pulling it out in *DomK* (§5.2).

7. Evaluation

We first obtained a base dataset with allocsites that IUBIK must instrument by profiling a wide array of workloads on Linux kernel v6.10, which supports the *memory allocation profiling* changeset required by our framework (§2.2). We carried out our experiments on a host equipped with a 128-core AMD EPYC 7551 CPU (2 sockets, 32 cores/socket), 8 NUMA nodes, and 128GB DDR4 RAM. We fixed the CPU frequency to 2.0GHz, disabled dynamic voltage and frequency scaling, and minimized background tasks to allow for reproducible performance results that can be compared with an uninstrumented kernel to determine the overhead that profiling adds to a workload.

Then, we evaluated IUBIK's runtime and memory overhead while hardening SLUB on Linux kernel v5.15, the latest kernel supported by Android 14 for Pixel 8 [75]. Fortunately, the allocsites recorded during our profiling on v6.10 also exist on v5.15. We conducted our experiments on bare metal in a Debian-like *chroot* environment (configured with debootstrap [76]) on a rooted Pixel 8. Our device ships with Google's Tensor G3 SoC, and is equipped with 8GB LPDDR5X RAM, 9 CPU cores (1x 2.91GHz Cortex-X3, 4x 2.37GHz Cortex-A715, 4x 1.70GHz Cortex-A510), and 128GB storage. To reduce noise across measurements, we configured the CPUs on a fixed frequency, and pinned our benchmarks on a subset of the CPUs—we describe our configuration in more detail for each particular benchmark.

We also assess IUBIK's effectiveness by testing its hardening primitives against a suite of synthetic test cases, and by deploying 1 and surveying 31 real-world kernel exploits against IUBIK.

7.1. Usercopy Profiling

We conducted several experiments with IUBIK's *usercopy profiler* to dynamically identify heap allocsites that may be user-controlled. We drove our dynamic analysis via a rich set of programs that trigger user–kernel interactions, allowing us to reveal such locations. We included tests from *LMbench* [73], the *Phoronix Test Suite (PTS)* [74], the *nftables* framework [77], *Kselftests* [78], and the *Linux Test Project (LTP)* [79]. We describe our results in Table 1.

Table 1: Memory allocation profiling results. 'S' and 'U' stand for privileged and non-privileged, respectively.

Benchmark		Allocsites									Usercopies		
		Qty.	`kmem_cache`	`kmalloc`	Flagged	Fixed	Flexible	From User	To User	`memcpy`	Qty.	Heap	Stack
LTP	S	130	5 (4%)	125 (96%)	8 (6%)	41 (32%)	27 (21%)	66 (51%)	90 (69%)	99 (76%)	266	49 (18%)	221 (83%)
	U	105	17 (16%)	88 (84%)	39 (37%)	35 (33%)	18 (17%)	73 (70%)	81 (77%)	88 (84%)	198	40 (20%)	167 (84%)
Kselftests	S	170	9 (5%)	161 (95%)	9 (5%)	37 (22%)	24 (14%)	114 (67%)	86 (51%)	144 (85%)	146	34 (23%)	112 (77%)
	U	95	18 (19%)	77 (81%)	24 (25%)	47 (49%)	15 (16%)	53 (56%)	77 (81%)	79 (83%)	265	39 (15%)	234 (88%)
LMbench	S	53	3 (6%)	50 (94%)	5 (9%)	19 (36%)	8 (15%)	21 (40%)	39 (74%)	38 (72%)	73	20 (27%)	53 (73%)
	U	42	9 (21%)	33 (79%)	18 (43%)	21 (50%)	8 (19%)	20 (48%)	36 (86%)	30 (71%)	136	24 (18%)	114 (84%)
PTS	S	52	4 (8%)	48 (92%)	5 (10%)	20 (38%)	8 (15%)	20 (38%)	39 (75%)	39 (75%)	69	18 (26%)	51 (74%)
	U	47	10 (21%)	37 (79%)	18 (38%)	22 (47%)	8 (17%)	21 (45%)	40 (85%)	33 (70%)	147	26 (18%)	123 (84%)
nftables	S	47	4 (9%)	43 (91%)	4 (9%)	19 (40%)	8 (17%)	17 (36%)	35 (74%)	35 (74%)	66	16 (24%)	50 (76%)
	U	87	14 (16%)	73 (84%)	36 (41%)	30 (34%)	15 (17%)	57 (66%)	66 (76%)	72 (83%)	154	30 (19%)	127 (82%)
Total	S	292	12 (4%)	280 (96%)	16 (5%)	68 (23%)	50 (17%)	183 (63%)	163 (56%)	237 (81%)	351	79 (22%)	276 (79%)
	U	212	30 (14%)	182 (86%)	64 (30%)	82 (39%)	31 (15%)	131 (62%)	167 (79%)	168 (79%)	341	82 (24%)	273 (80%)

Notably, the dataset for each collection of tests was recorded independently of the others—i.e., the test machine was rebooted after running each test set, resetting `/proc/{alloc,copy}info`. 'Total' is the union of the results for each individual test suite. On an uninstrumented, baseline kernel, the full set of tests took 12h35m to run, while on a kernel instrumented via the usercopy profiler, the tests took 12h59m to run—the overall overhead of the usercopy profiler is 3.12%. Future users may disable individual workloads used in our experiments, or add new ones (e.g., via *syzkaller*), depending on their time budget.

Allocsites. The usercopy profiler recorded a total of 292 privileged allocsites and 212 non-privileged allocsites during the executed test sets, out of a total of 6902 instrumented allocsites, with LTP and Kselftests contributing most significantly. Out of the total privileged allocsites recorded, 129 (44%) and 109 (37%) were accessed during a `copy_{from,to}_user` operation (or a derivative), while 54 (18%) were accessed by both. For non-privileged allocsites, 45 (21%) and 81 (38%) were accessed during a `copy_{from,to}_user` operation (or a derivative), while 86 (41%) were accessed by both. 95% of privileged and 86% of non-privileged allocsites were *not* instrumented by the original kernel codebase with a usercopy flag (column: 'Flagged')—i.e., `GFP_KERNEL_ACCOUNT`—, thus making them available for exploitation. Additionally, 77% of privileged and 61% of non-privileged allocsites generated heap objects that were accessed by a usercopy routine at an offset larger than 0 (column: 'Flexible'), hinting that these likely contain an embedded or flexible array. Moreover, 81% of privileged and 79% of non-privileged allocsites propagated usercopy data through `memcpy` or its derivatives (column: '`memcpy`'), highlighting the importance of tracking such cases. Finally, 23% of privileged allocsites were part of *out-of-tree* kernel modules, while the other 77% were part of the core kernel. Non-privileged allocsites followed a similar pattern, with 20% being part of out-of-tree kernel modules, while 80% were part of the core kernel.

Usercopies. The usercopy profiler recorded a total of 351 privileged usercopy sites and 341 non-privileged usercopy sites in the analyzed codebases—these numbers represent the number of instrumented usercopy sites that used either the heap *or* the stack *at runtime* out of 3394 total usercopy sites we instrumented. Out of these, 79 (22%) privileged sites and 82 (24%) non-privileged sites accessed heap objects during our profiling. After examining the respective code, we noticed that many usercopy sites are part of generic copy wrappers, s.a., `memdup_user`, which are widely used throughout kernel subsystems; thus, many usercopy sites get reused across the kernel's subsystems. Our experiments also revealed that 79% of recorded, privileged copysites and 80% of recorded, non-privileged copysites transferred data to and from user-space on the stack. However, as per our threat model (§3), we only focus on heap allocsites in this paper, as they are typically used in existing exploits (§4)—we plan on further investigating this finding in the future. Finally, out of a total of 351 recorded privileged sites, 116 (33%) were part of *out-of-tree* kernel modules, while the other 235 (67%) were part of the core kernel. In contrast, out of a total of 341 recorded non-privileged sites, the majority (96%) were part of the core kernel, with only 14 (4%) being part of out-of-tree kernel modules.

Result Validity. We ensured the validity of our results via a combination of manual analysis and functional testing. We first confirmed that the profiler did not record any false-positive call sites (i.e., alloc, user copy, `memcpy`, `free`) during our performance benchmarks (§7.2) by manually vetting the recorded data set against the kernel codebase; manually vetting the entire profiling data set could be done with additional effort, but is outside this paper's scope. Second, to prevent false negatives (i.e., missing call sites of interest) we wrote functional tests that capture the complex allocation and user-copy patterns described in Section 5.3, and confirmed that the profiler records them. We elaborate on how to improve the precision of our profiler in Section 8.

7.2. Runtime Overhead

To evaluate the runtime overhead of our prototype, we deploy a set of micro- and macro-benchmarks that execute a wide array of single-threaded and multi-threaded workloads.

7.2.1. LMbench. We deployed the LMbench [73] microbenchmark to evaluate `IUBIK` on tests that stress individual components of the underlying kernel, such as *socket* and *file* operations, which also trigger extensive usercopy interactions. To reduce noise, we pinned the tests on CPUs 4–7 and set their frequency to their max value of 2.37GHz.

Table 2: Performance results of IUBIK vs. vanilla SLUB.

	Overhead	Benchmark
LMbench	≈0%	syscall, read, write, select (500 fds), select (10 fds), select (500 tcp fds), select (10 tcp fds), sigaction, sig deliver, pipe, unix socket, tcp socket, udp socket
	2%	fork+"/bin/sh"
	3%	prot fault, fork+execve
	4%	stat, fork+exit
	7%	open/close
	8%	fstat
PTS	≈0%	unpack-linux, compile-linux, ffmpeg, openssl (sign), openssl (verify), nginx, sqlite, redis
	2%	hackbench

We also pinned the server programs for lat_udp and lat_tcp on CPUs 0–3 and ran them at 1.70GHz. Figure 2 shows that IUBIK incurs worst-case slowdowns of 8% on fstat and 7% on open/close. However, IUBIK incurs either negligible ($< 5\%$) or no slowdown in all the other tests. We investigated the root cause of the higher overhead on fstat and stat, and determined it is the result of struct filename->iname, which we hardened in IUBIK. Specifically, we pulled out its $4KB$ iname buffer and isolated it in *DomU* while keeping the rest of struct filename in *DomK*. This causes interference in the data and TLB caches which translates to more cycles, especially since the affected tests mainly involve allocating, accessing, and de-allocating this struct in the kernel.

7.2.2. Phoronix Test Suite. LMbench is composed of synthetic stress tests that may not fully capture the performance of IUBIK on real-world, end-to-end workloads. For example, LMbench only uses a fraction of available CPU cores on scheduler stress tests. Thus, we ran further experiments using macro-benchmarks from the Phoronix Test Suite (PTS) [74] that also trigger usercopy interactions in IUBIK. We pinned the tests on CPUs 0–3 and set their frequency to a lower value of 1.43GHz, to prevent the CPU's temperature sensors lowering it while benchmarking. We also pinned the server program for hackbench, nginx, apache, and redis on CPUs 4–7 and set their frequency to 1.42GHz. Table 2 shows the results of IUBIK versus the baseline, unmodified SLUB. IUBIK incurs negligible slowdown ($< 3\%$) in all tests.

7.2.3. System V Message Queues. System V Message Queues are an inter-process communication mechanism available in the Linux kernel and implemented via the struct msg_msg and struct msg_msgseg objects. In IUBIK, we extracted their flexible array fields and isolated them in *DomU*. Hence, we wrote a custom benchmark to measure the performance impact for doing so. We measured the latency of the kernel's load_msg function (by subtracting the values returned by ktime_get at the end and at the start), which is responsible for allocating memory for the IPC message and copying it from user space.

In unmodified Linux, msg_msg stores the first part of the message up to the size of a page (i.e., 4KB−sizeof(msg_msg)), while the rest of the message is stored in msg_msgseg objects. In IUBIK, we pull out the user data from msg_msg and msg_msgseg and isolate it in *DomU*. This has the advantage of using fewer msg_msgseg objects for messages that cross a page boundary. For example, a 4KB message would normally use one msg_msg and one msg_msgseg to fit the whole user data, while IUBIK requires only the msg_msg and the separated page for storing the user data. However, the number of accessed pages is the same in both cases, i.e., 2.

To test the performance impact of these, we spawned a client and a server and configured them to send 50 messages of 2048, 4096, and 8192 bytes when IUBIK was active and not active. We pinned the client on CPUs 0–3 @ 1.43GHz and the server on CPUs 4–7 @ 1.42GHz. No significant difference in the latency of load_msg was observed.

7.3. Memory Overhead

IUBIK configures a new set of kmalloc caches to store the isolated, user-controlled objects allocated from instrumented allocsites. For this prototype we also configured a new dedicated cache (i.e., kmem_cache) to allocate the user-controlled 4KB-wide field, iname, which we extracted from struct filename. These caches slightly increase memory consumption with added metadata structures. Also, for objects that store a user-controlled part immediately after their body, such as struct msg_msg and struct msg_msgseg, we add a new pointer that references the now-isolated, user-controlled buffer, increasing the size of such objects only by 8 bytes. We found that such occurrences are rare in the kernel. Finally, we require 8 bytes to store the PAC-protected backwards reference in the isolated, user-controlled object, which IUBIK needs for replacing container_of instances that reference the parent object—again, such occurrences were rare in our prototype.

To evaluate the impact of these on the system's memory consumption we measured the maximum *resident size set (RSS)* incurred by IUBIK during *boot* and compare it with the unmodified SLUB's. This experiment should be conclusive enough, as the kernel exercises a large number (87) of user-controlled allocsites during boot. IUBIK reached a maximum of 2.17% more in-use memory than SLUB during boot, amounting to ≈ 7MB.

7.4. Security Evaluation

We demonstrate IUBIK's security effectiveness by: (1) subjecting it to a suite of test cases that validate its security guarantees, (2) deploying it against one real-world exploit, and (3) surveying several other exploits that leverage user-controlled primitives for exploitation.

7.4.1. Security Testing. We wrote a kernel module that includes a suite of test-cases that validate IUBIK's security claims (§5.1) and deployed it against IUBIK and SLUB.

Specifically, we defined an elastic struct that contains a user-controlled flexible array field and its length, as well as a 64-byte struct that contains an embedded secret buffer and a function pointer—we call these structs the *primitive* and *target* structs. We then used them to craft attack scenarios (below) that imitate real-world exploits.

Same-cache attacks. First, we simulated same-cache attacks, by requesting a 64-byte chunk from SLUB via kmalloc to store the primitive struct, which we freed while keeping a dangling pointer to it. Note that the flexible array field could have allowed us to request memory for *any* other size. We then called kmalloc again to get memory for the target struct. This returned the same chunk pointed to by our dangling pointer, which allowed us to leak the secret buffer and overwrite the function pointer. Under IUBIK, we first rewrote the elastic struct and extracted its flexible array field. Therefore, by using the GFP_IUBIK flag when requesting memory from kmalloc, the user-controlled chunk was allocated from the *DomU* caches, while the header struct stayed in *DomK*. After freeing the primitive object and allocating the target, the dangling pointer to the (freed) primitive object could no longer leak or corrupt the target object's contents anymore.

Cross-cache attacks. Next, we simulated cross-cache attacks by allocating the primitive object and the target object from two different dedicated kmem_cache caches. After allocating the primitive, we kept a dangling pointer to it, and freed all objects in the slab until the Buddy allocator reclaimed the affected page. Then, we immediately allocated the target object, which used the freshly freed page from Buddy—the same page that our dangling pointer refers to. Under SLUB, we were able to access the target and manipulate its contents. Under IUBIK, the user-controlled field of the primitive gets allocated from *DomU* and the dangling pointer is tagged with *TagU*, while the target is allocated from *DomK* and its underlying memory is tagged with *TagK*. When we tried to access the target through the primitive's dangling pointer the CPU generated a fault, as the MTE tags of the dangling pointer (*TagU*) and the target's memory (*TagK*) did not match.

OOB attacks. We also crafted an OOB scenario by allocating the primitive and the target from two different, but consecutive, pages in memory. We then induced an overflow on the primitive object that spanned the neighboring target object, allowing us to access and manipulate the target object under SLUB. After applying IUBIK, this attack was no longer possible as we allocated the primitive object from *DomU* and the target object from *DomK*; therefore, our attempt to access the target was intercepted by the CPU due to the mismatched MTE tags. Notably, we also generated a backward reference in the extracted, user-controlled struct to its primitive object and signed it with ARM PA. Then, to test the effectiveness of this signing, we used the OOB scenario to corrupt the backwards reference and overwrite it with an arbitrary pointer. Authenticating the tampered pointer resulted in a crash because we could not guess the PAC; in contrast, we were able to authenticate the untampered pointer successfully.

7.4.2. Real-World Exploits. We ported an exploit that targets CVE-2022-32250 [13] on Linux kernel v5.15 with Ubuntu 22.04 to the Android kernel running on the Pixel 8 and deployed it against IUBIK. The attack abuses the *Netfilter* subsystem, which introduces a UAF on an nft_expr object with nft_lookup subtype, by omitting to cleanup its pointer from the binding linked-list of expressions stored in an nft_set object. This allows attackers to insert new nft_expr objects in the binding list and corrupt the freed slot of the victim nft_expr. The exploit abuses this by first allocating a user_key_payload object onto the freed nft_expr, and inserting a new nft_expr in the binding list, which overwrites the next field of the freed nft_expr with the address of the inserted one. However, as this falls exactly over the data buffer of the overlapping user_key_payload, attackers can exfiltrate it by reading it from user-space via copy_to_user, which reveals the address location of the heap.

Next, the exploit aims to bypass KASLR, and for that it first overlaps the freed nft_expr with a posix_msg_tree_node, which stores the head of a list of msg_msg objects. Then it adds a new nft_expr, which overwrites the next pointer of the msg_msg list with the address of the new nft_expr, thus interpreting it as an msg_msg. It then sprays several user_key_payload objects, where it builds fake msg_msg objects with custom input supplied via copy_from_user, aiming to overlap their rcu_head->callback function pointer onto the user data portion from the msg_msg. This allows the exploit to leak the callback function pointer to user-space via copy_to_user, effectively defeating KASLR. Next, the exploit computes the address of modprobe_path and overwrites its contents by unlinking a fake msg_msg object, which inserts it into the list and overwrites its string.

While the exploit was successful against the unmodified SLUB, IUBIK swiftly broke it in its first stage. Specifically, as user_key_payload is rewritten to allocate its user-controlled part in *DomU*, attackers can no longer use its variable size to target arbitrary objects (such as nft_expr), neither can they overlap with sensitive fields to read or write them via copy_{from,to}_user. IUBIK also breaks the subsequent stages too, as msg_msg is similarly rewritten to allocate its user-controlled portion in *DomU*.

7.4.3. Effectiveness Survey. We systematically analyzed 31 known exploits published in the past 5 years against temporal or spatial memory corruption CVEs in Linux and identified 28 (90.3%) that IUBIK is able to mitigate (see Table 7 in Appendix A). They all leverage user-controlled objects to leak and manipulate critical fields from kernel objects, which IUBIK isolates in two distinct MTE domains, i.e., *DomU* and *DomK*, where MTE's hardware tags prevents them from accessing each-other. Out of the collected samples, 21 of them abuse a temporal memory error (i.e., UAF or DF), while 7 target spatial memory errors (i.e., OOB). In addition, IUBIK is able to neutralize the memory write primitives used by SLUBStick [1] in their published exploits (see Table 7). They all rely on reclaiming memory pages

with a dangling reference to a user-controlled object as pages for page tables (to manipulate them). Under IUBIK, the former is tagged with *TagU*, while its underlying page is tagged with *TagK* when it gets reclaimed as a page table, making it inaccessible upon accessing the dangling pointer.

Moreover, most of the surveyed exploits leveraged the *flexible array* field of an *elastic object* to *massage* the heap into allocating a target object in a cache of their choosing, next to or on top of a victim object they control, aiming to corrupt its *header* (i.e., the fields in front of the flexible array). IUBIK renders this ineffective as it pulls the flexible array field out of such elastic kernel objects, naturally transforming them into fixed objects with a static length, prompting them to be allocated from the same cache every time. Furthermore, one exploit [69] abused an OOB in an embedded array, which becomes ineffective in IUBIK as we also pull out embedded arrays from such structs.

We encountered three exploits in our survey that IUBIK could not mitigate currently. First, one exploit targets CVE-2022-20409 [80] using the DirtyCred technique [4], which does not rely on user-controlled memory to build exploitation primitives. To thwart DirtyCred-like attacks, the defense proposed by the DirtyCred authors—i.e., splitting privileged and non-privileged allocations in different caches—could seamlessly be integrated into IUBIK's architecture. Second, one exploit targets CVE-2021-4154 [81] using the DirtyPage technique [3], which currently bypasses IUBIK by reclaiming freed slab pages (similarly to cross-cache attacks) as user-controlled pages allocated via Buddy. To circumvent DirtyPage-like attacks, a future IUBIK prototype would simply have to instrument the alloc sites that request user-controlled memory from Buddy, such as the alloc_one_pg_vec_page call in alloc_pg_vec, with the GFP_IUBIK_USER flag (§5.1). This results in isolating the page in *DomU* and preventing potentially dangling *DomK* objects from ever accessing it. Moreover, our usercopy profiler could be easily extended to track Buddy alloc sites that allocate pages accessed from user space. Finally, CVE-2022-38181 [82] abuses a DF vulnerability in SLUB to obtain a UAF on pages allocated by Buddy, which is outside the scope of our threat model (§3). IUBIK can be extended to harden Buddy similarly to SLUB.

8. Discussion and Future Work

Porting IUBIK in User Space. IUBIK's mechanism may also be adopted by user programs (e.g., web servers, browsers, cryptographic libraries) to shift efforts from protecting sensitive data—the current standard practice [10], [83], [84]—toward disrupting exploitation by isolating attacker-controlled input. For example, the OpenSSL object ssl3_record_st [85] suffered from an OOB on the heap, which led to the Heartbleed vulnerability [86]. It allowed attackers to exfiltrate sensitive user data (e.g., cryptographic material), as it was allocated from the same memory pools as the sensitive data, and later memcpy'ed into the response buffer sent back to the requester [87].

Conceptually, IUBIK could have prevented Heartbleed, since it would have isolated ssl3_record_st in *DomU*, where it could not overflow into sensitive objects from *DomK*. Moreover, several other attacks against web browsers and servers rely on attacker-controlled memory—e.g., read from or written to sockets—to corrupt or exfiltrate sensitive data [88], [89].

Alternative Hardware Extensions. Although in this prototype we use *MTE* to enforce strong temporal and spatial isolation between *DomU* and *DomK*, IUBIK is generally compatible with other hardware extensions that can isolate memory within the kernel's address space. For example, IUBIK may adopt the techniques we proposed recently [15] to assign a different *aliasing domain* to *DomU* and *DomK*, isolating them both spatially and temporally via Intel *Memory Protection Keys (MPK)* [90]. This could also be achieved via the hardware extensions used in prior memory isolation works, shown in Table 3, or via virtual address pinning, a technique proposed in SLAB_VIRTUAL that relies on page tables to combat cross-cache attacks. Nevertheless, any hardware primitive based on explicitly switching isolation domains, such as Intel MPK (via WRPKRU), requires instrumenting code that legitimately needs to access user-controlled objects in *DomU* and switching the domain's permissions—something that may incur additional overhead.

Identifying Additional Alloc Sites. Our profiler may increase its code coverage and identify more user-controlled alloc sites by integrating *kernel fuzzers* in its workload, such as *Syzkaller* [91]. Furthermore, in this prototype our profiler does not track user-controlled bytes that are first copied on the stack and then moved to heap objects, however it may be extended to do so. Besides automated approaches, identifying additional allocsites could also be done manually by verifying those that are already instrumented with the GFP_KERNEL_ACCOUNT, as the presence of this flag already indicates that they may allocate user-controlled objects. Another key advantage of IUBIK is that as soon as a kernel object is spotted in an exploit, its alloc site can be seamlessly instrumented, thus isolating the object in *DomU* and disrupting any further attempt to abuse it.

Limitations and Weaknesses. We identify the following weaknesses that could be used to subvert IUBIK, although we have not encountered them in any exploit in our survey. As demonstrated in previous research [92], attackers could reclaim victim slab pages in user-space, allowing them to bypass IUBIK and manipulate their contents. Nevertheless, such attacks are currently deemed impractical [3] due to the kernel/user zone separation in Buddy, which prevents kernel pages from being reallocated in user-space (except under memory pressure). Additionally, existing mitigations that prevent kernel memory from being accessible while allocated in user-space, such as XPFO [47], could be integrated into IUBIK. Attackers may also leverage temporal memory errors to overlap *DomU* pointers with *DomK* pointers, or spatial errors to corrupt them. Moreover, as IUBIK does not provide any further separation in *DomU* user data, based on privilege or ownership, attackers could corrupt privileged objects with non-privileged data.

Table 3: Comparison between IUBIK and prior hardware-based isolation techniques, sorted chronologically.

Name	Hardware Primitives	Isolation Policy	Kernel/User Space
IUBIK	**MTE+PAC**	**User-Controlled Data**	**Kernel**
PeTAL	MTE+PAC	Security-Critical Data	Kernel
Safeslab	MPK	Freed Heap Pages	Kernel
ISLAB	SMAP	Security-Critical Metadata	Kernel
PANIC	PAN	Untrusted Program Components Security-Critical Data	User
DOPE	MPK	Security-Critical Data	Kernel
HACK	MTE+PAC	Kernel Device Drivers	Kernel
Cerberus	MPK	Untrusted Program Components	User
Jenny	MPK	Untrusted Program Components	User
CETIS	CET	Untrusted Program Components Security-Critical Data	User
xMP	VT-x	Security-Critical Data	Kernel User

A future IUBIK prototype could address this issue by maintaining them in privileged *DomU* domains and protecting them with different MTE tags. Finally, if attackers obtain a write primitive on sensitive objects (e.g., `struct cred`) then IUBIK falls short. Nevertheless, we believe that obtaining such a primitive without the use of usercopy primitives will be challenging.

9. Related Work

Table 3 provides a high-level comparison between IUBIK and other hardware-based memory isolation techniques in user and kernel space. Notably, IUBIK is the first to propose mitigating exploits by isolating user-controlled data in shadow memory. We provide a more details below.

Protecting Memory with MTE and PA. PeTAL [9] isolates sensitive kernel objects, such as process credentials, with MTE and protects the pointers that reference them with PAC, as it assume attackers can target them with arbitrary read/write primitives. However, in order prevent pointers to non-sensitive objects from referencing sensitive ones, PeTAL must instrument all non-sensitive memory accesses and enforce tag 0 on the accessed pointer, which impacts the system's performance. Moreover, PeTAL uses PA to sign and authenticate all references to sensitive objects, which also impacts performance. HACK [70] leverages MTE and PAC to compartmentalize potentially-compromised kernel drivers into their own sandbox, which also requires instrumenting all memory accesses within the sandbox to authenticate the accessed pointers, thus slowing down the driver's execution. In contrast, as we focus on isolating user-controlled objects, which are the standard primitives to corrupt sensitive objects and their references, we can relax the attacker model in IUBIK, and only protect with PAC the few backward pointers that we keep in *DomU*, while leaving all the other memory accesses uninstrumented. This leads to a low performance impact.

Other Hardware Extensions for Memory Isolation. Intel MPK (*PKS* or *PKU*) was used by Safeslab [15] to mitigate temporal errors in OS kernel heap allocators, by DOPE [8] to isolate security-sensitive kernel data, including process credentials and page tables, and by Cerberus [93] and Jenny [94] to sandbox untrusted components in user programs. ISLAB [7] and PANIC [95] repurposed SMAP and PAN, two equivalent technologies on x86 and ARM CPUs for preventing `ret2usr` attacks [34], to isolate security-critical memory in kernel and user space, respectively. CETIS [96] repurposed Intel CET, an extension meant to assist implementing efficient CFI, to isolate sensitive user data. xMP [10] repurposed Intel's hardware virtualization technology VT-x to isolate sensitive data both in kernel and user space. While these techniques significantly raise the bar for kernel and user attackers, it is currently uncertain what other data/objects could be targeted by exploits. In contrast, IUBIK targets user-controlled data to mitigate exploits, which is easier to identify at scale. Moreover, such extensions require instrumenting code to enable/disable access to the isolated domain, which causes high performance overhead in some techniques. In contrast, thanks to ARM MTE and PAC, IUBIK's domain metadata is carried along with pointers as they are moved around, sparing us from having to add domain-switching instrumentation.

Investigating User-Controlled Objects. ELOISE [5] also studied the importance of user-controlled data in facilitating kernel exploits. However, there are several key differences between IUBIK and ELOISE. First, ELOISE focuses on kernel objects that can be leveraged to *leak* kernel data (via `copy_to_user`), whereas we also focus on objects that attackers may use to *corrupt* kernel-sensitive data (via `copy_from_user`), which is far more crucial to building exploit primitives (§4). Second, the proposed defence of ELOISE is similar to the `GFP_KERNEL_ACCOUNT` flag in Linux, which existing exploits can bypass, e.g., via cross-cache attacks (§4). In IUBIK, we not only allocate such objects in different caches, but we also protect them with MTE, which provides strong isolation from the rest of the kernel against both spatial and temporal errors. Finally, ELOISE employs static analysis to identify allocsites that are potentially user-controlled. This however produces a large number of false positives, which the authors had to manually separate from the true positives with great effort. Instead, in IUBIK, we leverage dynamic analysis to identify allocsites of interest, which only provides true positives and no false positives.

10. Conclusion

In this paper we presented IUBIK, a novel mechanism for compartmentalizing memory in OS kernels to mitigate memory corruption exploits. IUBIK leverages the MTE hardware feature from recent ARM processors to isolate data that is user-controlled in kernel space, preventing attackers from using it to manipulate kernel-sensitive objects. For that, we rewrote a wide range of objects, and isolated their user-controlled part, preventing attackers from tampering with sensitive fields, such as pointers. We also developed a profiling framework that explored the kernel codebase in depth to reveal code sites that allocate objects with user-controlled input, which we further isolated in IUBIK.

Finally, we implemented and evaluated IUBIK in the Linux kernel, achieving low performance overhead and memory consumption, and revealing 292 and 212 privileged and non-privileged allocation sites, respectively, via profiling.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their valuable feedback. This work was funded in part by the Bavarian Ministry of Science and Arts (STMWK), under the project "Security in everyday use of digital technologies (ForDaySec)," and the National Science Foundation (NSF) through award CNS-2238467. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, or STMWK.

Availability

IUBIK is available at: https://github.com/tum-itsec/iubik

References

[1] L. Maar, S. Gast, M. Unterguggenberger, M. Oberhuber, and S. Mangard, "SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel," in *USENIX Security Symposium (SEC)*, 2024, pp. 4051–4068.

[2] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability," in *USENIX Security Symposium (SEC)*, 2022, pp. 71–88.

[3] Z. Guo, D. K. Le, Z. Lin, K. Zeng, R. Wang, T. Bao, Y. Shoshitaishvili, A. Doupé, and X. Xing, "Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation," in *USENIX Security Symposium (SEC)*, 2024, pp. 1189–1206.

[4] Z. Lin, Y. Wu, and X. Xing, "DirtyCred: Escalating Privilege in Linux Kernel," in *ACM Conference on Computer and Communications Security (CCS)*, 2022, pp. 1963–1976.

[5] Y. Chen, Z. Lin, and X. Xing, "A Systematic Study of Elastic Objects in Kernel Exploitation," in *ACM Conference on Computer and Communications Security (CCS)*, 2020, pp. 1165–1184.

[6] Y. Chen and X. Xing, "SLAKE: Facilitating SLAB Manipulation for Exploiting Vulnerabilities in the Linux Kernel," in *ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 1707–1722.

[7] M. Momeu, F. Kilger, C. Roemheld, S. Schnückel, S. Proskurin, M. Polychronakis, and V. P. Kemerlis, "ISLAB: Immutable Memory Management Metadata for Commodity Operating System Kernels," in *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*, 2024, pp. 1159–1172.

[8] L. Maar, M. Schwarzl, F. Rauscher, D. Gruss, and S. Mangard, "DOPE: DOmain Protection Enforcement with PKS," in *Annual Computer Security Applications Conference (ACSAC)*, 2023, pp. 662–676.

[9] J. Kim, J. Park, Y. Lee, C. Song, T. Kim, and B. Lee, "PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux," in *ACM Conference on Computer and Communications Security (CCS)*, 2024, pp. 2919–2933.

[10] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective Memory Protection for Kernel and User Space," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 563–577.

[11] J. Zhou, J. Hu, Z. Pan, J. Zhu, W. Shen, G. Li, and Z. Qian, "Beyond Control: Exploring Novel File System Objects for Data-Only Attacks on Linux Systems," *arXiv preprint arXiv:2401.17618*, 2024.

[12] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[13] Theori Vulnerability Research, "Linux Kernel Exploit (CVE-2022–32250) with mqueue," https://blog.theori.io/linux-kernel-exploit-cve-2022-32250-with-mqueue-a8468f32aab5, 2022.

[14] Ruihan Li, "StackRot (CVE-2023-3269): Linux Kernel Privilege Escalation Vulnerability," https://github.com/lrh2000/StackRot, 2023.

[15] M. Momeu, S. Schnückel, K. Angnis, M. Polychronakis, and V. P. Kemerlis, "Safeslab: Mitigating Use-After-Free Vulnerabilities via Memory Protection Keys," in *ACM Conference on Computer and Communications Security (CCS)*, 2024, pp. 1345–1359.

[16] ARM, "Armv8.5-A Memory Tagging Extension," https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf, 2019.

[17] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida, "Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags," in *IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 217–217.

[18] H. Liljestrand, C. Chinea, R. Denis-Courmont, J.-E. Ekberg, and N. Asokan, "Color My World: Deterministic Tagging for Memory Safety," *arXiv preprint arXiv:2204.03781*, 2022.

[19] Linux, "The Kernel Address Sanitizer (KASAN)," https://docs.kernel.org/dev-tools/kasan.html, 2024.

[20] Kees Cook, "Bounded Flexible Arrays in C," https://people.kernel.org/kees/bounded-flexible-arrays-in-c, 2023.

[21] A. Sotirov, "Heap Feng Shui in JavaScript," in *Black Hat Europe*, 2007, pp. 11–20.

[22] Qualcomm, "Pointer Authentication on Armv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[23] K. C. Knowlton, "A Fast Storage Allocator," *Communications of the ACM (CACM)*, vol. 8, no. 10, pp. 623–624, oct 1965. [Online]. Available: https://doi.org/10.1145/365628.365655

[24] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in *USENIX Summer Technical Conference*, 1994, pp. 87–98.

[25] The Linux Kernel, "Memory Allocation Profiling," https://docs.kernel.org/mm/allocation-profiling.html, 2024.

[26] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM Pointer Authentication with Speculative Execution," in *International Symposium on Computer Architecture (ISCA)*, 2022, pp. 685–698.

[27] T. J. Killian, "Processes as Files," in *USENIX Summer Conference*, 1984, pp. 203–207.

[28] J. Corbet, "An updated guide to debugfs," https://lwn.net/Articles/334546/, 2009.

[29] G. Kroah-Hartman, "udev – A Userspace Implementation of devfs," in *Ottawa Linux Symposium (OLS)*, 2003, pp. 263–271.

[30] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities," in *USENIX Security Symposium (SEC)*, 2018, pp. 781–797.

[31] M. Larkin, "Kernel W^X Improvements In OpenBSD," https://www.youtube.com/watch?v=A7vtAAeW6zo, 2015.

[32] S. Liakh, "NX protection for kernel data," https://lwn.net/Articles/342266/, 2009.

[33] A. van de Ven, "Debug option to write-protect rodata: the write protect logic and config option," http://lkml.indiana.edu/hypermail/linux/kernel/0511.0/2165.html, 2005.

[34] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against Return-to-User attacks," in *USENIX Security Symposium (SEC)*, 2012, pp. 459–474.

[35] *ARM Architecture Reference Manual ARMv8, for A-profile architecture*, ARM, 2021.

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

[37] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse," in *European Conference on Computer Systems (EuroSys)*, 2017, pp. 420–436.

[38] A. van de Ven, "Add `-fstack-protector` support to the kernel," https://lwn.net/Articles/193307/, 2006.

[39] D. Rosenberg, "`kptr_restrict` for hiding kernel pointers," https://lwn.net/Articles/420403/, 2010.

[40] T. Garnier, "Randomizing the Linux kernel heap freelists," https://mxatone.medium.com/randomizing-the-linux-kernel-heap-freelists-b899bb99c767, 2016.

[41] D. Williams, "mm: shuffle initial free memory to improve memory-side-cache utilization," https://github.com/torvalds/linux/commit/e900a918b0984ec8f2eb150b8477a47b75d17692, 2019.

[42] Kees Cook, "mm: Add SLUB free list pointer obfuscation," https://patchwork.kernel.org/patch/9864165/, 2017.

[43] Ruiqi Gong, "Randomized slab caches for kmalloc()," https://github.com/torvalds/linux/commit/3c6152940584290668b35fa0800026f6a1ae05fe, 2023.

[44] Z. Lin, "How AUTOSLAB Changes the Memory Unsafety Game," https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game, 2021.

[45] J. Horn, "`MITIGATION_README`," https://github.com/thejh/linux/blob/slub-virtual/MITIGATION_README, 2022.

[46] Jonathan Corbet, "Hardened Usercopy Whitelisting," https://lwn.net/Articles/727322/, 2017.

[47] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *USENIX Security Symposium (SEC)*, 2014, pp. 957–972.

[48] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023, pp. 527–546.

[49] J. Corbet, "Control-flow integrity in 5.13," https://lwn.net/Articles/856514/, 2021.

[50] J. Moreira, S. Rigo, M. Polychronakis, and V. P. Kemerlis, "DROP THE ROP: Fine-grained Control-flow Integrity for the Linux Kernel," *Black Hat Asia*, 2017.

[51] J. Corbet, "Kernel Support for Control-Flow Enforcement," https://lwn.net/Articles/758245/, 2018.

[52] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Fast Intra-kernel Isolation and Security with IskiOS," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021, pp. 119–134.

[53] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[54] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive Last-Level caches," in *USENIX Security Symposium (SEC)*, 2015, pp. 897–912.

[55] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium (SEC)*, 2019, pp. 249–266.

[56] D. Jin, A. J. Gaidis, and V. P. Kemerlis, "BeeBox: Hardening BPF against Transient Execution Attacks," in *USENIX Security Symposium (SEC)*, 2024.

[57] N. Christou, A. J. Gaidis, V. Atlidakis, and V. P. Kemerlis, "Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies," in *ACM Conference on Computer and Communications Security (CCS)*, 2024, pp. 3913–3927.

[58] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0LTpwn: Attacking x86 Processor Integrity from Software," in *USENIX Security Symposium (SEC)*, 2020, pp. 1445–1461.

[59] P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable Rowhammering in the Frequency Domain," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 716–734.

[60] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 1675–1689.

[61] D. Jin, V. Atlidakis, and V. P. Kemerlis, "EPF: Evil Packet Filter," in *USENIX Annual Technical Conference (ATC)*, 2023, pp. 735–751.

[62] 0xdevil and koczkatamas, "Google Security Research (kernelCTF): CVE-2023-0461_mitigation," https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-0461_mitigation/docs/exploit.md, 2023.

[63] c0m0r1, "Google Security Research (kernelCTF): CVE-2023-3390_lts_cos_mitigation," https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-3390_lts_cos_mitigation/docs/exploit.md, 2023.

[64] Zi Fan Tan and Gulshan Singh and Eugene Rodionov, "Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938," https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938/#leak-primitive, 2024.

[65] Awarau and pql, "CVE-2022-29582 An io_uring vulnerability," https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/, 2022.

[66] William Liu, "CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers," https://www.willsroot.io/2022/01/cve-2022-0185.html, 2022.

[67] Arthur Mongodin, "[CVE-2022-34918] A crack in the Linux firewall," https://web.archive.org/web/20220725044426/https://www.randorisec.fr/crack-linux-firewall/, 2022.

[68] HAXX.IN, "Exploiting CVE-2021-43267," https://haxx.in/posts/pwning-tipc/, 2021.

[69] DEVIL, "[CVE-2021-42008] Exploiting A 16-Year-Old Vulnerability In The Linux 6pack Driver," https://syst3mfailure.io/sixpack-slab-out-of-bounds/, 2021.

[70] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing Kernel Hacks with HAKCs," in *Network and Distributed System Security Symposium (NDSS)*, 2022, pp. 1–17.

[71] Mike Rapoport and Vlastimil Babka and Rick Edgecombe, "Direct Map Management," https://lpc.events/event/11/contributions/1127/attachments/922/1792/LPC21%20Direct%20map%20management%20.pdf, 2024.

[72] The Linux Kernel, "Kernel Key Retention Service," https://docs.kernel.org/security/keys/core.html, 2024.

[73] L. W. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *USENIX Annual Technical Conference (ATC)*, 1996, pp. 279–294.

[74] PTS, "Phoronix Test Suite," https://www.phoronix-test-suite.com, 2024.

[75] Android, "Build Pixel Kernels," https://source.android.com/docs/setup/build/building-pixel-kernels, 2024.

[76] Matt Kraai, "debootstrap," https://linux.die.net/man/8/debootstrap, 2024.

[77] The Linux Kernel, "nftables," https://wiki.nftables.org/wiki-nftables/index.php/Main_Page, 2024.

[78] ——, "Linux Kernel Selftests," https://docs.kernel.org/dev-tools/kselftest.html, 2024.

[79] Linux Test Project, "Linux Test Project," https://linux-test-project.readthedocs.io/en/latest/, 2024.

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

[81] Lin, Zhenpeng, "DirtyCred: CVE-2021-4154," https://github.com/Markakd/CVE-2021-4154, 2022.

[82] Man Yue Mo, "Pwning the all Google phone with a non-Google bug," https://github.blog/security/vulnerability-research/pwning-the-all-google-phone-with-a-non-google-bug/, 2023.

[83] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang, "SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 592–607.

[84] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK)," in *USENIX Security Symposium (SEC)*, 2019, pp. 1221–1238.

[85] OpenSSL, "OpenSSL," https://www.openssl.org/, 2024.

[86] Synopsys, "The Heartbleed Bug," http://heartbleed.com/, 2020.

[87] Sean Cassidy, "Diagnosis of the OpenSSL Heartbleed Bug," https://www.seancassidy.me/diagnosis-of-the-openssl-heartbleed-bug.html, 2024.

[88] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 969–986.

[89] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of {Data-Oriented} Exploits," in *USENIX Security Symposium (SEC)*, 2015, pp. 177–192.

[90] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual," https://cdrdv2.intel.com/v1/dl/getContent/671200, 2024.

[91] Google, "Syzkaller - kernel fuzzer," https://github.com/google/syzkaller, 2020.

[92] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 414–425.

[93] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing," in *European Conference on Computer Systems (EuroSys)*, 2022, pp. 266–282.

[94] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing Syscalls for PKU-based Memory Isolation Systems," in *USENIX Security Symposium (SEC)*, 2022, pp. 936–952.

[95] J. Xu, M. Xie, C. Wu, Y. Zhang, Q. Li, X. Huang, Y. Lai, Y. Kang, W. Wang, Q. Wei *et al.*, "PANIC: Pan-assisted Intra-process Memory Isolation on ARM," in *ACM Conference on Computer and Communications Security (CCS)*, 2023, pp. 919–933.

[96] M. Xie, C. Wu, Y. Zhang, J. Xu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation," in *ACM Conference on Computer and Communications Security (CCS)*, 2022, pp. 2989–3002.

[97] conlonial, "KernelCTF CVE-2024-4004_lts_cos_mitigation," https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-4004_lts_cos_mitigation/docs/exploit.md, 2023.

[98] HexRabbit, "KernelCTF CVE-2024-26925_lts_cos," https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-26925_lts_cos/docs/exploit.md, 2024.

[99] Cedric Halbronn, "SETTLERS OF NETLINK: Exploiting a Limited UAF in `nf_tables` (CVE-2022-32250)," https://research.nccgroup.com/2022/09/01/settlers-of-netlink-exploiting-a-limited-uaf-in-nf_tables-cve-2022-32250/, 2022.

[100] conlonial, "KernelCTF CVE-2024-1085_cos," https://github.com/google/security-research/tree/master/pocs/linux/kernelctf/CVE-2024-1085_cos, 2024.

[101] Xiaochen Zou and Zhiyun Qian, "CVE-2022-27666: Exploit esp6 modules in Linux kernel," https://etenal.me/archives/1825, 2022.

[102] ii4gsp, "CVE-2020-27786 (Race Condition + Use-After-Free)," https://ii4gsp.github.io/cve-2020-27786/, 2024.

[103] Andy Nguyen, "CVE-2021-22555: Turning 0000 into 10000," https://github.com/google/security-research/blob/master/pocs/linux/cve-2021-22555/writeup.md, 2021.

[104] c0m0r1, "[CVE-2023-32233] Linux kernel use-after-free in Netfilter nf_tables when processing batch requests can be abused to perform arbitrary reads and writes in kernel memory," https://seclists.org/oss-sec/2023/q2/159, 2023.

[105] Vu Thi Lan, "Breaking the Code – Exploiting and Examining CVE-2023-1829 in `cls_tcindex` Classifier Vulnerability ," https://starlabs.sg/blog/2023/06-breaking-the-code-exploiting-and-examining-cve-2023-1829-in-cls_tcindex-classifier-vulnerability/, 2023.

[106] javierprtd Blog, "CVE-2020-27786 Exploitation: `userfaultfd` + Patching `file struct /etc/passwd`," https://soez.github.io/posts/CVE-2020-27786-exploitation-userfaultfd-+-patching-file-struct-etc-passwd/, 2020.

[107] kylebot's Blog, "[CVE-2022-1786] A Journey To The Dawn," https://blog.kylebot.net/2022/10/16/CVE-2022-1786/, 2022.

[108] Moshe Kol, "Racing Against the Lock: Exploiting Spinlock UAF in the Android Kernel," https://0xkol.github.io/assets/files/Racing_Against_the_Lock__Exploiting_Spinlock_UAF_in_the_Android_Kernel.pdf, 2023.

[109] Project Zero, "A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain," https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html, 2022.

[110] Alejandro Guerrero, "N-day Exploit for CVE-2022-2586: Linux Kernel `nft_object` UAF," https://www.openwall.com/lists/oss-security/2022/08/29/5, 2022.

[111] Valentina Palmiotti, "Put an io_uring on it: Exploiting the Linux Kernel," https://chomp.ie/Blog+Posts/Put+an+io_uring+on+it+-+Exploiting+the+Linux+Kernel, 2022.

[112] Man Yue Mo, "The Android kernel mitigations obstacle race," https://github.blog/2022-06-16-the-android-kernel-mitigations-obstacle-race/, 2022.

[113] Nick Gregory, "The Discovery and Exploitation of CVE-2022-25636," https://nickgregory.me/post/2022/03/12/cve-2022-25636/, 2022.

[114] Vincent Dehors, "Exploitation of a Double Free Vulnerability in Ubuntu `shiftfs` Driver (CVE-2021-3492)," https://www.synacktiv.com/publications/exploitation-of-a-double-free-vulnerability-in-ubuntu-shiftfs-driver-cve-2021-3492.html, 2021.

Appendix A.

Table 4: Hooked alloc wrappers.

Alloc Wrapper
kasprintf
kvasprintf
kvasprintf_const
kstrdup
kstrdup_const
kstrndup
kmemdup_nul
nla_strdup

Table 5: Hooked usercopy routines.

Usercopy Routine
get_user
put_user
copy_to_user
copy_from_user
copy_to_iter
copy_from_iter
copy_to_iter_full
copy_from_iter_full
copy_from_iter_nocache
copy_from_iter_full_nocache

Table 6: List of struct types rewritten in IUBIK. 'Flex.' stands for flexible and 'CO' stands for container_of.

Struct Name→Field	Field Type	Flex.	CO	LoC (+)	LoC (-)
external_name->name	unsigned char[]	✓	✓	53	12
simple_xattr->value	char[]	✓	✗	22	9
inotify_event_info->name	char[]	✓	✗	12	6
file_handle->f_handle	unsigned char[]	✓	✗	20	4
user_key_payload->data	char[]	✓	✗	15	5
tty_buffer->data	unsigned long[]	✓	✗	17	6
neighbour->primary_key	unsigned char[]	✓	✗	18	3
unix_address->name	struct sockaddr_un[]	✓	✗	19	4
fib6_info->fib6_nh	struct fib6_nh[]	✓	✗	20	8
Node	char* (after struct)	✓	✗	16	6
net_device->name	char[]	✗	✗	32	23
linux_binprm->buf	char[]	✗	✗	11	4
devkmsg_user->buf	char[]	✗	✗	11	3
mm_struct->saved_aux	unsigned long[]	✗	✗	20	8
filename->iname	char[]	✗	✗	38	63
netdev_hw_addr->addr	unsigned char[]	✗	✗	18	4
in_ifaddr->ifa_label	char[]	✗	✗	12	2
inet6_ifaddr->addr	struct in6_addr	✗	✗	104	89
inet6_dev->token	struct in6_addr	✗	✗	14	6
kobj_uevent_env->buf	char[]	✗	✗	52	15
ext4_inode_info->i_data	__le32[]	✗	✗	17	11
key->keyring_index_key.desc	char[]	✗	✗	11	23
input_dev->*bit	unsigned long[]	✗	✗	61	13
msg_msg	char* (after struct)	✓	✗	12	5
msg_msgseg	char* (after struct)	✓	✗	12	5
nft_set->data	char[]	✓	✗	10	2
nft_userdata->data	char[]	✓	✗	39	4
tipc_aead_key->alg_name,key	char[], char[]	✓	✗	53	8
sixpack->cooked_buf	char[]	✓	✗	12	4

Table 7: Public kernel exploits mitigated by IUBIK. In the "Tactic" column, SC stands for same cache, XC for cross cache, and PO for page overflow.

CVE	Kind	Tactic	Allocsite (User-Controlled Type)	PoC
CVE-2023-4004	DF	SC	nf_tables_newtable (nft_table->udata: char *)	[97]
CVE-2023-20938	UAF	SC XC	alloc_msg (msg_msg) alloc_msg (msg_msgseg)	[64]
CVE-2024-26925	DF	SC	nf_tables_newtable (nft_table->udata: char *) nf_tables_newobj (nft_object->udata: char *)	[98]
CVE-2022-32250	UAF	SC	user_preparse (user_key_payload) setxattr_copy (xattr_ctx->kvalue: void *) nf_tables_newset (nft_set)	[99]
CVE-2022-29582	UAF	XC	alloc_msg (msg_msgseg)	[65]
CVE-2024-1085	DF	SC	nf_tables_newtable (nft_table->udata: void *)	[100]
CVE-2022-27666	OOB	PO	alloc_msg (msg_msg) alloc_msg (msg_msgseg) user_preparse (user_key_payload)	[101]
CVE-2020-27786	UAF	SC	snd_rawmidi_runtime_create (snd_rawmidi_runtime->buffer: uchar *) resize_runtime_buffer (snd_rawmidi_runtime->buffer: uchar *) alloc_msg (msg_msg) alloc_msg (msg_msgseg)	[102]
CVE-2021-22555	OOB	SC	alloc_msg (msg_msg) alloc_msg (msg_msgseg) kmalloc_reserve (sk_buff->data: void *)	[103]
CVE-2023-32233	UAF	SC	nft_set_elem_init (nft_set_ext) nft_log_init (nft_log->prefix: char *) nft_obj_init (nft_object)	[104]
CVE-2023-3390	UAF	SC	nf_tables_newset (nft_set) alloc_msg (msg_msg) alloc_msg (msg_msgseg) user_preparse (user_key_payload)	[63]
CVE-2023-0461	UAF	SC	user_preparse (user_key_payload)	[62]
CVE-2023-3269	UAF	XC	msg_alloc (msg_msg) msg_alloc (msg_msgseg)	[14]
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
CVE-2023-1829	UAF	SC	nf_tables_newtable (nft_table->udata: uchar *) nf_tables_newobj (nft_object->udata: uchar *)	[105]
CVE-2020-27786	UAF	SC	snd_rawmidi_runtime_create (snd_rawmidi_runtime->buffer: uchar *) resize_runtime_buffer (snd_rawmidi_runtime->buffer: uchar *)	[106]
CVE-2022-1786	IF	SC	msg_alloc (msg_msg) msg_alloc (msg_msgseg)	[107]
CVE-2022-20421	UAF	SC	do_tty_write (tty_struct->write_buf: uchar *)	[108]
CVE-2022-32250	UAF	SC	user_preparse (user_key_payload) msg_alloc (msg_msg) msg_alloc (msg_msgseg)	[13]
CVE-2021-25370	UAF	SC	kbase_api_mem_profile_add (char *)	[109]
CVE-2022-34918	OOB	SC	user_preparse (user_key_payload) simple_xattr_alloc (simple_xattr->value: char *)	[67]
CVE-2022-2586	UAF	SC	nf_tables_newobj (nft_object->key.name: char *) nf_tables_newtable (nft_table->udata: char *)	[110]
CVE-2021-41073	IF	SC	simple_xattr_alloc (simple_xattr->value: char *)	[111]
CVE-2022-22057	UAF	SC	msg_alloc (msg_msg) msg_alloc (msg_msgseg)	[112]
CVE-2022-0185	OOB	SC	legacy_parse_param (legacy_fs_context->legacy_data: char *) msg_alloc (msg_msg) msg_alloc (msg_msgseg)	[66]
CVE-2022-25636	OOB	SC	msg_alloc (msg_msg)	[113]
CVE-2021-43267	OOB	SC	tipc_aead_init, _crypto_key_rcv, _crypto_work_tx (tipc_aead_key->alg_name: char[32], tipc_aead_key->key: char[]) msg_alloc (msg_msg)	[68]
CVE-2021-42008	OOB	SC	sixpack_open (sixpack->cooked_buf: char[400])	[69]
CVE-2021-3492	DF	SC	shiftfs_btrfs_ioctl_fd_replace (btrfs_ioctl_vol_args)	[114]
SLUBStick-signal	DF	XC	do_signalfd4 (signalfd_ctx) do_add_key (char *)	[1]
SLUBStick-key	DF	XC	keyctl_pkey_verify (void *) do_add_key (char *)	[1]
SLUBStick-snd	DF	XC	replace_user_tlv (unsigned int *) do_add_key (char *)	[1]

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper introduces IUBIK, which isolates user-controlled objects using ARM's Memory Tagging Extension (MTE). By separating user-controlled and kernel objects into distinct memory domains, IUBIK aims to prevent memory corruption attacks. IUBIK develops a profiling tool to identify allocation sites that allocate user-controlled objects and evaluate IUBIK's effectiveness, showing low overhead and protection against many known kernel exploits.

B.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue

B.3. Reasons for Acceptance

1) IUBIK provides a structured and hardware-assisted method to isolate user-controlled objects while maintaining compatibility with existing kernel structures.
2) The runtime memory profiling tool provides insights into memory allocation patterns in the Linux kernel, which may enable further research in memory safety and exploit mitigation.
3) By leveraging ARM Memory Tagging Extension (MTE) and Pointer Authentication (PA), IUBIK presents a novel, hardware-supported approach to mitigating memory corruption attacks.

B.4. Noteworthy Concerns

1) While IUBIK would thwart a large portion of exploitation attempts, it may have potential security limitations and weaknesses that could, in theory, allow attackers to bypass its mitigation, although such bypasses have not yet been demonstrated.
2) The accuracy of the usercopy profiler can be an issue. While the empirical and manual evaluation report neither false negatives nor false positives, its design does not guarantee zero false negatives and false positives.