Safeslab: Mitigating Use-After-Free Vulnerabilities via Memory Protection Keys

Marius Momeu[*]
Technical University of Munich
Munich, Germany
marius.momeu@tum.de

Simon Schnückel
Technical University of Munich
Munich, Germany
simon.schnueckel@tum.de

Kai Angnis
Technical University of Munich
Munich, Germany
kai.angnis@tum.de

Michalis Polychronakis
Stony Brook University
Stony Brook, NY, USA
mikepo@cs.stonybrook.edu

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@cs.brown.edu

ABSTRACT

Restricting dangling pointers from accessing freed memory is a promising technique for mitigating use-after-free vulnerabilities in memory-unsafe programming languages. However, existing solutions suffer from high performance overheads, as they rely on conventional page table manipulation to make dangling pointers inaccessible. In this paper, we present `Safeslab`: a heap-hardening extension that aims to mitigate use-after-free vulnerabilities via a novel and efficient address aliasing approach. `Safeslab` assigns multiple virtual aliases to each memory page in the system, and manages their access rights via the recently introduced Memory Protection Keys hardware extension, which is designed to provide a fast alternative to page tables for memory management. This allows `Safeslab` to drastically reduce the number of page table modifications, while blocking dangling pointers efficiently. We integrated `Safeslab` into the Linux kernel, replacing its default heap allocator (`SLUB`). The results of our experimental evaluation with real-world benchmarks show that `Safeslab` incurs a negligible runtime overhead of up to 4% and moderate memory waste.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Software security engineering**.

KEYWORDS

kernel hardening, heap protection, memory isolation, Intel MPK

ACM Reference Format:
Marius Momeu, Simon Schnückel, Kai Angnis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2024. `Safeslab`: Mitigating Use-After-Free Vulnerabilities via Memory Protection Keys. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670279

[*]Also with Brown University.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10.
https://doi.org/10.1145/3658644.3670279

1 INTRODUCTION

Temporal memory errors, such as *use-after-free (UAF)*, remain one of the most common exploitation instruments leveraged by attackers to take over low-level systems software [115]. They are especially prevalent in memory that is managed dynamically by *heap allocators*, which corresponds to a large portion of the address space that applications use. Temporal memory errors are typically inflicted in the presence of pointers that reference freed objects, so-called *dangling pointers*, whose referenced memory is reused to host other objects, potentially security-critical. While dangling pointers stem from programming errors (e.g., race conditions), memory reuse is a design construct adopted by many heap allocators to improve performance and reduce memory consumption [33, 34, 47, 52, 65].

Several techniques have been proposed to mitigate temporal memory bugs. Some mechanisms try to identify dangling pointers in target programs via dynamic program analysis (e.g., *fuzzing*). However, limitations such as incomplete code coverage [76], or the absence of a sound definition for formalizing race conditions [112], render fuzzing incomplete for finding all temporal memory errors. A different line of research assumes that dangling pointers are present in target programs, and aims to neutralize them. On one hand, some mechanisms rely on augmenting memory pointers with metadata (e.g., *pointer tagging*) to track a referenced object's allocation status [21, 30, 50, 62, 117]. The additional per-pointer metadata, however, coupled with the additional runtime checks performed on every memory access, render pointer tagging unsuitable for performance-critical software, such as an OS kernel. Hardware extensions for pointer tagging, such as ARM's MTE, only provide probabilistic security guarantees and can be circumvented [9].

On the other hand, many solutions track memory pointers as they propagate at runtime, and either invalidate them once objects get freed [66, 100, 113], or reuse them only after ensuring the absence of dangling copies [90]. Nevertheless, accurately tracking the flow of *all* memory pointers throughout execution requires instrumenting all instructions that operate on them, which has two major limitations. First, identifying all instructions that operate on pointers cannot be achieved accurately in un-typed programming languages, such as C and assembly—pointers can be stored as integers, which is, in fact, a common practice used in the Linux kernel. Second, tracking all operations performed on pointers at runtime incurs a significant performance overhead [66, 90, 100, 113].

As an alternative, mitigation techniques based on avoiding the reuse of pointers (or virtual addresses) have received increased attention lately, as they are tailored towards performance. However, despite being fast, they intrinsically suffer from exhaustion of *physical* and/or *virtual* memory. To tackle the former, existing techniques unmap virtual addresses of freed pointers, and reuse their physical memory for other virtual addresses (called *aliases*) [27, 39, 106]. However, (un)mapping virtual memory requires modifying entries in the system's *page table* (PT), which entails several memory accesses as well as flushing the modified entries from the TLB of all CPUs (*TLB shootdown*), both of which incur a significant performance slowdown. The latter is typically tackled by scanning the program memory (and registers) similarly to a garbage collector [11], and re-enabling freed virtual addresses for which no dangling pointers are found [1, 39]. Nevertheless, existing techniques require scanning memory too frequently which results in high performance overhead, and simply ignore false positives which may be abused by attackers to exhaust the resources of the system.

In this paper we present `Safeslab`, a heap hardening extension that mitigates temporal memory errors efficiently in the presence of dangling pointers. `Safeslab`'s key mitigation strategy is hybrid: first, it temporarily *quarantines* freed memory objects, and second, once the quarantine reaches a certain threshold, it reuses freed memory with new (alias) virtual addresses, while disabling access to the old ones. With `Safeslab`, dangling pointers either reference quarantined objects, or invalid address aliases of reallocated objects, rendering them ineffective for building exploitation primitives. To achieve that, we propose a novel mechanism that enables `Safeslab` to optimize two costly operations that lead to slowdowns in existing solutions: (a) reducing the frequency of pointer scans for safely recycling used aliases, and (b) quickly re-enabling quarantined objects with new aliases while disabling old ones.

For (a), we introduce a novel memory layout abstraction, dubbed *temporal aliasing domains (TADs)*, which are pre-configured ranges of virtual addresses that *alias*, i.e., they map the same physical memory. Within the lifespan of a TAD, the physical memory freed by `Safeslab` gets quarantined and not reused until the quarantine hits a certain *threshold*. Only then, `Safeslab` switches to a new TAD, which enables new aliases and disables the old ones, allowing `Safeslab` to reuse the quarantined memory. As opposed to prior work, which relies on slow PT manipulation for (de-)activating aliases, we achieve (b) via the *Memory Protection Keys (MPK)* feature of Intel CPUs [48], which allows `Safeslab` to control access rights to virtual pages efficiently. `Safeslab` maintains a finite pool of TADs (aliases), which could be depleted in the presence of many object (de-)allocations. Instead, inactive TADs are recycled safely, by only re-enabling addresses that have no dangling pointers. This is achieved using a *pointer scanner* (similar to a garbage collector [11]) that examines the system's memory and registers, and records references to freed objects. We deactivate dangling pointers found by the scanner (including false positives) by disabling their PT mappings, which allows `Safeslab` to reuse their physical memory with other aliases, thus avoiding memory exhaustion. Most existing heap hardening solutions target user-space applications [1, 34, 39, 90–92, 106, 117], while significantly fewer case studies have been carried on heap allocators used in kernel space [21, 46, 47, 50].

Nevertheless, new CVEs that target temporal memory bugs, and new heap exploitation mechanisms, are constantly being discovered in OS kernels [19, 20, 26, 88, 89], while several public exploits exist for kernel vulnerabilities that target temporal memory errors [2, 8, 15, 36–38, 51, 53, 63, 64, 67, 80, 85, 86, 96, 98, 103, 116]. We aim to fill this gap with our work, and provide a prototype and evaluation for `Safeslab` that hardens `SLUB`, the default heap allocator in the Linux kernel. Our research prototype is available as open-source software, and can be used by the community for adopting and extending it.

In summary, we make the following main contributions:

- A novel heap hardening mechanism, called `Safeslab`, which mitigates temporal memory errors in dynamic memory allocators via isolated aliasing domains (TADs) and Intel MPK.
- A pointer scanning approach that allows `Safeslab` to safely reuse memory pages without access for dangling references, while also preventing exhaustion due to false positives.
- Implementation and evaluation of `Safeslab` for `SLUB`, the default heap allocator in the Linux kernel.

2 BACKGROUND

2.1 Slab-based Allocation in Linux

The Linux kernel uses primarily two *dynamic memory allocators* to manage in-kernel memory: a *page allocator* and an *object allocator*. The former leverages the *buddy system* to satisfy memory requests on a page granularity [60], while the latter leverages the *slab* approach to facilitate efficient memory allocation on a sub-page granularity [12]. The slab allocator 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*).

`SLUB` caches freed objects in linked lists called *freelists* [12], stored within the slabs themselves, and uses them to serve allocation requests. Reallocating freed objects via freelists is one of the main weaknesses abused by attackers to exploit UAF bugs in `SLUB` [74] (and other performance-oriented allocators), which is why our solution replaces `SLUB` in the Linux kernel and avoids reallocating freed objects. Additionally, `Safeslab` does not store any kind of metadata within object slabs, since they can be corrupted by attackers in the presence of spatial memory errors (this has been used to exploit `SLUB`'s freelists [41, 63, 72, 77, 81, 104]).

`SLUB` uses a dedicated data structure (called the *object cache*) to manage multiple object slabs (some per-CPU, others per-NUMA node) for a specific object type [13]. The per-CPU slabs allow `SLUB` to perform simultaneous memory (de-)allocations onto the same object cache without holding a lock. Furthermore, the kernel stores slab-specific information in the data structure `struct page` [13], representing the physical pages that make up the slab [57]. These are kept in the `vmemmap` [59] area of the kernel. `Safeslab` also uses these structures to manage its slabs.

2.2 Memory Errors

Software written in memory- and type-unsafe languages, such as C/C++/ASM, may generally encounter two types of memory errors on heap-allocated objects [114]. *Spatial memory errors*, manifested as *out-of-bound* (OOB) errors, enable attackers to corrupt or leak data stored in neighboring objects (linear OOB), or arbitrary objects (non-linear OOB)—such errors are out of scope for `Safeslab`.

Temporal memory errors, manifested as *use-after-free* (UAF), *double-free* (DF), and *invalid free* (IF) errors, occur when *dangling pointers* reference previously-freed objects, which are later reused in different (potentially sensitive) execution contexts, effectively leading to illegal accesses on those objects. Dangling pointers stem from (i) improperly invalidating memory pointers once they get freed (UAF), (ii) race conditions that lead to freeing the same memory pointer twice (DF), or (iii) programming errors that allow calling `free()` on attacker-controlled pointer values (IF). Attackers can abuse the allocator to build exploitation primitives in two ways: (a) if the dangling pointer already references the same page as the target object, attackers manipulate the allocator's freelist (via *heap spraying*) to reallocate the target object so that the dangling pointer references it [114]; (b) if the dangling pointer does not yet reference the target object's page, attackers manipulate the page allocator to return the dangling pointer's page when allocating new heap pages [108]. `Safeslab` is resilient to both, as it never reallocates memory objects or pages for which dangling pointers exist.

2.3 Intel Memory Protection Keys

Intel MPK [48] is a recent feature of x86 CPUs that complements the memory access permissions maintained in the PTs for faster memory access management. Under MPK every PT entry stores a four-bit *protection key* (PK), allowing the whole virtual address space to be tagged with one of the 16 possible keys at page granularity. Each key is assigned one of four possible access rights (`enable`, `read`, `write`, `disable`) through a corresponding two-bit entry in a dedicated CPU register (`PKRU`). If a key's access rights are set to `disabled`, the CPU generates a page fault even if the page table's access rights permit the access. `Safeslab` leverages Intel MPK to isolate its TADs and block dangling pointers.

Intel MPK has two modes: *Protection Keys for Userspace* (PKU) [94] and *Protection Keys for Supervisor* (PKS) [24]. The former regulates access to user pages, and the latter to kernel pages. As PKS is not yet available, `Safeslab` configures the system to enable PKU in kernel mode—similarly to prior work [40]. PKU extends the CPU's ISA with a new instruction (`WRPKRU`), which can be executed by code to enable/disable access rights for protection keys in the `PKRU` register. Intel MPK's great advantage is its speed: a write in `PKRU` takes only about ≈25 cycles. `Safeslab` executes the `WRPKRU` instruction to modify the PK access rights when switching TADs.

3 THREAT MODEL

We assume that the hardened system (i.e., the OS kernel) contains vulnerable code that generates dangling pointers to heap-allocated objects (e.g., via UAF, DF, IF temporal memory errors). Attackers can trigger an arbitrary sequence of (de-)allocations to manipulate the allocator into overlapping dangling pointers with victim objects and build exploitation primitives. `Safeslab` prohibits overlapping by *not* reallocating freed memory objects.

As we integrate `Safeslab` into the Linux kernel, we only configure it to mitigate UAFs on the memory normally managed by `SLUB`, and we consider out of scope mitigating UAFs on the stack (which are short-lived in the kernel), or on other in-kernel memory allocators, such as `vmalloc` [25] and the subsystems that allocate memory from the buddy allocator directly.

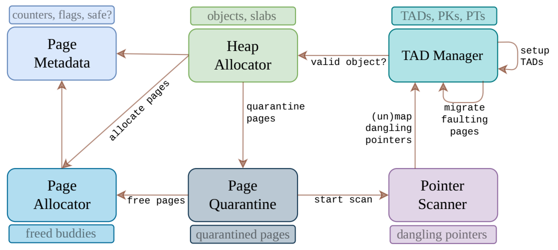

Figure 1: High-level overview of `Safeslab`'s architecture.

Nevertheless, most public UAF exploits in the kernel target `SLUB` [20, 109], and are thus mitigated. Finally, although `Safeslab` withstands trivial OOB bugs that target allocator metadata, generic spatial memory errors are outside the scope of our work, as well as attacks exploiting micro-architectural flaws [14, 54, 61, 69, 111].

4 DESIGN

In this section, we first present `Safeslab`'s overall architecture, and then we describe in detail the inner workings of its components.

4.1 Architectural Overview

A system hardened with `Safeslab` generally consists of the components illustrated in Figure 1. The *Heap Allocator* is responsible for object (de-)allocations on memory pages that are used for the heap. These pages are retrieved from the *Page Allocator*, which manages the system's free memory at page granularity. When freeing pages, instead of giving them back to the page allocator (like the original allocator does), `Safeslab` puts them into the *Page Quarantine*. The system maintains auxiliary page-management metadata for each page in the *Page Metadata* region, which serve either a functional or a security purpose—these are used by all `Safeslab` components.

`Safeslab` provides multiple virtual aliases for each memory page via *temporal aliasing domains (TADs)*. TADs are maintained by the *TAD Manager*, which leverages the underlying hardware (i.e., PTs and PKs) to (de-)activate them efficiently. Once the page quarantine reaches a certain *threshold* (e.g., 4*GB*), `Safeslab` initiates a TAD switch on all CPUs, and releases the quarantined pages back to the page allocator. This allows the heap allocator to safely reuse the freed pages with fresh virtual addresses (i.e., aliases) from the newly activated TAD, while access to dangling pointers in the deactivated TAD is blocked by PTs or PKs.

`Safeslab` switches TADs optimistically, without waiting for all the objects that were allocated during the prior TADs to be freed (e.g., long-living objects used by the system). Accesses to such objects will trap in the page fault handler (#PF); however, since these accesses are legitimate, `Safeslab` *migrates* them into the active TAD, where they can be accessed. Nevertheless, in order to overcome the finite amount of TADs, `Safeslab` reuses them in a safe manner—i.e., when it approaches the last TAD, `Safeslab` invokes the *Pointer Scanner* to search for dangling pointers to deactivated TADs. Then, when reactivating a TAD, it only uses memory pages for which the scanner found no dangling pointers in that TAD.

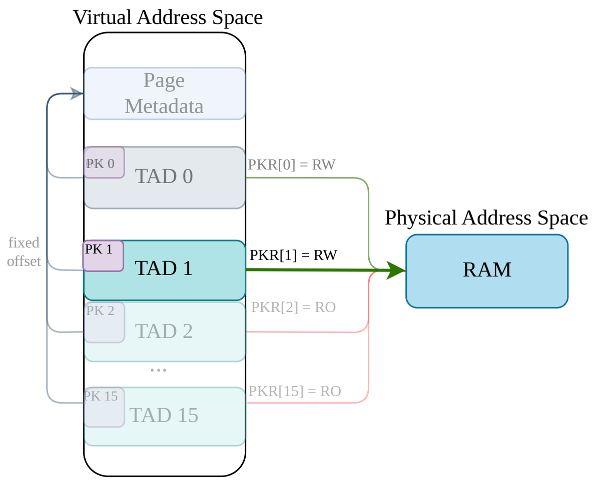

Figure 2: Memory layout of Safeslab with 16 *temporal aliasing domains (TADs)*, isolated via 16 *protection keys (PKs).*

4.2 Temporal Aliasing Domains

We define a *temporal aliasing domain (TAD)* as a contiguous range of virtual addresses that maps the entire physical RAM. Different TADs provide *virtual aliases* for each physical page. This is key to our UAF mitigation approach, since aliases allow Safeslab to use a new virtual address every time a freed physical page is reallocated. TADs are implemented via mappings in the kernel's PTs, and in order to avoid their management overhead at runtime, Safeslab (pre-)configures multiple TADs during initialization (in the case study presented in this paper, Safeslab uses 16 TADs). Figure 2 illustrates the memory layout of Safeslab once the TADs are set.

Safeslab marks each TAD with a different PK during initialization and governs their access rights at runtime as it (de-)activates TADs. Specifically, at any time, Safeslab maintains a single (active) TAD whose PK rights are enabled, while access rights of the other PKs, and, consequently, their corresponding TADs, are disabled. This allows Safeslab to safely reuse a freed physical page with a new alias from the newly activated TAD—any dangling pointers to it that may have been produced in prior TADs are blocked thanks to the lack of access rights to their PKs. Safeslab uses the maximum number of available PKs to configure TADs during initialization; in the case of Intel's MPK, this value is 16 (see Section 2.3). We reserve TAD#0 (tagged with PK#0) to serve memory that is not managed by Safeslab, such as the stack, and we never disable access to it, since the system needs it to function correctly—protecting this memory is outside the scope of our threat model (see Section 3).

Safeslab leverages a segregated region of memory to maintain auxiliary metadata for each physical page available in RAM—the *Page Metadata* region. This is accessed by all Safeslab components, frequently, and is essential for memory management. Thus, Safeslab configures it at a fixed offset from the pool of TADs, allowing it to be accessed quickly at runtime via trivial pointer arithmetic. The metadata store several auxiliary fields regarding both the functionality and security of each page in memory.

For example, Safeslab maintains object counters for each page, and uses them to perform efficient object (de-)allocations at runtime (see Section 4.3). Moreover, we keep track of the safety status for each memory page, which is updated by the pointer scanner when it finds dangling pointers to the page (see Section 4.8).

4.3 The Heap Allocator

The heap allocator manages object (de-)allocations within a memory page, and requests new pages from the page allocator when the allocated ones are full (and releases them back when they become empty). Safeslab begins a series of configurations for every new page, which prepare it for serving object (de-)allocations. Most notably, it sets the index of the current active TAD in its page metadata, which is used to compute the virtual addresses of the objects returned during allocation requests. Thus, object users get a different (alias) virtual address for their objects with each TAD.

Safeslab adopts a *no-reuse* object management strategy: it computes the next available virtual address for every new allocation, and never reuses that address during the life-cycle of that page. This prevents attackers from manipulating Safeslab into overlapping new objects on memory referenced by dangling pointers, which is the main technique used for exploiting UAFs. Hence, Safeslab does not need to maintain a freelist for keeping track of freed slots to be reused on subsequent allocations; this is a common approach adopted by insecure heap allocators (such as SLUB [65] or ptmalloc [107]). As such, Safeslab not only prevents attackers from abusing UAF bugs, but also stops them from manipulating in-object allocator metadata via OOB accesses to craft (arbitrary R/W) exploitation primitives. If it runs out of unallocated objects on a memory page, Safeslab gets a new one, as described above.

When Safeslab frees an object it first performs a double-free check by testing the object's allocation status. If the object has not been freed before, Safeslab marks it as such, and proceeds by updating that page's memory management metadata, where it keeps track of its remaining unallocated slots. When all objects on a page are freed, Safeslab quarantines it until it can safely be reused with a new alias—this does not happen, however, until the next TAD switch (see Section 4.4). Hence, the page allocator cannot be manipulated into reallocating freed pages that might be referenced by dangling pointers, thus preventing *cross-cache* UAF attacks.

Contrary to performance-oriented allocators (such as SLUB or ptmalloc), Safeslab does not reuse objects once they are freed, thereby resulting in higher virtual address-space utilization. As such, Safeslab must invoke the page allocator for fetching/freeing memory much more often, which adds performance overhead. To alleviate this, we increase the number of pages that Safeslab allocates at once to 4× the amount requested by the original allocator.

4.4 The Quarantine

Safeslab avoids reusing freed pages within the life-cycle of a TAD by quarantining them in a singly-linked list until the next TAD switch. The page quarantine is a global resource, for which all CPU threads are competing, hence we synchronize access to it via a lock. Although this could be avoided by configuring per-CPU quarantines, in our experiments we did not find quarantining to be a significant source of overhead (see Section 6.1).

We define a configurable quarantine threshold that, when reached, triggers a TAD switch, after which Safeslab releases the quarantined pages back to the page allocator. For the case study presented later in the paper, we set the quarantine watermark to 4GB of memory. Based on our experiments, a quarantine threshold of 2GB does induce a performance penalty on some benchmarks, while increasing it to 8GB did not result in any observable difference.

Before switching to the new TAD, Safeslab demotes the current quarantine, and promotes an empty one, which allows it to release the quarantined slabs on one CPU, while the others can continue quarantining freed slabs simultaneously. Safeslab switches TADs by sending *inter-processor interrupt (IPI)* signals to the other CPUs, which get interrupted to execute the *TAD-switching routine*. Concretely, each CPU first updates their *active-TAD*, a per-CPU variable that we use to keep track of the active TAD on each CPU; this spares them from waiting for the other CPUs until they switch their TAD. Second, each CPU updates its TAD access permissions in the PKRU register via the WRPKRU instruction, by disabling RW access for the old TAD, and enabling it for the new one. This prevents any dangling pointers to prior TADs from accessing the pages that are released from the quarantine.

Once IPIs are sent, the initiating thread proceeds with emptying the demoted quarantine, while also releasing the quarantine lock, which makes it available for other threads to use. Releasing the demoted quarantine is fairly straightforward: Safeslab walks the linked list and frees every page to the page allocator. Next, as Safeslab eventually runs out of available TADs, the working thread checks whether it has reached a predefined TAD (for the case study presented in this paper this is TAD#12—i.e., the *scanning TAD*), which is a sign that it will soon have to recycle prior TADs. In that case, Safeslab invokes the pointer scanner, which is configured as a separate task during initialization (see Section 4.6).

4.5 Page Migration

As Safeslab disables the PK of a TAD when releasing the quarantine, it may remove access to long-living objects that lie on pages allocated in the deactivated TAD. Subsequent memory accesses to such objects will trigger CPU exceptions due to invalid access rights, even though, semantically, these accesses are legitimate and should be allowed. Safeslab addresses this by hooking the page fault (#PF) handler and marking a faulting page with the PK of the currently active TAD—we call this process *migrating pages across TADs*. Safeslab achieves this by updating the PK bits of the faulting address' translation entry in the page table, which re-enables its access upon re-executing the faulting instruction. Note that migrated pages can contain both allocated and freed objects—this makes no difference in the migration approach.

Also, note that besides getting their PKs updated during page migrations, TADs are fixed mappings and do not change their view of physical memory at runtime. This is key to our approach, as it empowers Safeslab to avoid keeping track of all pointers in the system for adjusting their addresses, like typical garbage collectors do for migrating objects on different pages; we only need to update their PT entry to reflect the PK of the active TAD, which restores their access. Nevertheless, Safeslab makes sure to only migrate pages if the faulting address points into a valid page, i.e., allocated, and whose TAD address matches that of the page's, which we store in the page's auxiliary metadata. This way, attackers cannot migrate dangling pointers into the active TAD and overlap them onto allocated objects, thus preventing UAF bugs.

For a faulting address on a legitimate heap object, Safeslab proceeds by walking the PTs and fetching its corresponding leaf PT entry (needed to update the PK of the page), which, in modern systems equipped with *huge pages*, may map virtual memory in chunks of $2MB$ or $1GB$ blocks (to reduce the number of translation steps [48]). Safeslab migrates memory conservatively—it only migrates the faulting page across TADs, thus breaking huge mappings into smaller ones until it obtains the $4KB$ page that the faulting address maps onto. Next, Safeslab updates the PK bits of the resulting PT entry with the value of the current TAD, flushes the TLB of the active CPU, and resumes execution. Note that Safeslab's object migration is not a *security* step, but rather a functional one, which does not require a *TLB shootdown* for flushing the affected translation from all CPUs. In the worst case, another CPU may use a cached TLB entry with the older PK value, which will trap in the page fault handler and detect it as correctly mapped, allowing it to simply resume execution.

4.6 The Pointer Scanner

The pointer scanner finds all dangling pointers that reference freed pages and marks them as unsafe. At runtime, the heap allocator checks the safety status of pages retrieved from the page allocator and rejects those that were marked as unsafe. This allows Safeslab to only reuse aliases from deactivated TADs that have no dangling pointers, which renders UAF exploits ineffective. State-of-the-art pointer scanners can mostly run concurrently with the scanned tasks, and only require stopping them briefly at the beginning and end of the scan to prevent dangling pointers from evading the scan [1, 11, 29]. Although Safeslab is compatible with such techniques, for simplicity, we stop all tasks running in *kernel mode* during the pointer scan. User-space tasks continue to handle non-kernel workloads, as we only block them when they invoke the kernel via system calls during the scan. Although we did not find the pointer scanner to be a significant source of overhead (see Section 6.1.2), we discuss ways for optimizing it in Section 7.

Safeslab adopts a conservative pointer scanning strategy to deal with *weakly-typed* codebases (like the Linux kernel): it assumes every data word in the system's memory may potentially store a pointer. Therefore, Safeslab reads all registers and memory that are in-use (i.e., stack, heap, .bss, .data, *etc.*, sections) in the target system, in 8-byte increments, as it assumes that pointers always reside at 8-byte aligned addresses in memory to satisfy *natural alignment* requirements [95]. When encountering a sequence of bytes that may form a dangling pointer, the scanner skips marking its page as unsafe if they are part of the same TAD, which means that the page is still in-use. This does not weaken Safeslab's security, as such a pointer can reference either a freed object or a living object on an allocated page, which is *guaranteed* by the heap allocator to not overlap with any other object thanks to its *no-reuse* approach. Otherwise, the scanner marks the page as unsafe for the TAD it belongs to—Safeslab will only reuse it in other TADs.

Safeslab carefully marks as unsafe all pages that may be reached by the dangling pointer, such as those stemming from multi-page objects. For that, Safeslab keeps track of each domain the page is used in by storing additional information in the page's metadata. To improve efficiency when scanning Safeslab's own heaps, the scanner only traverses memory pages that are in-use by the heap allocator, and only scans heap objects that are allocated. This does not allow attackers to violate Safeslab's temporal safety property because: (a) attackers can only reach dangling pointers through allocated objects; and (b) both Safeslab and the page allocator wipe the contents of freed memory before allocating it, thus preventing dangling pointers from propagating into newly allocated objects.

4.7 Addressing False Positives

Safeslab's conservative approach impedes it from distinguishing real dangling pointers from *false positives*, i.e., arbitrary sequences of bytes that yield valid pointers. Thus, we must treat them conservatively too, and mark their respective memory pages as unsafe, which signals the heap allocator to reject and put them in the quarantine. Consequently, a whole page can become unusable due to a single false positive, which may put unwanted pressure on the system's memory consumption when large amounts are found. This issue has largely been ignored by prior work [1, 39], even though codebases that process and store high amounts of data, like the Linux kernel, exhibit non-negligible amounts of pointer-looking false positives (see Section 6.2). To alleviate this problem, we fall back to PT manipulation and unmap the PT entry of all (true or false) dangling pointers found by the scanner, on a per TAD basis.

Specifically, we invalidate the page table translation of a dangling pointer's TAD address, which allows Safeslab to safely reuse its physical page with aliases from the other TADs, as well as for other, non-heap memory, such as the stack, since any attempt to access it after this procedure is blocked by the unmapped PT entry. Then, Safeslab restores access to unmapped pages, if it finds no dangling pointers to them in the current scan. We refrain from interrupting the other threads for flushing stale translations from their TLBs, in order to avoid penalties in performance. Instead, Safeslab postpones reusing a page whose dangling pointer got unmapped until every thread performs a TLB flush (usually done during the next context switch).

4.8 Marking Unsafe Pages

Safeslab relies on the scanner to determine if dangling pointers to memory pages exist anywhere in the system. However, Safeslab must make sure to not reuse a freed memory page in a TAD before the scanner gets to be executed. Nevertheless, as Safeslab does not impose any restriction on the lifetime of a page with respect to its originating TAD, memory pages can be freed while a different TAD is active, or they can be freed after multiple reuse cycles of all TADs. This breeds several scenarios in which Safeslab cannot guarantee that a dangling pointer to a freed page exists or not, and hence requires the heap allocator to preventively mark pages as unsafe, until the scanner is executed. Safeslab marks a page as unsafe if it is freed in the following scenarios: (1) if its an address in the scanning TAD (TAD#12 in our case)—this is because Safeslab does not know exactly when the scheduler will awake the scanner while TAD#12 is in use, and thus we avoid the risk of reusing pages whose dangling pointers may not have been found by the scanner; (2) when the page's TAD address is larger than the CPU's active TAD, but smaller than the scanning TAD; and (3) when the active TAD is larger than the scanning TAD and the page's TAD address is either larger than the active TAD or smaller than the scanning TAD. Both (2) and (3) are required because the scanner does not execute between the current TAD and the freed page's TAD.

In addition, Safeslab marks pages as unsafe whenever they are migrated—this is to avoid reusing a page that got migrated into a TAD and freed before the CPU switches into that TAD, since the scanner cannot guarantee that it has no dangling pointers, as it did not run. Nevertheless, Safeslab needs a mechanism to determine if a page that was previously marked as unsafe becomes safe for reuse. This can be either because the previously-found dangling pointers have disappeared, or because the page was not part of the scenarios described above. For that, we configure the pointer scanner to maintain a *scan id* (sid), which it increments at the end of every scan. Both the scanner and the heap allocator also set the sid of a page whenever they mark it as unsafe. The sid of a page is then checked by the heap allocator before accepting a new page, and if it does not match the last sid of the scanner, it considers the page as safe, although it may have been marked as unsafe previously (i.e., during a previous scan).

5 IMPLEMENTATION

To assess Safeslab's effectiveness, we integrated it into the Linux kernel (v6.2.0) and configured it to replace the kernel's default heap allocator (SLUB). Thus, Safeslab is able to leverage several traits available to an OS kernel, such as direct access to privileged PTs, or executing privileged instructions (e.g., for flushing the TLBs). We discuss in Section 7 the challenges that need to be addressed in order to deploy Safeslab in user space.

5.1 PKU in Kernel Space

In our current prototype, we isolate TADs using Intel's MPK extension (we discuss in Section 7 how other isolation primitives could be integrated into Safeslab). However, at the time of this writing, MPK only supports its user-space mode (called *PKU* [94], see Section 2.3). Hence, we repurposed PKU to isolate kernel memory, by marking its PT translation entries as user-mode (i.e., by setting the U/S bit). In order to prevent user applications from accessing kernel pages we rely on KPTI [42, 58]; we also rely on existing solutions [40] to re-implement SMEP/SMAP and prevent the kernel from accessing user memory in a *confused deputy* manner.

Moreover, since PKU is an unprivileged extension, applications can use it to manipulate the PKRU register via the wrpkru and xrstor instructions. This could be abused by malicious programs to disable Safeslab's isolation before entering kernel mode. We mitigate this issue by saving the PKRU value in a per-CPU variable before entering user mode (i.e., by using the RDPKRU instruction, whose latency is ≈1 cycle [79]), and checking its integrity upon returning to kernel mode. This way, any modifications applied on PKRU in user space (via wrpkru/xrstor) will be detected by Safeslab, which can then restore its saved value.

We believe this is a *lean* approach, which enables Safeslab to avoid the expensive requirement of scanning code pages at a program's startup time (to ensure that untrusted code does not contain instructions like wrpkru/xrstor [40, 97]) or intercepting syscalls (e.g., mmap, mprotect, process_vm_readv, ptrace, open) to prevent wrpkru/xrstor from being injected at runtime [40, 45, 97]. Safeslab can also avoid intercepting syscalls that may trick the kernel into a confused deputy scenario to change PKs on isolated pages (as demonstrated in prior work [23, 87, 102]), since syscalls are carefully designed to avoid operating on the kernel portion of the address space. Nevertheless, we plan on integrating PKS [24] into Safeslab once it becomes available, which will allow us to disable all the above defenses.

As PKU and PKS are twin technologies, we do not expect a significant engineering effort in doing so. Although manipulating access permissions in PKS will be done via rdmsr/wrmsr, which might have a different latency than rdpkru/wrpkru, we do not anticipate any differences in performance since MPK-based domain switches in Safeslab are rare (TAD switching). We also do not expect a difference in memory consumption or functionality.

5.2 Memory Management

Safeslab's heap allocator reuses several constructs from SLUB, as they were designed to facilitate performance. Specifically, it also uses the *slab-based* allocation strategy, by grouping several memory pages to form object slabs that it uses to serve (de-)allocations for several types of objects. Safeslab also configures per-CPU object slabs, which allows it to serve simultaneous (de-)allocations from the same object cache on multiple CPU threads, without locking. Contrary to SLUB, Safeslab does not maintain slab freelists, since it never re-allocates slab objects once they are freed. Instead, Safeslab maintains an object *bitmap* for each slab, where it keeps track of each object's allocation status—Safeslab uses this to detect *double-free* bugs (see Section 4.3). Safeslab sequesters the bitmap outside the slabs themselves, in dedicated memory, where it is protected from overflows on slab objects. This results in higher security than SLUB's current design, which keeps freelist pointers within the objects themselves, where they can be easily corrupted in the event of overflows [74].

Safeslab fetches new pages to form slabs from the kernel's *buddy allocator* (see Section 2.1), which manages all freed physical pages. As invoking the buddy allocator frequently may result in performance overhead, we configured Safeslab to allocate 4*x* more pages than SLUB would for an object slab. As such, the number of pages in a slab will always be a multiple of 4; we leverage this to reduce the number of pages touched by Safeslab when marking and checking the safety of a slab, by only iterating over the slab pages that are at an index whose value is a multiple of 4. Furthermore, when migrating pages across TADs, in the page fault handler, Safeslab adopts a conservative approach: it only migrates the accessed page, and not the whole slab.

Safeslab leverages the kernel's *vmemmap* region [59], which reserves a 64-byte object, called *struct page*, to maintain additional metadata for each physical page of the system. For every page, we store its slab *order* and its slab *index* in the slab it was last part of (in each TAD). This is used by the scanner upon detecting a dangling pointer to unmap all pages that were part of that slab, which also allows us to block access to both single- and multi-page objects. We configured Safeslab's TADs based on the kernel's *physmap* region [57], which provides a direct mapping of the entire RAM at a constant offset in the virtual address space. We also modified the kernel macros used to compute various values based on a virtual address, such as virt_to_{phys, page}, to take into account the TAD that the address belongs to. This is done via pointer arithmetic operations, as TADs are contiguous memory regions at a fixed offset from each other (and the beginning of the address space).

5.3 The Pointer Scanner

We configured Safeslab's scanner to look for dangling pointers in the following memory regions: the .bss section, the global data, the .ro_after_init section, the per-CPU regions, the kernel stacks, and the memory allocated dynamically by Safeslab. When Safeslab signals the running threads to stop for allowing the scan, it only targets those that run in kernel mode, while user-mode threads can continue running—as they do not use Safeslab for memory management, user tasks cannot introduce dangling pointers in the kernel. We also modified the task scheduler to avoid scheduling tasks that were executing in kernel mode when they got de-scheduled, and yield the CPU to allow other user tasks to resume execution. Safeslab considers kernel threads trusted (e.g., the page swapper, worker threads), allowing them to run during the scan.

To serve its role as a trusted channel between user programs and privileged hardware, the OS kernel must process and store user data. Malicious user applications may abuse this to fill kernel memory with pointer-looking values (i.e., false positives), which may lead to virtual address depletion, as they get marked as unsafe by the pointer scanner and rejected by Safeslab. To counter that, we configure the scanner to skip reading kernel memory that contains user data, by leveraging the *hardened* usercopy feature of the Linux kernel [55], which keeps track of offsets and lengths in heap objects that store user data (via the useroffset/usersize fields of the kmem_cache). While this works reliably with kmem_cache caches, it is too coarse-grained for kmalloc caches, as it must allow multiple objects with different user-space access patterns to be stored in the same slab. To mitigate this, we extend Safeslab's *bitmap* for kmalloc caches to also store the offset and length within the object where user data is stored. We also instrumented the routine copy_from_user, which is the main interface for copying user data into the kernel, to store for each kmalloc object that it touches the offset and length of where it stored user data. This is then used by the pointer scanner to skip those regions at runtime.

6 EVALUATION

We evaluate Safeslab when used as replacement to Linux's default memory allocator (i.e., SLUB). First, we analyze its performance and memory overhead, as these are crucial for an in-kernel heap allocator. Next, we describe how our extension mitigates UAF bugs exhibited by SLUB. Our tests were carried out on a host armed with a 24-core 12th Gen Intel Core i9-12900 CPU (1 socket, 16 cores/socket, 2 threads/core), 1 NUMA node, and 64GB of RAM.

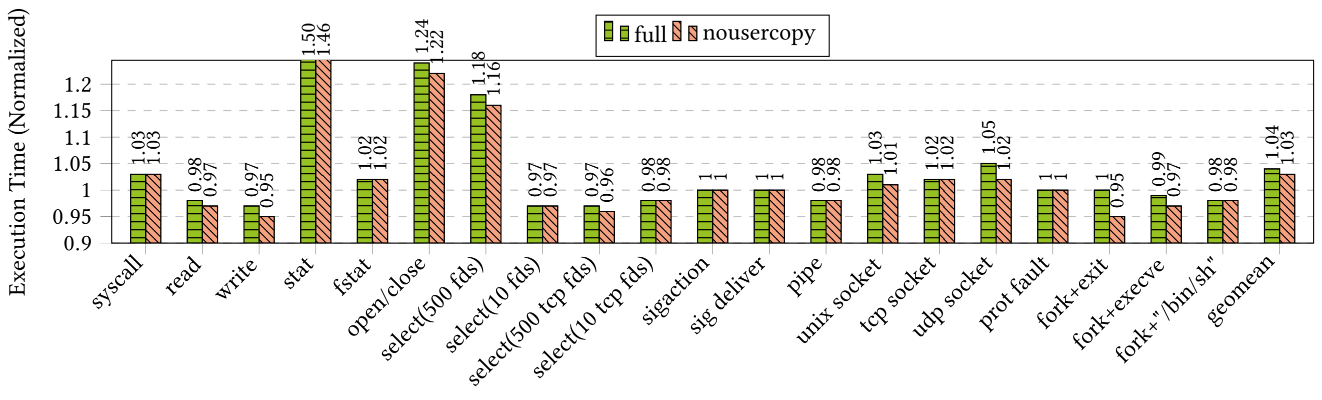

Figure 3: LMbench performance results on Safeslab. Results are averaged over 20 runs and normalized to (vanilla) SLUB.

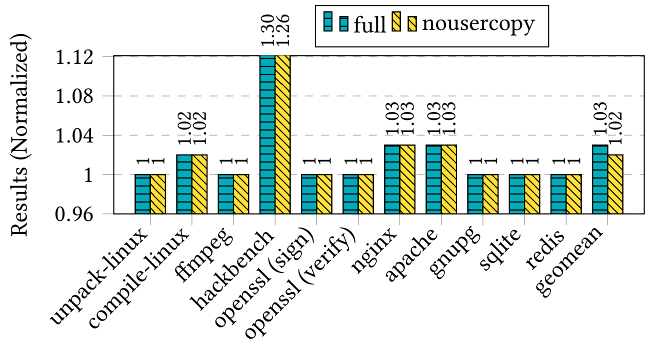

Figure 4: Performance overhead of macro-benchmarks from PTS on Safeslab. The default parameters were used for each benchmark, except for hackbench (16 proc. groups), apache (100 conc. requests), nginx (100 conn.), and redis (16 threads) to saturate (i.e., increase the utilization of) the CPU. Results are normalized to (vanilla) SLUB.

6.1 Runtime Overhead

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

6.1.1 LMbench. We employ the LMbench [73] micro-benchmark to examine Safeslab on tests that stress individual components of the underlying kernel. As these tests execute small payloads repeatedly, they are well suited to evaluate the performance of Safeslab's heap allocator (see Section 4.3) and the implications of quarantining freed memory (see Section 4.4). The other Safeslab components, i.e., TAD switching, migrating slabs across TADs, and the scanner are not triggered during the LMbench tests—we use macro-benchmarks to evaluate them (next section).

Figure 3 shows the results of our experiments carried in two settings: with all Safeslab features enabled (*full*) and with Safeslab's usercopy on kmalloc mechanism disabled (*nousercopy*). In the following, we use *full* as the default setting. The results show that in more than $\frac{3}{4}$ of the LMbench tests Safeslab exhibits either negligible ($< 5\%$) or no slowdown at all, with a geometric mean (*geomean*) of 4%. Note that Safeslab's geomean is $\approx 5x$ smaller than ViK's 21%, a recent UAF-mitigation solution for the Linux kernel based on pointer tagging [21]. The worst-case overhead incurred by Safeslab is 50% on stat, 24% on open/close, and 18% on 'select (500 fds)'. In order to better understand the nature of the overhead on these tests, we further analyzed the behavior of the system via the tracing tools perf [28] and BCC [49].

stat's execution mainly takes place in the kernel's syscalls, where it allocates a single object from the names object cache, performs a number of operations on it, and then frees it. The test repeatedly executes this sequence over and over again, causing no interference in-between subsequent allocations, which triggers the best-case scenario in SLUB: subsequent allocations always return the same memory object to the caller. This benefits SLUB in two ways: first, the first memory access of the object user yields a *cache-hit*, as its RAM entry is already in the data-cache thanks to prior accesses; second, SLUB reuses the same slab to serve the subsequent of (de-)allocations, without having to invoke the buddy allocator for allocating new slabs, or freeing them.

In contrast, as Safeslab quarantines freed objects for neutralizing dangling pointers, it never returns the same memory address on subsequent allocations—this inflicts more cache-misses on stat compared to SLUB. We confirmed this by tracing execution via perf and measuring the hardware *Performance Monitoring Counters (PMCs)*—indeed, Safeslab inflicted 100x more cache-misses than SLUB on stat. Furthermore, as it does not reuse freed slab slots, Safeslab also invokes the buddy allocator much more often, despite allocating a larger chunk of pages for each slab (4x more in the default configuration). In fact, using BCC, we measured that SLUB only invoked buddy for allocating and freeing slabs 10x, while Safeslab did it for ≈45000x on the stat benchmark. We observed a similar behavior on 'select (500 fds)'.

On open/close Safeslab and SLUB invoked the buddy allocator the same number of times (measured via BCC). Safeslab exhibited 60x more cache misses compared to SLUB (measured via perf). This is because with every slab allocation request, Safeslab always gets a fresh, unused slab (since used ones are quarantined), while SLUB receives slabs whose entries are already in the data cache.

On the 'fork+"/bin/sh"' test, where Safeslab does not exhibit any slowdown, Safeslab invokes the buddy allocator ≈700x, while SLUB does it for only ≈70x. Additionally, Safeslab inflicts only ≈4x more cache-misses than SLUB. This can be justified by the fact that the payload is more heterogeneous—it works with a larger variety of object types, it runs on multiple threads, and it spends more time doing user space work (≈4% of its time is spent in-kernel)—this causes more interference between subsequent heap allocations, which inflicts more cache-misses in SLUB. Moreover, the test itself allocates ≈10x less objects on 'fork+"/bin/sh"' compared to stat (i.e., ≈34K vs. ≈300K object allocations/deallocations).

We also performed additional experiments to test whether invoking the buddy allocator less often reduces the overhead. Specifically, we reran the LMbench tests with Safeslab configured to allocate 8x more pages than SLUB would for a slab, but we observed no difference in performance compared to Safeslab's default setting of 4x more pages. This hints that Safeslab's main source of overhead is attributed to the higher rate of cache-misses on subsequent allocations. Additionally, as Figure 3 shows, Safeslab's extended usercopy feature generally induces between 1% and 4% overhead on a fraction of the benchmarks, and 1% overall with a geomean of 3%. This is because of the additional bookkeeping required when the kernel copies data from user space into kmalloc'd objects, as Safeslab must keep track of the offsets where they are stored within the object. The other tests remain unaffected, as they copy less data from user space into the kernel's kmalloc caches.

6.1.2 Phoronix Test Suite. The stress-tests from LMbench often trigger the best-case scenario in SLUB (see previous section), which Safeslab does not benefit from. However, this is rarely met in real-world, end-to-end workloads. For example, LMbench works with a small set of objects, and does not even utilize all of the available CPU cores to their full potential, as only a few of them are active on tests that stress the scheduler (such as 'fork+"/bin/sh"'). Moreover, most LMbench tests execute small payloads that interact with a small (sometimes only one) object caches in the kernel, and they do not exercise the other critical components of Safeslab, i.e., TAD switching, slab migration, and the scanner.

We therefore conducted additional experiments via several macro-benchmarks from the Phoronix Test Suite (PTS) [84]. Figure 4 shows the results when our extension is used vs. the baseline (i.e., unmodified SLUB). Safeslab encounters the worst-case overhead of 30% on hackbench, a scheduler stress-test that keeps all CPU cores occupied. In all other tests, Safeslab exhibits negligible ($<$ 3%) or no slowdown at all. In order to better understand the nature of the overhead on hackbench, we further analyzed its behavior via the tracing tools perf [28] and BCC [49]. hackbench heavily exercises the kernel's heap allocator (≈1.8M object allocations/-deallocations; ≈92.5% of its time is spent in-kernel), during which Safeslab exhibits ≈2x more cache-misses than SLUB.

Additionally, even in the 4x more pages setting, Safeslab invokes the buddy allocator 100x more often than SLUB on hackbench. To determine how much this contributes to Safeslab's overhead, we tested whether allocating 8x more pages per slab reduces slowdown on hackbench—this is not the case, however. Moreover, during hackbench, Safeslab performed 59 domain switches and 4 scans, triggered ≈5200 page faults, (i.e., page migrations across different TADs), and during handling these page faults it had to split huge 2MB-pages into 4K pages ≈3750x. To test the extent to which these events contribute to the overhead, we reran hackbench with Safeslab's scanner and the PK protection disabled—however, the overhead is the same. This means, again, that Safeslab's main source of overhead are cache-misses. In the case of hackbench, Safeslab's extended usercopy feature adds a 4% performance penalty, which is similar to the few affected tests described in the previous section, and because of the same reasons.

By comparison, on compile-linux, where Safeslab only exhibits a negligible performance overhead of 2%, although the benchmark exercises the allocator ≈50x more than hackbench (≈110M object allocations/deallocations), Safeslab and SLUB exhibit the same amount of cache-misses, while Safeslab invokes the buddy allocator 20x more often than SLUB. This is because compile-linux works with a larger variety of kernel objects, while its tasks keep getting migrated across CPUs, and spends more time doing user-space work (≈12.25% of its time is spent in-kernel)—this leads to more cache-misses in SLUB. During this benchmark, Safeslab performed 14 TAD switches, 1 scan, ≈6000 page faults (i.e., TAD migrations), and ≈5900 huge-page breaks. These however did not seem to increase Safeslab's performance overhead in our experiments.

Furthermore, to asses the impact of the pointer scanner on user-space applications we reran the PTS benchmarks with all Safeslab features enabled, except for the pointer scanner, and observed that even on I/O intensive benchmarks its overhead is negligible: ≈3% on hackbench (≈92.5% of its time is spent in-kernel), ≈2% on apache (≈88.5% of its time is spent in-kernel), ≈1% on nginx (≈40% of its time is spent in-kernel), 0% on compile-linux (≈12.25% of its time is spent in-kernel). That this is due to the scanner's policy of allowing user-mode tasks to continue running even during the scan, while only blocking those that enter kernel mode. Moreover, we analyzed the (false positive) dangling pointers found by the scanner during 10 runs of each of said benchmarks, and counted the amount of pages unmapped and remapped by Safeslab as part of its strategy for reducing false positives (see Section 4.7). Table 2 summarizes our results. Although a fairly large average number of page (un-)mappings are performed on every scan, Safeslab's performance did not seem to be affected. This is attributed to the fact that scans occur rarely thanks to the MPK-isolated TADs.

6.2 Memory Overhead

We distinguish between two main sources of memory overhead in Safeslab: *static*, which stems from additional structures defined at compile- or boot-time, and *dynamic*, which is introduced at runtime.

6.2.1 Static Memory Overhead. Safeslab requires additional memory for storing pages that map the TADs in the PTs (see Section 4.2), which varies with the size of a TAD, the number of TADs configured, the address-mapping granularity (i.e., 4K, 2MB, or 1GB translations), and the number of PT translation levels (i.e., 4-level or 5-level). We conducted our experiments on hardware that supports 2MB mappings with 4-level paging enabled, where one single 4K page maps 256TB of virtual memory in the first layer of the PT, 512GB in the second layer, and 1GB in the third layer. As such, we devise the following formula for determining the amount of additional memory required by Safeslab to map TADs: $\lceil t \cdot s + (\frac{t \cdot s}{512}) \rceil$,

Benchmark	max RSS	total alloc'ed	total freed
Boot	2.6x (110 MB)	15.9x	40x
Build Linux	1.1x (13.5 MB)	122x	130x
Hackbench	2.3x (55 MB)	360x	311x
Nginx	19.5x (71.5 MB)	698x	705x
Redis	6.5x (11.2 MB)	54.5x	55x
SQlite	62x (135 MB)	277.5x	277.5x
Apache	21x (584 MB)	1832x	2184x

Table 1: Memory consumption observed during PTS benchmarks. Values represent the number of pages compared to baseline (`SLUB`). The second column also shows in parentheses the difference between `Safeslab` and baseline in MB.

Bench	# scans	unmapped	remapped
Hackbench	40	5014 (20MB)	718 (3MB)
Apache	24	15105 (59MB)	3432 (14MB)
Nginx	24	7343 (29MB)	3208 (13MB)
Build-Linux	10	10176 (40MB)	2618 (11MB)
Total	98	9410 (37MB)	2495 (10MB)

Table 2: Number of unmapped and remapped pages (false positives) found by the pointer scanner during 10 runs. The results are averaged over the number of scans.

where s is the size of a TAD in GiB and t is the amount of TADs configured by `Safeslab`. In our experiments we configured the size of a TAD to be equal to the size of the machine's available RAM, i.e., 64GB, and we created 15 TADs at boot time, since that is the number of PKs provided by Intel (in addition to the default one).

All in all, `Safeslab` required 96 pages for mapping the TADs, which amounts to 4MB. In reality, the actual overhead is slightly less than this, since we do not map in the TADs the memory regions that correspond to immutable sections, such as the `.rodata`, `.ro_after_init`, and the code segment(s). On systems that support 1GB mappings, the overhead becomes even lower, since the PTs can use only two levels for translating TADs—in such a setting, the memory overhead would be reduced to only two 4K pages. Additional memory consumption for maintaining virtual memory aliases is a common trade-off made by other solutions that adopt a similar approach [39, 106], and is not specific to `Safeslab`.

In addition, `Safeslab` also introduces auxiliary global data for managing the quarantine, for keeping track of per-CPU TADs, and for the pointer scanner—in total, `Safeslab` requires less than one 4K page for its global data. Furthermore, the scanner spawned at boot-time introduces 6.4KB for its `task_struct`.

6.2.2 Dynamic Memory Overhead. Due to its security-oriented design, `Safeslab` adds memory overhead in the following ways: (1) a larger number of allocated pages, which may become only sparsely occupied since freed slots are not reused; (2) quarantined memory pages that cannot be reused by the system in order to avoid UAF scenarios; (3) memory pages marked as unsafe cannot be reused by `Safeslab` in the TADs for which dangling pointers to them exist; (4) additional `usercopy` metadata for keeping track `kmalloc`'d regions that contain user data (see Section 5); (5) additional 64-byte metadata objects for all pages used by `Safeslab`; (6) large PT mappings (i.e., 2MB) that are split into smaller ones (4KB) due to alias migrations across TADs.

To evaluate (1) and (2) we monitor `Safeslab`'s memory consumption during selected benchmarks from PTS, which trigger many object (de-)allocations in the kernel. During each test, we record the benchmark's *maximum resident set size* (max RSS, measured in pages), and the total number of allocated and freed pages throughout the entire benchmark. Table 1 shows the results. We encountered the worst-case memory overhead on SQLite and Apache, where `Safeslab`'s RSS was 62× and 21× larger than `SLUB`'s, which amounted to 135MB and 584MB of additional memory, respectively. This is due to large fragmentation on live slabs, whose empty slots are *not* reused by `Safeslab` in order to mitigate dangling pointers.

Moreover, `Safeslab` (de-)allocated on Apache many more pages than `SLUB`, overall, leading to two pointer scans (i.e., two domain switches)—this means that it reached its 4GB quarantine threshold twice. SQLite works with much smaller objects than Apache, and hence did not trigger any domain switch. However, as it uses large memory objects that fill up the quarantine much faster, `hackbench` allocated and freed most memory overall on `Safeslab`, leading to 59 domain switches—thus, the unusable quarantined memory reached the worst-case of 4GB for 59×. Overall, `Safeslab` (de-)allocated much more memory than `SLUB`—this is due to its *no-reuse* approach. Nevertheless, memory overhead stemming from quarantining is also exhibited by other solutions that use a similar approach [1].

We also measured `Safeslab`'s RSS after boot and compared it to ViK's [21]. `Safeslab` used 159% more memory than baseline, while ViK used 42.5%—this is due to `Safeslab`'s no-reuse approach, which puts more presure on memory than ViK's pointer-tagging approach. ViK did not conduct memory overhead experiments on PTS, and so we cannot further compare our results with it. Moreover, ViK was implemented on Linux v4.14, while `Safeslab` on v6.2, thus making the porting process of one to the other a non-trivial task.

We evaluated (3) by counting the number of pages unmapped and remapped by the pointer scanner during 10 runs of the I/O-intensive benchmarks described in Table 2—the experiment triggered ≈100 scans. The results show that `apache` triggers most false positives, leading to 15K unsafe pages on average in-between scans, amounting to 59MB. These pages however are only unusable in the TADs for which dangling pointers were found, but can be used by the other TADs, as well as by the other kernel subsystems and user-space applications. The results also show that on average, in the `apache` benchmark, around 2432 pages (14MB) loose their dangling references in-between scans, allowing `Safeslab` to remap and reuse their pages in their respective TADs.

For (4), we measured the amount of additional `usercopy` metadata maintained by `Safeslab` for `kmalloc` caches after boot, and determined that it needs ≈5MB for storing them. Similarly for (5), after boot, `Safeslab` required ≈7.5MB to store the per-page metadata objects that are used to keep track of their safety status. In terms of (6), during the performance experiment with `hackbench` described in Section 6.1.2, `Safeslab` had to split 2MB-pages into 4K ones 3,750×, which increased memory consumption by ≈15MB; on `compile-linux`, `Safeslab` had to split 5,900 huge-pages into small ones, consuming 23MB of additional memory.

6.3 Security Analysis

6.3.1 SLUB. In what follows we summarize the security properties of the SLUB allocator (in Linux, as of v6.2) with respect to UAF, DF, IF, and OOB exploitation scenarios.

Use-After-Free. SLUB has no dedicated mechanism for detecting or mitigating UAFs in the presence of dangling pointers. These are mainly abused by attackers to overlap security-sensitive objects onto victim objects, with the ultimate goal of manipulating their contents—we observed this "pattern" in all public kernel exploits surveyed in Section 6.4. Attackers achieve this via one of the two different approaches (a) and (b) described in Section 2.2, by manipulating SLUB's freelist to overlap memory objects, or the buddy allocator to overlap memory pages, respectively. Although SLUB makes freelist manipulation harder by randomly *shuffling* the freelist when the slab is initialized, techniques such as *heap spraying* [114] are typically leveraged by attackers to bypass it.

Double-Free. SLUB only catches DFs when the address cached at the top of the freelist matches the freed address. However, attackers can easily circumvent this by freeing a dummy object in-between the two free invocations on the affected object.

Invalid-Free. SLUB's security checks against IFs are only executed when a freelist is demoted from the per-CPU slab. Note, however, that this is only performed if object caches are explicitly configured to do so; this feature is not enabled by default for every cache.

Out-Of-Bounds. Attackers can use OOB accesses against freed objects in SLUB to corrupt their freelist pointers, as they are stored inline, and use them to build primitives for exploiting the system. Although, SLUB mangles the freelist pointer (by XOR'ing it with a secret key) before storing it in objects, the protection is weak and has been repeatedly circumvented by attackers to further build exploitation primitives [16–18, 63].

6.3.2 Safeslab. In the following we juxtapose the security properties of Safeslab with respect to UAF, DF, IF, and OOB scenarios.

Use-After-Free. Safeslab mitigates both approaches that attackers may use to exploit UAFs in the presence of dangling pointers in the kernel—i.e., (a) and (b) described in Section 2.2. It defeats (a) by not reallocating slab objects once they are freed—this is exactly the reason why Safeslab does not maintain a freelist. It mitigates (b) by quarantining the memory pages that form slabs when they get freed, so that the buddy allocator returns unused memory pages when allocating new slabs. Therefore, there is no way for an attacker to manipulate Safeslab or the buddy allocator to reallocate an object or a slab so that it overlaps with a dangling pointer.

We only reallocate quarantined pages after deactivating their access rights for the PK that corresponds to their old TAD, thus blocking any dangling pointers they may have. Although dangling pointers in active TADs are still accessible, they cannot be used to overlap security-critical objects and only have a considerably limited use as an exploitation primitive. Moreover, we have not encountered any exploit in our security evaluation (see Section 6.4) that leverages such a primitive. Safeslab recycles TADs once they get depleted, in order to avoid exhausting the virtual address space.

To do that safely, it employs the *pointer scanner* to mark as unsafe all memory pages for which dangling pointers exist in the kernel. Then, it discards all pages that are marked as unsafe when allocating new pages from the buddy allocator to form new slabs. We prevent dangling pointers from escaping the scanner by stopping the tasks that execute in kernel mode during the scan.

Although the scanner currently cannot detect dangling pointers that are hidden, e.g., via obfuscation, to the best of our knowledge, such pointers are not a common construct in the Linux kernel (except for SLUB's mangled freelist pointers, which are obsolete in Safeslab). Moreover, Safeslab proactively marks as unsafe those quarantined slabs that might be reallocated before the scanner can guarantee that they have no dangling pointers (see Section 4.8). We also prevent attackers from abusing the *page migration* mechanism to activate dangling pointers that are blocked by PKs, by only migrating an accessed memory page if the faulting address references a valid, allocated slab, and their TADs match.

Double-Free. We mitigate DFs via the bitmap that keeps track of each slab object's allocation status. Namely, when an object is freed its bitmap entry is set; however, if the entry was already set then Safeslab detects the DF attempt.

Invalid-Free. We prevent IFs by ensuring on *every* free that the freed address: (a) is freed to the object cache from which it belongs, (b) references a valid object slab, (c) is aligned to the size of its cache's object size, and (d) does not point to an object that has not been allocated yet.

Out-Of-Bounds. Safeslab does not store metadata within corruptible slab objects, but rather in segregated regions, outside an attacker's reach [74]. Therefore, attackers cannot use in-slab OOB vulnerabilities to corrupt its metadata for building exploitation primitives. However, as we do not employ any dedicated memory isolation mechanism [83], attackers may target Safeslab's metadata through (arbitrary) memory R/W primtives obtained via other means. Furthermore, we do not employ any dedicated mechanism for preventing OOB accesses against general object data—this is outside the scope of our work (see Section 3).

6.4 Real-world Exploits

We surveyed 30 known exploits (targeting SLUB) against use-after-free CVEs found in the Linux kernel (between 2016 and 2023), and determined that Safeslab can mitigate all of them [2–8, 15, 31, 35–38, 44, 51, 63, 64, 67, 68, 70, 80, 85, 86, 96, 98, 101, 103, 116]. They all rely on SLUB's weakness to re-allocate sensitive objects after they are freed, which allows attackers to manipulate their contents via dangling pointers. These exploits are ineffective against Safeslab, as we do not re-allocate objects at the same virtual address unless the scanner found no dangling pointers to them. Additionally, we also analyzed 6 known exploits that abuse OOB-related CVEs in the Linux kernel to target allocator metadata (such as freelist pointers), and determined that Safeslab is not affected by them [41, 63, 72, 77, 81, 104]. This is because, unlike SLUB, which keeps allocator metadata within slab objects, Safeslab does store it in segregated memory, where attackers cannot manipulate it. Nevertheless, note that Safeslab does not mitigate OOB bugs on other types of sensitive data, such as function and/or data pointers.

7 DISCUSSION AND FUTURE WORK

Memory Consumption. Safeslab requires additional memory for pre-configuring TADs (see Section 6.2), which is however on par with other solutions that adopt a similar *no-reuse* approach [1, 39, 106]. Nevertheless, our PT mappings for TADs scale with their number, their size, and the number of PT translation levels, which are fixed at compile time (see Section 4.2). With 1GB huge-pages enabled, the TAD mappings could be realized with just two 4KB pages. Additionally, by either configuring a smaller number of TADs, or giving them a smaller size of RAM to map, the memory overhead can be reduced further. Moreover, Safeslab's memory overhead introduced by quarantining could be reduced by engineering the allocator to allow the system to use quarantined pages for other purposes, such as user applications or the stack. However, access to the page's TAD aliases must be disabled in the PT before reusing it, in order to prevent dangling pointers from accessing it.

MPK Security. If an attacker obtains code execution by exploiting spatial memory errors, then Safeslab can be bypassed using the techniques identified in prior work [23, 87, 102]. However, this is currently a limitation of *all* solutions that focus solely on mitigating temporal memory errors [21, 39, 90], and is not specific to Safeslab or MPK. Nevertheless, if future work aims at tackling this limitation, CFI solutions could be used in Safeslab to prevent attackers from executing wrpkru/xrstor instructions to bypass MPK—recent CFI improvements demonstrated that effective solutions with low-overhead are possible [32].

If efficient CFI solutions are unavailable, then techniques leveraged by prior work on securing MPK-sandboxes could be adopted in Safeslab—prior work demonstrated that secure MPK-callgates can be designed in a performant manner [43]. Note that, since Safeslab runs in kernel mode, future attacks cannot abuse syscalls to bypass MPK in the same fashion as prior work on bypassing MPK-sandboxing in user space does; however, these attacks, and the proposed mitigations should be considered if adopting Safeslab in user space. In addition, isolating Safeslab's sensitive metadata (such as page tables, the quarantine, or the struct page metadata), could prevent data-only attacks—this could be achieved by MPK itself, which prior work [71] demonstrated that incurs a worst-case $\approx$ 9% performance cost on PTS macro-benchmarks (however, it protected more kernel objects than those used by Safeslab).

Pointer Scanner. We did not find the pointer scanner to be a critical source of overhead (see Section 6.1.2), which is attributed to its infrequent execution due to Safeslab's TADs. Still, certain applications might not tolerate being blocked in kernel mode even for a short time window. Such cases can be addressed by leveraging the io_uring feature [56], which allows processes to register syscalls asynchronously and continue user-space work until the syscall is serviced. Additionally, a capability flag could be introduced in Safeslab (e.g., CAP_SAFESLAB_NOBLOCK) that system administrators can grant to whitelisted tasks, allowing them to invoke the kernel without blocking during Safeslab's scan. Moreover, although user-supplied dangling pointers might get copied out of kmalloc'ed objects, Safeslab could be extended to keep track of a task's PID, and kill that task if the amount of dangling pointers it induces exceeds a certain threshold.

Compatibility. Safeslab relies on MPK to prevent access to disabled TADs (i.e., aliases), which may also be achieved via other hardware features. For example, memory virtualization could provide Safeslab with a higher number of TADs (e.g., 512) [83] at the cost of higher runtime/memory overhead due to resource virtualization and more expensive domain switches [45]. Moreover, as virtualization controls (guest) physical addresses, implementing Safeslab's TADs will require re-configuring both the PTs and EPTs, which might further impact performance. Security-wise, virtualization also relies on an unprivileged instruction to switch domains, i.e., VMFUNC, making it susceptible to the same security issue as MPK (see "MPK Security" above). Other hardware primitives that have been repurposed for intra-process isolation, such as Intel CET [110] or SMAP [74] may not be well-suited for Safeslab, as they define an unprivileged domain that can also be accessed by the privileged one, thus failing to prevent access to disabled TADs.

Portability. Safeslab is compatible with other types of allocators, such as those used in user space [107]. However, as it runs in kernel mode, Safeslab benefits from direct access to the PTs and privileged instructions, which speeds-up execution. For a user-space solution, Safeslab needs to rely on system calls for performing privileged operations, such as modifying PTs when migrating pages across TADs, which also require flushing the TLB of the executing CPU. However, as page migrations are rare, we do not expect this to cause high performance overhead—Intel MPK is available in user space (see Section 2.3) and could be used when switching TADs. Moreover, Safeslab could also borrow some of the PKs for protecting other memory regions against temporal errors, such as the vmalloc region, and even against arbitrary R/W—both on its own memory-management metadata (e.g., struct page objects) [10, 74] or other sensitive data (e.g., PTs [83]). However, this may come at a greater performance penalty in Safeslab, since the pointer scanner will have to execute more frequently.

8 RELATED WORK

Most prior solutions that mitigated temporal errors were designed for user-space applications, and porting them to kernel space, or alternatively porting Safeslab to user space, to perform an experimental comparison, would be non-trivial. Nevertheless, we provide in the following a juxtaposition.

Memory Quarantining. MineSweeper [29] and MarkUs [1] also quarantine freed objects until a certain threshold is hit, and perform a pointer scan before reusing them. However, they must invoke the pointer scanner every time their quarantine is full, which happens 15x more often than with Safeslab—thanks to TADs, we only do this when we run out them. Nevertheless, as they do not leverage multiple TADs, which require additional pages to store PT mappings, these techniques might benefit from less memory overhead than Safeslab. Quarantining techniques with *probabilistic guarantees* [78, 82, 91, 92] release quarantined objects in random order during execution, making it harder for attackers to predict when a target freed object becomes available. Such techniques usually fall short against attackers that can execute object (de-)allocations arbitrarily (e.g., via heap spraying), which is usually the case in real-world exploits. Safeslab does not rely anything probabilistic.

Address Aliasing. Existing solutions that mitigate temporal errors via address aliasing [27, 39, 106] disable used aliases via page table manipulations, which is a costly operation that requires several accesses in memory to update the corresponding PT entry, as well as a TLB shootdown on all CPUs. Although it is unclear how much TLB shootdowns affected performance, we believe they can have a devastating impact in the kernel when done on every object deallocation. On the contrary, Safeslab disables used aliases via Intel MPK that is fast and does not require updating TLB entries, and only resorts to PT manipulations for disabling dangling pointers (including false positives), which is a rare occurrence. DangZero [39] and Oscar [27] associate each physical object with multiple virtual aliases, which puts pressure on the available virtual addresses—as the kernel works with a large number of objects, we expect such an approach to exhaust the address space much faster, which, in turn, would require scanning for dangling pointers more frequently (to facilitate alias reuse). However, DangZero's PT compression, combined with the fact that is does not quarantine memory, might lead to less memory overhead than Safeslab.

Pointer Tagging. Pointer tagging solutions instrument most memory accesses with an additional check that validates whether a pointer is allowed to access the referenced object [21, 50, 62, 75, 117]. However, such solutions typically rely on fetching additional metadata during the check, which require several lookups in memory, incurring significant slowdowns. Furthermore, techniques based on hardware extensions for tagging pointers either require substantial hardware modifications (i.e., CHERI [105]), or are probabilistic (such as ARM's MTE [9]) and can be bypased due to tag collisions (16 tags available). In contrast, Safeslab only entails a performance penalty on the first memory access of a newly allocated object, while subsequent memory accesses are unaffected (geomean Safeslab 5% vs. ViK 21%; see Section 6.1.1), and provides deterministic security guarantees. However, pointer tagging metadata might incur less memory overhead(s) than approaches based on no-reuse.

Pointer Tracking. UAF mitigation techniques that keep track of dangling pointers to invalidate [66, 100, 113] or count them [90] rely on static analysis to instrument all instructions that manipulate memory pointers. Such solutions are known to be incomplete since accurately identifying all instructions that manipulate pointers at runtime cannot be guaranteed in weakly-typed programming languages such as C and ASM. Additionally, keeping track of memory locations that store pointers may considerably increase memory consumption for pointer-rich systems (e.g., the Linux kernel), while updating the pointers' metadata for every instruction that creates, modifies, or destroys them, may lead to significant performance overhead due to additional memory accesses. In contrast, Safeslab does not rely on static analysis to provide its UAF mitigation, and the additional memory accesses for updating its metadata are far less frequent, as demonstrated by its low overhead.

Type Isolation. Solutions that avoid reusing virtual addresses across different types of objects, in user [34, 99] and kernel [46, 47] space, *pin* a range of virtual memory to each object class and always reuse it for new allocations. Nevertheless, such techniques require (un-)mapping the pinned virtual memory in the PT when getting new/releasing old physical pages, which hinders performance. Moreover, they only mitigate UAFs across different object classes, while UAFs within the same class are still possible. In contrast, Safeslab prevents UAFs in both cases.

Intel MPK. Intel MPK has been mainly used in prior research for sandboxing untrusted code within a process and preventing it from tampering with sensitive memory [22]. Many MPK-based solutions isolate sensitive data in user-space applications [45, 79, 97, 102], while others explored sandboxing untrusted microkernel drivers, in kernel mode [43], or untrusted unikernel drivers [93]. Although these solutions are effective against attackers that *already poses* arbitrary R/W primitives in the address space, Safeslab mitigates temporal memory errors (which are used to build such primitives) via Intel MPK—this has not been addressed by prior work.

9 CONCLUSION

In this paper we presented Safeslab, a heap hardening extension that mitigates temporal memory errors in the presence of dangling pointers. It leverages the novel concept of temporal aliasing domains, implemented via Intel MPK, to reuse quarantined memory with new virtual aliases quickly, and an infrequent pointer scanner to reuse freed virtual addresses safely. We implemented Safeslab atop the default heap allocator in the Linux kernel (SLUB) and showed in our evaluation that it can mitigate several existing UAF exploitation techniques. Additionally, we demonstrated that its performance overhead on realistic benchmarks is negligible, while its memory consumption is moderate.

AVAILABILITY

Our Safeslab prototype is available at: https://github.com/tum-itsec/safeslab

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 awards CNS-2238467, CNS-2104148, and CNS-1749895. 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.

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 2020. MarkUs: Drop-In Use-After-Free Prevention for Low-Level Languages. In *IEEE Symposium on Security and Privacy (S&P)*. 578–591.
[2] Alejandro Guerrero. 2022. N-day Exploit for CVE-2022-2586: Linux Kernel nft_object UAF. https://www.openwall.com/lists/oss-security/2022/08/29/5.
[3] Alexander Popov. 2017. Race for Root: Analysis of the Linux Kernel Race Condition Exploit. https://program.sha2017.org/system/event_attachments/attachments/000/000/111/original/a13xp0p0v_race_for_root_SHA2017.pdf.
[4] Alexander Popov. 2019. CVE-2019-18683: Exploiting a Linux Kernel Vulnerability in the V4L2 Subsystem. https://a13xp0p0v.github.io/2020/02/15/CVE-2019-18683.html.
[5] Alexander Popov. 2021. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.
[6] Andrey Konovalov. 2017. CVE-2016-2384: Exploiting a double-free in the Linux kernel USB MIDI driver. https://xairy.io/articles/cve-2016-2384.
[7] Andrey Konovalov. 2017. CVE-2017-6074: Exploiting a Double-Free in the Linux Kernel DCCP Sockets. https://xairy.io/articles/cve-2017-6074.

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

[9] Joe Bialek, Ken Johnson, Matt Miller, and Tony Chen. 2020. Security Analysis of Memory Tagging. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Securityanalysisofmemorytagging.pdf.

[10] William Blair, William Robertson, and Manuel Egele. 2022. MPKAlloc: Efficient Heap Meta-data Integrity Through Hardware Memory Protection Keys. In *International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 136–155.

[11] Hans-J Boehm, Alan J Demers, and Scott Shenker. 1991. Mostly Parallel Garbage Collection. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 157–164.

[12] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer Technical Conference*. 87–98.

[13] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel*. 294–350.

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (SEC)*. 249–266.

[15] Cedric Halbronn. 2022. 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/.

[16] Silvio Cesare. 2020. An Analysis of Linux Kernel Heap Hardening. https://blog.infosectcbr.com.au/2020/04/an-analysis-of-linux-kernel-heap.html.

[17] Silvio Cesare. 2020. Bit Flipping Attacks Against Free List Pointer Obfuscation. https://blog.infosectcbr.com.au/2020/04/bit-flipping-attacks-against-free-list.html.

[18] Silvio Cesare. 2020. Weaknesses in Linux Kernel Heap Hardening. https://blog.infosectcbr.com.au/2020/03/weaknesses-in-linux-kernel-heap.html.

[19] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A Systematic Study of Elastic Objects in Kernel Exploitation. In *ACM Conference on Computer and Communications Security (CCS)*. 1165–1184.

[20] Yueqi Chen and Xinyu Xing. 2019. SLAKE: Facilitating SLAB Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *ACM Conference on Computer and Communications Security (CCS)*. 1707–1722.

[21] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: Practical Mitigation of Temporal Memory Safety Violations Through Object ID Inspection. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 271–284.

[22] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P. Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection against Native Node.js Add-ons. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*. 429–442.

[23] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium (SEC)*. 1409–1426.

[24] Jonathan Corbet. 2020. Memory protection keys for the kernel. https://lwn.net/Articles/826554/.

[25] Jonathan Corbet and Alessandro Rubini. 2001. Linux Device Drivers, Second Edition. https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch07s04.html.

[26] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 161–175.

[27] Thurston HY Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *USENIX Security Symposium (SEC)*. 815–832.

[28] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. 2024. Linux Kernel Profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial.

[29] Márton Erdős, Sam Ainsworth, and Timothy M Jones. 2022. MineSweeper: a "Clean Sweep" for Drop-In Use-After-Free Prevention. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 212–225.

[30] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security Symposium (SEC)*. 1037–1054.

[31] Flat Security Inc. 2021. CVE-2021–20226: A Reference Counting Bug which Leads to Local Privilege Escalation in io_uring. https://flattsecurity.medium.com/cve-2021-20226-a-reference-counting-bug-which-leads-to-local-privilege-escalation-in-io-uring-e946bd69177a.

[32] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 527–546.

[33] GNU libc. 2024. malloc. http://man7.org/linux/man-pages/man3/malloc.3.html.

[34] Google. 2024. PartitionAlloc Design. https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md.

[35] Google Project. 2018. A Cache Invalidation Bug in Linux Memory Management. https://googleprojectzero.blogspot.com/2018/09/a-cache-invalidation-bug-in-linux.html.

[36] Google Security Research. 2023. CVE-2023-0461. https://github.com/google/security-research/tree/master/pocs/linux/kernelctf/CVE-2023-0461_mitigation/docs.

[37] Google Security Research. 2023. CVE-2023-3390. https://github.com/google/security-research/tree/master/pocs/linux/kernelctf/CVE-2023-3390_lts_cos_mitigation/docs.

[38] Google Security Research. 2023. 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.

[39] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2022. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *ACM Conference on Computer and Communications Security (CCS)*. 1307–1322.

[40] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. 2021. Fast Intra-kernel Isolation and Security with IskiOS. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 119–134.

[41] GRIMM Cyber. 2021. New Old Bugs in the Linux Kernel. https://blog.grimm-co.com/2021/03/new-old-bugs-in-linux-kernel.html.

[42] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (ESSoS)*. 161–176.

[43] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *USENIX Annual Technical Conference (ATC)*. 401–417.

[44] Hardened Linux. 2016. Exploiting on CVE-2016-6787. https://hardenedlinux.github.io/system-security/2017/10/16/Exploiting-on-CVE-2016-6787.html.

[45] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process Isolation for High-throughput Data Plane Libraries. In *USENIX Annual Technical Conference (ATC)*. 489–504.

[46] Jann Horn. 2022. MITIGATION_README. https://github.com/thejh/linux/blob/slub-virtual/MITIGATION_README.

[47] Apple Inc. 2022. Towards the next generation of XNU memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/.

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

[49] iovisor. 2024. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[50] Jann Horn. 2020. Mitigating (Some) Use-After-Frees in the Linux Kernel. https://lssna2020.sched.com/event/c74I/mitigating-some-use-after-frees-in-the-linux-kernel-jann-horn-google.

[51] javierprtd Blog. 2020. 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/.

[52] jemalloc. 2024. jemalloc. https://jemalloc.net.

[53] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. EPF: Evil Packet Filter. In *USENIX Annual Technical Conference (ATC)*. 735–751.

[54] Di Jin, Alexander J Gaidis, and Vasileios P. Kemerlis. 2024. BeeBox: Hardening BPF against Transient Execution Attacks. In *USENIX Security Symposium (SEC)*.

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

[56] Jonathan Corbet. 2020. The Rapid Growth of io_uring. https://lwn.net/Articles/810414/.

[57] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium (SEC)*. 957–972.

[58] The Linux Kernel. 2024. Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html.

[59] The Linux Kernel. 2024. Physical Memory Model. https://docs.kernel.org/mm/memory-model.html.

[60] Kenneth C Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* (1965), 623–624.

[61] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*. 1–19.

[62] Mathias Krause. 2022. Canary in the Kernel Mine: Exploiting and Defending Against Same-Type Object Reuse. https://grsecurity.net/exploiting_and_defending_against_same_type_object_reuse.

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

[64] Lam Jun Rong. 2022. io_uring – New Code, New Bugs, and a New Exploit Technique. https://www.starlabs.sg/blog/2022/06-io_uring-new-code-new-

bugs-and-a-new-exploit-technique/.
[65] Christoph Lameter. 2014. Slab Allocators in the Linux Kernel: SLAB, SLOB, SLUB. https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf.
[66] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-After-Free With Dangling Pointers Nullification. In *Network and Distributed System Security Symposium (NDSS)*.
[67] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *ACM Conference on Computer and Communications Security (CCS)*. 1963–1976.
[68] Lin Ma. 2021. Blue Klotski (CVE-2021-3573) and the Story for Fixing. https://f0rm2l1n.github.io/2021-07-23-Blue-Klotski/.
[69] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory From User Space. In *USENIX Security Symposium (SEC)*. 973–990.
[70] Lizzie Dixon. 2017. Notes about CVE-2016-7117. https://blog.lizzie.io/notes-about-cve-2016-7117.html.
[71] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. 2023. DOPE: DOmain Protection Enforcement with PKS. In *Annual Computer Security Applications Conference (ACSAC)*. 662–676.
[72] Maxime Peterlin, and Philip Pettersson, and Alexandre Adamski, and Alex Radocea. 2020. Exploiting a Single Instruction Race Condition in Binder. https://www.longterm.io/cve-2020-0423.html.
[73] Larry W McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference (ATC)*. 279–294.
[74] Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Michalis Polychronakis, and Vasileios P. Kemerlis. 2024. ISLAB: Immutable Memory Management Metadata for Commodity Operating System Kernels. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*.
[75] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *International Symposium on Memory Management (ISMM)*. 31–40.
[76] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 47–62.
[77] Vitaly Nikolenko. 2016. CVE-2016-6187: Exploiting Linux Kernel Heap Off-by-One. https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit.
[78] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *ACM Conference on Computer and Communications Security (CCS)*. 573–584.
[79] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference (ATC)*. 241–254.
[80] Patryk Sondej and Piotr Krysiuk. 2023. CVE-2023-32233: Privilege Escalation in Linux Kernel due to a netfilter nf_tables Vulnerability. https://www.tarlogic.com/blog/cve-2023-32233-vulnerability/.
[81] Alexander Popov. 2017. Race for Root: The Analysis Of the Linux Kernel Race Condition Exploit. https://media.ccc.de/v/SHA2017-295-race_for_root_the_analysis_of_the_linux_kernel_race_condition_exploit.
[82] Alexander Popov. 2020. Linux Kernel Heap Quarantine Versus Use-After-Free Exploits. https://a13xp0p0v.github.io/2020/11/30/slab-quarantine.html.
[83] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*. 563–577.
[84] PTS. 2024. Phoronix Test Suite. https://www.phoronix-test-suite.com.
[85] Querijn Voet. 2023. CVE-2023-3389 – LinkedPoll. https://qyn.app/posts/CVE-2023-3389/.
[86] Ruihan Li. 2023. StackRot (CVE-2023-3269): Linux Kernel Privilege Escalation Vulnerability. https://github.com/lrh2000/StackRot.
[87] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium (SEC)*. 936–952.
[88] SecWiki. 2020. Linux Kernel Exploits. https://github.com/SecWiki/linux-kernel-exploits.
[89] SecWiki. 2021. Windows Kernel Exploits. https://github.com/SecWiki/windows-kernel-exploits.
[90] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. 2019. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *Network and Distributed System Security Symposium (NDSS)*.
[91] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *ACM Conference on Computer and Communications Security (CCS)*. 2389–2403.
[92] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A Tunable Secure Allocator. In *USENIX Security Symposium (SEC)*. 117–133.
[93] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-unikernel Isolation with Intel Memory Protection Keys. In *ACM International Conference on Virtual Execution Environments (VEE)*. 143–156.
[94] The Linux Kernel. 2024. Memory Protection Keys. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html.
[95] The Linux Kernel. 2024. Unaligned Memory Accesses. https://www.kernel.org/doc/html/next/core-api/unaligned-memory-access.html.
[96] Theori Vulnerability Research. 2022. Linux Kernel Exploit (CVE-2022–32250) with mqueue. https://blog.theori.io/linux-kernel-exploit-cve-2022-32250-with-mqueue-a8468f32aab5.
[97] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK). In *USENIX Security Symposium (SEC)*. 1221–1238.
[98] Valentina Palmiotti. 2022. Put an io_uring on it: Exploiting the Linux Kernel. https://www.graplsecurity.com/post/iou-ring-exploiting-the-linux-kernel.
[99] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Annual Computer Security Applications Conference (ACSAC)*. 17–27.
[100] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-After-Free Detection. In *European Conference on Computer Systems (EuroSys)*. 405–419.
[101] Vincent Dehors. 2021. 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.
[102] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing. In *European Conference on Computer Systems (EuroSys)*. 266–282.
[103] Vu Thi Lan. 2023. 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/.
[104] Wang, Yong. 2019. From Zero to Root: Building Universal Android Rooting with a Type Confusion Vulnerability. https://github.com/ThomasKing2014/slides/blob/master/Building%20universal%20Android%20rooting%20with%20a%20type%20confusion%20vulnerability.pdf.
[105] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)*. 20–37.
[106] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *USENIX Security Symposium (SEC)*. 2453–2470.
[107] Wolfram Gloger. 2006. ptmalloc. http://www.malloc.de/en/.
[108] Nicolas Wu. 2024. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.
[109] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *USENIX Security Symposium (SEC)*. 781–797.
[110] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In *ACM Conference on Computer and Communications Security (CCS)*. 2989–3002.
[111] Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and Their Mitigations. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–36.
[112] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. KRACE: Data Race Fuzzing for Kernel File Systems. In *IEEE Symposium on Security and Privacy (S&P)*. 1643–1660.
[113] Yves Younan. 2015. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Network and Distributed System Security Symposium (NDSS)*.
[114] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2022. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *USENIX Security Symposium (SEC)*. 71–88.
[115] Google Project Zero. 2022. The More You Know, The More You Know You Don't Know. https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.
[116] Zhenpeng Lin. 2023. 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.
[117] Jie Zhou, John Criswell, and Michael Hicks. 2023. Fat Pointers for Temporal Memory Safety of C. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 316–347.