SeaK: Rethinking the Design of a Secure Allocator for OS Kernel

Zicheng Wang[†‡], *Yicheng Guang*[‡], *Yueqi Chen*[†],
Zhenpeng Lin[§], *Michael Le*[¶], *Dang K Le*[§],
Dan Williams[⋆], *Xinyu Xing*[§], *Zhongshu Gu*[¶], *Hani Jamjoom*[¶],
[†]*University of Colorado Boulder,* [‡]*Nanjing University,*
[§]*Northwestern University,* [¶]*IBM Research,* [⋆]*Virginia Tech*
[†]*{zicheng.wang, yueqi.chen}@colorado.edu,* [‡] *guangyichengok@gmail.com,*
[§]*{zplin, dang.le, xinyu.xing}@u.northwestern.edu,*
[¶]*{mvle, zgu, jamjoom}@us.ibm.com,* [⋆]*djwillia@vt.edu*

Abstract

In recent years, heap-based exploitation has become the most dominant attack against the Linux kernel. Securing the kernel heap is of vital importance for kernel protection. Though the Linux kernel allocator has some security designs in place to counter exploitation, our analytical experiments reveal that they can barely provide the expected results. This shortfall is rooted in the current strategy of designing secure kernel allocators which insists on protecting every object all the time. Such strategy inherently conflicts with the kernel nature.

To this end, we advocate for rethinking the design of secure kernel allocator. In this work, we explore a new strategy which centers around the "atomic alleviation" concept, featuring flexibility and efficiency in design and deployment. Recent advancements in kernel design and research outcomes on exploitation techniques enable us to prototype this strategy in a tool named `SeaK`. We used real-world cases to thoroughly evaluate `SeaK`. The results validate that `SeaK` substantially strengthens heap security, outperforming all existing features, without incurring noticeable performance and memory cost. Besides, `SeaK` shows excellent scalability and stability in the production scenario.

1 Introduction

In the Linux kernel, exploitation targeting heap vulnerabilities such as use-after-free, heap out-of-bound write and read, and uninitialized heap is very prevalent. In the Google and Alphabet Vulnerability Reward Program held in 2022 [36], 42 revealed Linux kernel exploits are all against the kernel heap. In the Pwn2Own contest since 2020, all nine showcased Linux kernel exploits target the kernel heap, with awards reaching up to $300k. In addition, from a variety of sources including industry summits like BlackHat and personal blogs of famous whitehat hackers (*e.g.*, [1, 9]), we collected public exploits, PoCs, and write-ups over the past five years, and found that 143 out of 173 Linux kernel exploits are against the kernel heap. Hence, securing the heap is of paramount importance for kernel protection.

Though there are security features in place in the Linux kernel to counter heap-based exploitation, they can barely provide the expected protection. On one hand, the features that are enabled by default in most distros are overly specific to certain exploitation techniques, making them bypassable. Consider freelist randomization as an example: by design, it only works for exploits leveraging spatial corruption, like heap out-of-bound write and read, and falls short in defending against temporal corruption, such as use-after-free. On the other hand, the features that are disabled by default have intrinsic weaknesses and fail to deliver the promised security improvements. To enumerate, structure layout randomization faces challenges in securely storing the random seed, while, based on our experiments, KFENCE rarely achieves its goal of separating kernel objects.

In parallel, in the user space, the design of secure allocators has been well researched [19, 45, 57, 59], introducing features like red zones, poisoning, user tracking, and ad-hoc sanity checks. These features can be found in a kernel mechanism named slub_debug. Unfortunately, slub_debug is regarded primarily as a debugging feature and serves as the building block for sanitizers like KASAN, due to its high cost.

In this work, we conducted a multi-faceted measurement of existing security features, revealing the fundamental obstacle in designing a kernel secure allocator: the allocator is the core kernel subsystem and is invoked with high frequency. Given the current strategy in security feature design, which insists on protecting every object all the time [51], performance and memory overhead accompany each allocation and free. This level of overhead is unacceptable for the kernel as it must offer services for user space with high efficiency but does not have unlimited memory.

Since the obstacle is intrinsic to the kernel nature, we advocate for rethinking the design of kernel secure allocators. In this work, we explore a new strategy which centers around the "atomic alleviation" (AA) concept. One AA offers the most granular level of exploit alleviation by separating a specific type of kernel object. We can orchestrate particular sets of AAs to meet distinct security needs, focusing only on critical

objects instead of every object indiscriminately. Besides, the enforcement and retirement of AAs do not bother to recompile and reboot the kernel, supporting continuous protection upgrading.

This strategy had been previously infeasible until recent advancements in the extended Berkeley Packet Filter (eBPF) [29, 35, 49, 54, 66, 71, 72]. Using eBPF, we prototyped this new strategy in a tool named `SeaK`, representing **Se**cure **a**llocator for **K**ernel. We further developed techniques that can automatically identify allocation and free sites of objects selected for separation in a certain scenario, and synthesized eBPF programs that construct separation at runtime. The separation is shipped with guard pages and up to 43 entropy randomization.

This strategy wouldn't be reliable security-wise without the outcomes of research on exploitation techniques in recent years. We applied `SeaK` to two scenarios to illustrate how it enhances kernel heap security. Given that not all objects are worth protecting, recent works [21, 22, 65] have been focusing on identifying security-sensitive objects in the Linux kernel. Advancements in this direction pave the way for deploying `SeaK` to separate and protect these critical objects. In another scenario where the kernel patching process is lengthy, we showcase how `SeaK` can separate corruptions introduced by N-day vulnerabilities, thereby protecting the kernel from potential exploitation before patches are available.

We evaluated `SeaK` in terms of security improvement, performance and memory overhead, and scalability, by using real-world cases. Our experimental results validate that `SeaK` can effectively thwart state-of-the-art kernel heap exploitation, even the newest DirtyCred attack [44], outperforming existing security features. In the meanwhile, the performance overhead of a single AA is negligible - less than 1% on average when separating the most frequently used object; and the memory overhead is not noticeable when separating the most durable object. `SeaK` is scalable as there is no obvious increase in performance overhead and memory overhead when more than 64 AAs are orchestrated.

To our knowledge, `SeaK` is the first practical secure allocator in open-source kernels. `SeaK` has been deployed on the authors' machine, having supported daily research and education activities for 2.5 months, and running stably to date. In summary, this work makes the following contributions:

- A multi-faceted measurement to disclose the inherent obstacles of designing a secure allocator for OS kernel.
- The introduction of a new and practical strategy to secure kernel heap and its core concept - atomic alleviation.
- Open-source design and implementation of the strategy in `SeaK` which is the first practical secure kernel allocator. `SeaK` can remediate vulnerability before patches are available.
- A comprehensive evaluation of `SeaK` which validates the effectiveness and efficiency of the new strategy.

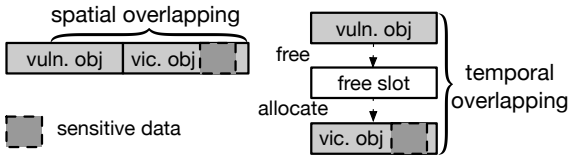

Figure 1: Kernel heap exploitation which creates overlapping between spatial/temporal memory corruption from the vulnerable object and sensitive data in the victim object.

2 Background

In this section, we will describe the heap management and heap exploitation techniques in the Linux kernel.

2.1 Kernel Heap Management

In the Linux kernel, the term of heap often refers to the memory located in the direct mapping region which has 64TB in total and maps all physical memory. The heap is coordinately managed by the buddy allocator and the SLAB/SLUB allocator, which roughly work as follows.

Buddy Allocator. The buddy allocator partitions physical memory into pages and manages objects larger than one page. During allocation, if the requested size (*e.g.*, 2 pages) is unavailable, a larger chunk (*e.g.*, 8 pages) is repeatedly halved (*e.g.*, 8 pages to 4 pages to 2 pages) until a chunk of the exact size is produced. The two halves are referred to as buddies: one satisfies the allocation request and the other remains free. When the allocated buddy is later freed and its counterpart buddy is not used, they will be merged back and reclaimed (*e.g.*, two 2-page buddies back to one 4-page chunk).

SLAB/SLUB Allocator. The SLAB/SLUB allocator manages small objects by acquiring pages from the buddy allocator. These pages are known as slab caches and are divided into slots - each slot aims to hold one object. Objects stored in the same slab cache either share the same type or have a similar size. When slots are free, they are linked in a singly-linked list called the freelist. The freelist works in a LIFO (Last In, First Out) fashion: retrieving the first slot in the freelist for allocation and adding the slot back to the beginning of the list during free. If all slots in these pages are freed, the SLAB/SLUB allocator returns these pages to the buddy allocator.

2.2 Kernel Heap Exploitation

In recent years, exploitation techniques against the kernel heap have been thoroughly discussed in both academia [21, 22, 67, 68, 70] and industry [38]. Due to the space limit, we won't elaborate on concrete exploits but instead summarize the common ideas behind various exploitation techniques.

Create Overlapping. Figure 1 illustrates an essential step in exploitation which is to create overlapping between heap corruption and sensitive data such as function pointers and credentials. The overlapping can be categorized into spatial overlapping and temporal overlapping according to the nature of vulnerabilities. To exploit heap out-of-bound write/read, attackers manipulate heap layout [22,68] to place victim objects that contain sensitive data adjacent to the vulnerable object. By triggering the vulnerability, the corruption has a spatial overlapping with the sensitive data, allowing attackers to tamper with it and achieve IP control or privilege escalation. To exploit use-after-free, attackers first free the vulnerable object which still has a dangling pointer referring to it. The freed slot is recycled back to SLAB/SLUB allocator. Then, attackers spray victim objects containing sensitive data to reclaim the same memory, leveraging the LIFO feature. By dereferencing the dangling pointer, attackers tamper with sensitive data in the victim object and obtain exploitable primitives.

Cross-cache Exploitation. If the victim object and the vulnerable object are in the same slab cache, it is often referred to as within-cache exploitation. Cross-cache exploitation stands for the situation where the victim object and the vulnerable object are in different slab caches. To exploit a heap out-of-bound write/read in a cross-cache exploitation manner, attackers manipulate the heap layout at the buddy allocator level, ensuring that two pages — the one containing the victim object and the other containing the vulnerable object are adjacent though they belong to distinct caches. To exploit a use-after-free in a cross-cache exploitation manner, attackers first free all vulnerable objects within the same cache so that the cache is reclaimed to the buddy allocator. Then, they allocate a number of victim objects, forcing the buddy allocator to re-halve the just reclaimed slab cache pages - previously storing vulnerable objects - for storing victim objects instead. Thus, there is a temporal overlapping between the vulnerable and victim objects.

Same-type Exploitation. In most exploits, regardless of within-cache or cross-cache, the vulnerable object and the victim object are of distinct types so the difference between their semantics allows successful tampering. However, the newest DirtyCred attack [44] demonstrates that vulnerable objects and victim objects can be of the same type in exploitation. Technically, two objects of the same type (*e.g.*, `struct cred` but carrying different privilege levels (*e.g.*, `cred.uid == 1000` and `cred.uid == 1000`) can be overlapped to achieve privilege escalation. This attack bypasses all existing security features in the kernel allocator, even slub_debug.

3 Obstacles in Existing Designs

The newest security features in the Linux kernel allocator are in three categories: (C1) features that are enabled by default in mainstream Linux distros such as Ubuntu and CentOS; (C2) features that are designed for protection but are disabled by default in most distros; (C3) features that are commonly employed in user space secure allocators and integrated into slub_debug [37].

In this section, we will evaluate the security and overhead of these features, followed by investigation into the particular obstacle in secure kernel allocator design.

3.1 Security Analysis

By-default Enabled Features (C1). In this category, ❶ freelist randomization [26] randomizes the order of slots in the freelist of the SLAB/SLUB allocator so that attackers can hardly accurately predict the slab cache layout. ❷ The freelist obfuscation [27] aims to hinder attackers from manipulating the allocator into returning a memory under the attackers' control. ❸ The heap zeroing [52] on allocation initializes the slot during allocation so that attackers cannot read sensitive information belonging to the object that previously occupied the slot. These features raise the bar of exploitation but are too specific to certain vulnerabilities and attack techniques and thus can be easily bypassed [2–4, 7, 10]. Taking freelist randomization as an example, it is designed only against vulnerabilities that cause spatial overlapping like out-of-bound write, and doesn't work for vulnerabilities that cause temporal overlapping like use-after-free. Even for spatial overlapping, it can be bypassed through heap grooming [39].

By-default Disabled Features (C2). In this category, ❶ structure layout randomization [25] shuffles the field orders in structures each time the kernel boots up so that attackers cannot predict the offset between sensitive data and the start of corruption. However, the kernel must reveal the random seed to support compiling third-party kernel modules. How to securely store the seed continues to be a challenge nowadays [33]. Besides, it only randomizes structures that contain only function pointers and cannot cover all structural types. ❷ KFENCE allocates objects from a pre-reserved pool. Each object takes pages and is surrounded by red zones and guard pages, which can detect out-of-bound access. When the object is freed, the corresponding page is unmapped so that use-after-free can also be detected. However, to reduce overhead, KFENCE randomly samples objects for separation, no matter whether the object is security-related or not. Further, the sampling only happens in the fast path of allocation and it only samples one object in a certain time window. Thus, in our experiment, KFENCE can only protect 0.005% - 0.35% sensitive objects even after its capability is maximized. Features in this category are designed for protection but fall short of providing assured security improvement, presumably the reason why they are disabled by default.

Lightweight "Debugging" Features (C3). This category includes ❶ slub_debug which is primarily regarded as a debugging feature. It provides a full spectrum of security fea-

tures typically found in userspace secure allocators: red zones around objects enabled by Z flag, poisoning and user tracking of freed memory by P and U flags respectively, and additional sanity checks by F flag. Compared with KASAN, slub_debug is lightweight but for protection, it is too heavy. We will show this in the following.

3.2 Overhead Measurement

Benchmarks and Settings. We use two benchmarks to evaluate the overhead of existing security features. One is LMbench [47] which is a micro-benchmark widely used for system call level measurement. Another is Phoronix Test Suite [11] which is a macro-benchmark. It runs real-world applications and we use it to measure the impact of `SeaK` on the overall system. In particular, from Phoronix Test Suite, we choose seven applications: OpenSSL, 7zip-compress, FFmpeg, Redis, SQLite, and Apache as representatives of different workloads to comprehensively test processor, OS, and system.

Our experiments were conducted using a bare-metal machine running Ubuntu 22.04 LTS with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and 16GB RAM. We built a vanilla kernel image as the baseline using the Linux kernel v5.15 - the latest Long-term support (LTS) version at the time of experimentation. The vanilla kernel doesn't contain any security features described in this section. Then, we built another three images: one with all three features in C1, one with the two features in C2, and one with slub_debug in C3. To minimize fluctuation and rule out outliers, we repeatedly ran benchmarks until the overhead of the recent five executions had a coefficient of variation smaller than 3.5% - a default setting in Phoronix. Between each round, we rebooted the whole system to ensure a clean environment for the benchmark.

Performance Overhead. Table 1 presents the performance overhead. Though a recent study [55] reveals that freelist randomization in C1 showcases 45% peak overhead in certain situations (*i.e.*, big-select and big-fork) because of poor locality, our measurement over both LMbench and Phoronix indicates that the overall overhead of C1 is not obvious: -2.74% to 3.91% which is within reasonable fluctuation range. For the two features in C2 that are disabled by default, their overhead is slightly noticeable: -1.88% to 5.25% for LMbench, and -1% to 1.34% for real-world applications in Phoronix. While this extent of overhead is tolerable, as we discussed in the security analysis (Section 3.1), both features in C2 have inherent weaknesses that prevent them from producing the expected secure enhancement.

For slub_debug in C3, its overhead is prohibitively high, reaching 177.22% for LMbench and 57.58% for Phoronix. Note that, in the table, some benchmarks (*e.g.*, 7zip-compress) show negative overhead because their execution time predominantly takes place in user space with minimal kernel involvement. These negative numbers shouldn't be mis-interpreted as indicating that slub_debug has no overhead. In fact, for benchmarks that are dependent on kernel services, the overhead arises significantly.

LMbench	**C1**	**C2**	**C3**
Simple syscall	0.35%	1.06%	0.90%
Simple read	0.98%	3.73%	0.70%
Simple write	0.41%	1.71%	2.46%
Select on 100 fd's	-0.64%	1.21%	0.04%
Signal handler install	-1.35%	-1.88%	-1.17%
Signal handler overhead	0.75%	3.29%	169.16%
fork+exit	0.60%	1.76%	168.17%
fork+execve	2.42%	1.56%	177.22%
fork+/bin/sh -c	1.21%	2.32%	151.55%
UDP latency	3.91%	4.97%	144.34%
TCP/IP connection	-2.74%	5.25%	129.81%
AF_UNIX bandwidth	-0.20%	0.27%	52.16%
Pipe bandwidth	0.80%	1.16%	-1.98%
Phoronix	**C1**	**C2**	**C3**
Sockperf (Msgs/sec)	-0.27%	-0.61%	57.58%
OSBench (Ns/Event)	-0.08%	-1.00%	6.25%
7-Zip Compress (MIPS)	-0.34%	0.54%	-0.39%
FFmpeg Live (FPS)	-0.14%	0.28%	1.25%
OpenSSL SHA256 (B/s)	0.01%	0.04%	0.01%
Redis SET (Reqs/sec)	-0.37%	0.47%	0.55%
SQLite Speedtest (sec)	0.52%	1.34%	4.05%
Apache 100 (Reqs/sec)	-0.50%	-0.42%	46.29%

Table 1: Performance overhead of security features in different categories. See Section 3.1 for more details of each category.

Memory Overhead. Figure 2 presents the memory overhead when running LMbench (Phoronix's result are moved to [18] due to space limit). From the figure, we can observe that vanilla, C1, and C3 exhibit similar memory usage while C2 consumes an additional 400 MB of memory. This is because KFENCE in C2 allocates objects from a pre-reserved pool and each object separated by KFENCE occupies pages that are surrounded by red zones and guard pages.

3.3 Behind Security / Overhead Trade-off

Summarizing from security analysis and overhead measurement, existing security designs in the Linux kernel allocator either fail to provide substantial security enhancement (C1 and C2) or are impractical due to excessive memory and performance overhead (C2 and C3). To have a deeper understanding of the root obstacle, we selected Apache from Phoronox - a case with noticeable overhead - as our target and conducted a case study.

Table 2 presents the statistics collected by bpftrace during the execution of Apache. From the table, we can observe that,

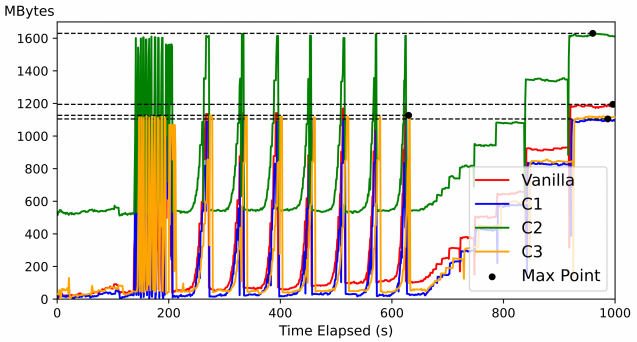

Figure 2: Increase of memory usage of existing security features when running LMbench.

	Vanilla	C1	C2	C3
Inst. # (million)	1,661,099	1,641,186	1,673,330	2,708,439
Invoked #/sec	324,194	321,347	321,634	214,793
Time Proportion	17.1%	17.1%	17.7%	41.4%
Time Per Req (μs)	17.8	17.7	17.7	33.2

Table 2: Statistics collected through the Apache Benchmark in five minutes. "Inst. # (million)" indicates the # of instructions executed within the kernel, represented in millions. "Invoked #" is the # of times the allocator is called. "Time Proportion" is the fraction of time spent within the allocator inside the kernel. "Time Per Req" averages time required to handle a request.

in a given time, the kernel with security features executes a substantially higher number of instructions, particularly for C3 which exhibits a 63.1% increase compared to vanilla. This is attributed to C3's most complete spectrum of security features, which introduces numerous additional instructions to the kernel allocator. Further, we observed that the kernel allocator is invoked very frequently, reaching 214,793 to 324,194 times per second. As a result, in C3, the time proportion dedicated to the allocator doubles, jumping from 17.1% to 41.4%. This bloating not only decelerates the kernel, as evidenced by the reduced number of allocator invocations in a given time, but also impacts the entire system, leading to longer response times for Apache to handle a request.

Conclusion. Based on our analytical experiments, we have the following conclusion. First, the allocator is a core subsystem in the kernel and is frequently used. Second, each time the kernel invokes the allocator, it incurs the overhead associated with executing the additional code introduced by security features. Given these facts, the current strategy of protecting every object all the time can hardly work within the kernel which must offer services for user space efficiently.

4 Design Overview

Drawing on lessons learned from evaluating existing features, a more realistic strategy would be conserving resources on critical objects rather than protecting every object indiscriminately. We explored this new strategy and prototype it in the design of `SeaK`. In this section, we will first describe the eBPF ecosystem on top of which we build `SeaK`, followed by the threat model and the design overview.

4.1 The eBPF Ecosystem

The extended Berkeley Packet Filter (eBPF) is an in-kernel virtual machine that allows **privileged** users to run programs in the kernel space. The eBPF programs are written in the C language and compiled to bytecode using the LLVM eBPF backend. Privileged users can install the compiled eBPF programs into the kernel and attach them to arbitrary instructions, monitoring and modifying kernel behaviors.

The **safety** of eBPF programs is guaranteed by a static verifier which establishes three properties: (1) memory safety, ensuring that the program only accesses pre-defined memory locations, (2) information flow security, ensuring that no secret kernel state is exposed, and (3) all execution must terminate. Over decades of development, eBPF has significantly improved in **expressiveness**. The eBPF helper functions allow the installed eBPF programs to interact with other kernel subsystems. The BPF maps can store arbitrary data. They allow eBPF programs to communicate with each other and with privileged userspace processes by looking up and updating the stored data through keys. For the sake of **efficiency**, a variety of optimization techniques has been adopted in the eBPF ecosystem. For instance, the eBPF bytecode is executed by a Just-In-Time (JIT) engine rather than a slow interpreter to achieve native-machine code level performance. Further, the instructions once attached, are overwritten to `call` or `jmp` instructions instead of the previously used `int3` interrupt. As such, the time spent on switching context to eBPF programs is saved.

eBPF for Security. The eBPF ecosystem has undergone swift evolution across commodity OS kernels, including Linux, Windows, FreeBSD, and macOS [5, 6]. In the security area, PET [66] instruments eBPF programs to error sites, preventing kernel vulnerabilities from being triggered. Sifter [32] filters malicious syscall with eBPF to make attack surfaces in security-critical kernel modules unreachable. RapidPatch [31] allows RTOS developers to hot-patch embedded device firmware using eBPF.

4.2 Threat model

`SeaK` has the following assumptions about the capabilities of attackers and defenders.

Attacker. Attackers possess a vulnerability that corrupts the kernel heap. This vulnerability can be exploited using the newest kernel exploitation techniques. It can be present in any kernel subsystem, including the eBPF subsystem on top

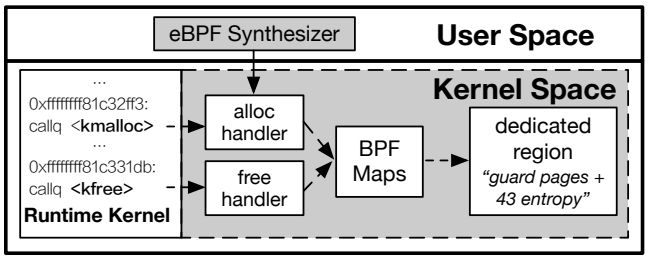

Figure 3: Overview of one atomic alleviation (AA) in SeaK. The eBPF program synthesizer in the userspace installs an eBPF program with alloc handlers and free handlers into the kernel space, separating objects of interest.

```
1  // essential utilities
2  int alloc_handler(struct pt_regs* ctx,
3      u64 kpi_type) {...}
4  int free_handler(struct pt_regs* ctx,
5      u64 kpi_type) {...}
6
7  SEC("kprobe/?") // attach to allocation site
8  int probe_alloc_? (struct pt_regs* ctx) {
9      return alloc_handler(ctx, ?);
10 }
11 SEC("kprobe/?") // attach to free site
12 int probe_free_? (struct pt_regs* ctx) {
13     return free_handler(ctx, ?);
14 }
```

Listing 1: The snippet of synthesis template. Once the "?" is filled in, it can be directly compiled and installed into the kernel space.

of which SeaK is built, as long as the eBPF programs can be installed. Further, we assume attackers are unprivileged users so that they cannot directly disable SeaK.

Defender. Defenders are privileged users or system administrators so that SeaK is granted the right privilege to install eBPF programs into the kernel. The installed eBPF programs are free from bugs and their safety is guaranteed by the sound verified. We assume that defenders are aware of the vulnerability threat by either catching its exploit in the wild or obtaining necessary information (*e.g.*, vulnerability reports) from public resources including the dashboard of Syzkaller – the most widely used kernel fuzzer developed by Google, the National Vulnerability Database, and more.

4.3 SeaK at a Glance

The **atomic alleviation** (AA for short) is the key concept in SeaK. One AA is responsible for separating a specific type of kernel objects - objects of interest. Figure 3 illustrates the design of one AA in SeaK.

In user space, the eBPF synthesizer first analyzes the kernel source code to pinpoint the allocation and free sites for objects of interest. Then, it investigates debug information to map the source-code-level sites into corresponding binary addresses of `call` instructions that invoke memory allocation and free functions (*e.g.*, `kmalloc/kfree`). After this, the synthesizer produces an eBPF program that can be installed into kernel space to achieve alleviation.

In the kernel space, the eBPF program shoulders a complete allocation and free logic for objects of interest: The alloc handler attached to the allocation sites intercepts the original allocation and obtains memory from a dedicated region that is separated; the free handler attached to the free sites prevents the memory in dedicated regions from being directly returned to the buddy system and recycles memory only for objects that are allowed by the security policy. The status of dedicated regions is maintained by BPF maps. If one dedicated region runs out of space, additional memory will be assigned. When the threat landscape changes and the AA is no longer useful or requires updates, the eBPF program can exit gracefully and be reinstalled later if needed again.

Like secure allocators in user space (*e.g.*, [19,45,57,59]), to prevent spatial overlapping, the dedicated region is equipped with guard pages and randomization with up to 43 entropy. To prevent temporal overlapping, SeaK currently enforces the most restrictive separation policy to deal with the newest exploitation techniques (Section 2.2): only recycle within objects that are allocated from the same site, having the same size, carrying the same privilege level, and in the same zone.

Advantages. SeaK is flexible from the following perspectives. ❶ Design-wise, one AA offers the most granular level of alleviation. We can strategically orchestrate different sets of AAs to meet distinct security needs. It allows efficient use of resources by focusing on crucial objects, rather than an indiscriminate separation. ❷ Deployment-wise, AAs can be enabled on the fly without disrupting running computation services, thus maintaining system availability. ❸ Evolution-wise, The separation policy enforced in AA can be dynamically upgraded when new exploitation techniques are disclosed. In comparison, the implementation of existing solutions is fixed once integrated.

5 Technical Details

In this section, we will present more technical details of AA in SeaK, from the eBPF synthesizer in the user space to the runtime separation in the kernel space.

5.1 eBPF Program Synthesis

To separate objects of interest, an eBPF program is generated. The essential elements needed to construct the eBPF program include the binary addresses of the allocation site and free site, as well as the prototype of the kernel function that is called for allocation or free. Readers can refer to List 2 in Appendix for the illustration of a synthesized program.

Synthesis Template. List 1 shows the template SeaK uses to synthesize eBPF programs. All the "?" in the template represent the elements that need to be customized per site. In line 7 and line 11, SEC decoration denotes the locations of the allocation and free sites in the format of func+offset. In this format, func is the symbol of the kernel function where allocation and free are called, offset represents the offset of the call instruction from the start of func. We use func+offset instead of the absolute address of the call instruction (*i.e.*, 0xffffffff81c331db) because of Kernel Address Space Layout Randomization (KASLR) [28] which randomizes the base address of kernel image during boot time. Using func+offset, the eBPF program can work across machines with different base addresses without further modification.

In lines 8 and 12, the "?" serves as a placeholder for a unique identifier to differentiate multiple allocation sites and free sites. In line 9 and 13, the "?" is used to differentiate prototypes of allocation and free functions being called. This Kernel Programming Interface (KPI) information (kpi_type) is needed so that the eBPF programs (line 1-5) can determine how to obtain the requested size of the allocation and the address of object to be freed from the run-time context. For example, we differentiate kmalloc and kmem_cache_alloc because the allocation size is stored in the 1st parameter of kmalloc but in the 2nd parameter of kmem_cache_alloc, passed through $rdi and $rsi, respectively, by the x64 convention.

Determining Allocation and Free Sites. Given the object type, SeaK finds the allocation sites and free sites by first searching at the source code level and then converting the sites into the corresponding binary addresses in the format of func+offset.

At the source code level, the SLAB/SLUB allocator uses two series of kernel functions for allocation and free. One is kmalloc/kfree series for objects in the general cache. The other is kmem_cache_alloc/kmem_cache_free series for special cache. For the buddy system, the functions for allocation and free are alloc_pages/free_pages series. The code lines that call these functions are allocation and free sites. We further narrow down the scope and identify the sites specific to the objects of interest by analyzing the return values or arguments of these function calls. Because their return values or arguments are always pointers referencing the objects allocated or freed, by analyzing them, we can easily figure out whether the object is of interest. Technically, we perform a use-def analysis and resolve memory alias. Along the use-def chain of returned value or arguments, we track instructions relevant to typecasting, pointer dereferencing, and argument passing. The operands of these instructions explicitly reveal the type of variables. By using this information, we can easily infer and conclude the type of each allocated or freed object.

At the binary level, sites in the source code that allocate or free objects of interest are mapped to binary addresses via debugging information in the kernel image. Note that, kmalloc/kfree series functions at the source code level are occasionally inlined into kmem_cache_alloc/kmem_cache_free series at the binary level through compiler optimization. Therefore, we synthesize the kpi_type part in the eBPF programs according to the series used in the kernel image rather than the source code to eliminate inaccuracies.

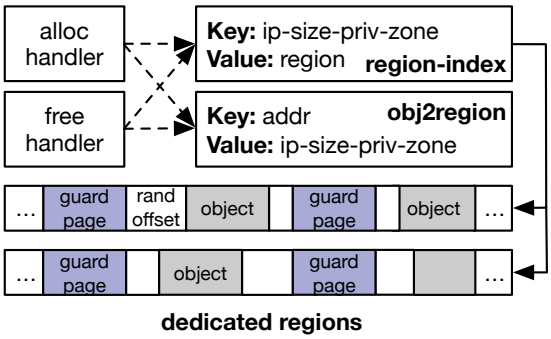

Figure 4: Run-time separation in the kernel space. The attached malloc handlers and free handlers leverage two BPF maps to manage dedicated regions.

5.2 Run-time Separation

Here, we delve into more details of how eBPF programs and BPF maps separate objects of interest.

Data Structures. As illustrated in Figure 4, in one AA, an alloc handler is attached to each allocation site, and a free handler is attached to each free site. The synthesizer described in Section 5.1 is responsible for identifying these sites and synthesizing the corresponding handlers.

To track the status of dedicated regions that store separated objects, each AA has two BPF maps: region-index and obj2region. The region-index map is used to locate dedicated regions. Its key is customized according to the security policy. Its value represents the dedicated region which is the address of either a struct kmem_cache object if the requested size is smaller than one page or a struct page object if the requested size is larger than one page. The key of the obj2region map is the address of an individual object that is separated into dedicated regions. Its value indicates which region the object belongs to and can be used to index the region-index map.

Evolving with Exploitation Techniques. In Figure 4, we use ip-size-priv-zone as the key to index the dedicated region. As such, the memory is recycled only within objects that are allocated from the same site (*i.e.*, ip), with the same size, carrying the same privilege level, and in the same zone. It is the most restrictive policy that can alleviate the newest exploitation techniques that have been disclosed thus far. As new exploitation techniques evolve in the future, this policy can be strengthened accordingly.

Guard Page and Randomization. Each separated object occupies multiple pages. Even for the object the real size of which is smaller than one page, it takes over at least one page. These pages are further surrounded by guard pages that are not mapped. The offset of the object from the start of the pages is randomized. The randomization entropy within the page is 7, considering pointer alignment in x64. Since the dedicated region can be located anywhere in the direct mapping area, the overall entropy is 21 for the 16 MB DMA zone or 43 for the 64TB normal zone, which is much higher than the widely-used KASLR which is 8. Given such a high entropy, the chance for attackers to brute-force the address of separated objects in one shot is nearly zero.

Workflow. Now, we describe how the aforementioned designs are used through the life cycle of an object that is separated.

When the object is allocated, the alloc handler will be executed. According to the security policy, it obtains the execution context: (1) the address of allocation sites from `$rip` register, (2) the request size from `$rdi` for `kmalloc`, `$rsi` for `kmem_cache_alloc`, `$rsi*PAGE_SIZE` for `alloc_pages`. (3) the privilege level of the current process through the helper function, (4) the requested zone which is specified in GFP flags - `$rsi` for `kmalloc`, `$rdx` for `kmem_cache_alloc`, `$rdi` for `alloc_pages`. This context information is concatenated to form the key. The alloc handler uses the key to look up the region-index map to examine if there is already a dedicated region that fits. If no dedicated region is available, the alloc handler creates a new slab cache or a new buddy through the helper function and records related information in the region-index map. Otherwise, the alloc handler retrieves the slab cache or buddy structure from the region-index map, storing the object and setting guard pages through help functions. Following this, the address of the object is used as the key to update the obj2region map. Finally, the alloc handler overwrites the return address to directly jump to the next instruction of allocation `call` and thus skip the original allocation.

When the object is freed, the free handler will be executed. It first obtains object's address from the `$rdi` register and looks it up in the obj2region map to determine if the object to separated. If not, the kernel continues the routine free operation. Otherwise, the free handler looks up the region-index map, obtains the dedicated region, and frees the object from the region using a helper function. Note that, this free operation does not indeed return the memory. Instead, the memory is recycled for the subsequent allocations of objects with the same `ip-size-priv-zone` key, as the security policy requires.

Concurrency and Exit. Since objects of interest can be allocated and freed at many kernel sites, the alloc handlers and free handlers attached to these sites will concurrently access the shared region-index and obj2region maps. Therefore, it is essential to ensure the atomicity of map read and write. Fortunately, the current eBPF ecosystem provides a spin lock per key and a read-copy-update (RCU) counter per value in BPF maps to achieve concurrency control.

When one AA is no longer needed or a stronger policy is developed, it exits gracefully without causing memory leakage. More specifically, `SeaK` creates a kernel task that periodically scans the entire kernel memory to check for pointers referring to the separated objects indexed in the obj2region map. If no more pointers refer to the separated objects, the occupied memory can be safely returned to the buddy system. Otherwise, the AA stays in the kernel until the obj2region map becomes empty.

6 Application

In this section, we showcase how to orchestrate a set of AAs to meet security requirements in different scenarios.

6.1 Separating Security-Sensitive Objects

Given not all objects are security-sensitive, protecting each one indiscriminately is unnecessary. Kernel objects such as `cred`, `msg_msg`, and `key_payload` are well-known to be security-sensitive because they contain data like credentials and function pointers. Separating them is one key task in prior works, including xMP [53], kalloc_type [30], AutoSlab [40], and slab_virtual [64]. `SeaK` can complete this task in a straightforward way: each AA separates one specific sensitive type.

A series of research efforts have been undertaken to identify security-sensitive objects in the Linux kernel: SLAKE [22] collects objects with function pointers; ELOISE [21] focuses on elastic objects that can provide stronger write and read primitives; AlphaExp [65] finds objects based on within-cache exploitation scheme. As exploitation techniques evolve, more objects will be identified as security-sensitive. By constructing and installing more AAs, `SeaK` can easily adapt to these newly discovered objects. This flexibility is a significant advantage over prior works because their design and implementation are fixed. Moreover, `SeaK` can enforce separation on the fly while prior works require recompilation and rebooting which disrupts system availability.

6.2 Separating Vulnerability Corruptions

Due to the lack of manpower and design complexity, the patching process is quite lengthy for the Linux kernel. A study two years ago [61] revealed that the average patching window in the Linux kernel was 66 days, and the situation is getting worse [8]. During this time window, patches are not available for these N-day vulnerabilities and attackers have the full freedom to develop exploits and launch attacks.

To mitigate this threat, we can construct AAs to separate corruptions introduced by vulnerabilities, thereby limiting the damage. For vulnerabilities such as heap out-of-bound write and read, corruption happens when there is beyond-boundary

access. One AA can be installed to separate the overflowed object. Thus, guard pages can catch the spatial overlapping caused by the vulnerability, and due to randomization, the overlapping cannot precisely tamper with targeted kernel data. For vulnerabilities such as use-after-free, corruption happens when a dangling pointer dereferences a freed object. After separating the freed object using one AA, the heap memory is recycled only for objects that are allowed by the separation policy, which is restrictive enough to fail newest exploitation attempts. Furthermore, SeaK creates a task that periodically scans the entire kernel memory to check for the existence of dangling pointers and starts recycling only when there is no dangling pointer. Presented with multiple N-day vulnerabilities, we employ a set of AAs to protect the kernel until patches for these vulnerabilities are released.

7 Implementation

The implementation of SeaK includes 416 lines of C code for eBPF templates, 152 lines of C code for the new helper functions in the kernel, and 649 lines of Python code for eBPF program synthesis and framework integration. In addition, SeaK has 3998 lines of C++ code based on LLVM infrastructure, to pinpoint allocation and free sites for objects of interest and support the scenario of separating vulnerability corruption by identifying vulnerable objects. SeaK is implemented over Linux and can be migrated to other open-source kernels thanks to the consistent design of the eBPF ecosystem across platforms. SeaK is open-source in GPLv2 Licence[1].

eBPF Program Synthesis. Given the type of objects for separation, SeaK analyzes the kernel image with debug information to pinpoint allocation sites and free sites in order to synthesize eBPF programs. We use LLVM infrastructure to search for these sites of interest at the source code level and use Binary Ninja to map the results to binary addresses. More specifically, we provide a list of kernel allocation and free KPIs (*e.g.*, `kfree`, `kmem_cache_free`, `kfree_skbmem`, etc) for Binary Ninja. Binary Ninja can get the addresses of all symbols, cross-reference them to their call sites, retrieve the address of each call site, and then write them into a file. After that, each address in this file is passed to `llvm-symbolizer` to create an allocation site mapping between the kernel image binary and the source code. The mapping itself is stored as a Python dictionary within a `.pickle` file. We can easily look up the mapping for the exact `call` instructions that allocate and free objects of interest. All the steps and sub-steps mentioned above are wrapped in a Python script to fully automate the entire analysis workflow.

New Helper Functions & BPF Maps. To support separating objects of interest, we extend the eBPF mechanism by adding new helper functions, including `bpf_create_slab_cache` and `bpf_create_buddy` to create dedicated regions, `bpf_get_zone` to obtain the requested zone (*e.g.*, normal zone or DMA zone), `bpf_cache_alloc` and `bpf_buddy_alloc` to allocate memory in the dedicated regions, and `bpf_set_pt_present` to set guard pages. The two BPF maps, region-index and obj2region, used in the eBPF program are both of type `BPF_MAP_TYPE_HASH`, which supports quick lookup and update. The maximum number of entries for both maps is set to 2^{14}, allowing the management of up to 16,384 dedicated regions. Our implementation is based on v5.15 the latest Long-term support (LTS) kernel version when we did our experiments. It can be easily migrated to other versions with minor modifications. We leverage the LLVM toolchain to compile eBPF programs.

Support for Loadable Kernel Modules (LKMs). In some corner cases, objects of interest are allocated and freed by loadable kernel modules. Without loading these modules, the attached address cannot be determined. To deal with this issue, we first install an eBPF program that attaches to the `load_module()` kernel function to monitor which module is loaded. Once the module is loaded, it sends out signals, allowing separation-purposed eBPF programs to be installed. Note that SeaK does not need the absolute address of the kernel module as the attached sites are based on symbols.

Support for Application Scenarios. To identify vulnerable objects that introduce corruption, we employ the approach in prior work [43] to analyze reports generated by sanitizers such as KASAN, KMSAN, KCSAN, and more. We pay special attention to the soundness of our analysis by including situations not previously considered. More specifically, some kernel objects are ordinary arrays and do not belong to any structure or union type (*e.g.*, `char* p = kmalloc(0x10)`). If we naively treat `char*` as the vulnerable type, SeaK will inevitably isolate a number of irrelevant arrays in the kernel, resulting in unnecessary overhead. Through investigation, we observe that these arrays are either referenced by a pointer field in a structure (*e.g.*, `bitmap_ip.members`) so that they can be used across system calls, or used as a temporary buffer that will be passed as a function argument, which can be tracked by our analysis. Therefore, for each array in the analysis, we create an anonymous type `type_name+offset` for differentiation. Here, `type_name` indicates the associated structure type (*e.g.*, `struct bitmap_ip`) and `offset` records the offset of the pointer field (*e.g.*, `members`). we patch the LLVM compiler to dump bitcodes before any optimization passes, thus preventing compiler optimization from influencing the accuracy of our analysis.

8 Evaluation

In this section, we use real-world cases to evaluate SeaK in terms of security improvement, performance and memory overhead, as well as scalability and stability.

[1] https://github.com/a8stract-lab/SeaK

Exploits	Sensitive Object Type	C1	C2	C3	SeaK
2021-4154 (exp1) [41]	msgseg, pipe_buffer	○	○/◑	●	●
2021-22600 (exp3) [16]	msg_msg, pipe_buffer	○	○/◑	●	●
2022-0185 (exp4) [24]	msg_msg, pipe_buffer	○	○/◑	●	●
2022-27666 (exp6) [73]	xattr, xfrm_policy	○	○/◑	●	●
2022-29582 (exp9) [56]	msgseg, tls_context	○	○/◑	●	●
2022-1786 (exp13) [69]	timerfd_ctx	○	○/◑	●	●
2022-20409 (exp15) [42]	cred	○	○/◑	○	●

Table 3: Results of SeaK's security improvement for separating security-sensitive objects. The "Exploits" column includes CVE IDs and the internal exploit ID from Google. The "Sensitive Object Type" means the type of objects misused in the exploit. ○ indicates failing to prevent exploitation, ◑ stands for working occasionally due to the sampling nature, ● means succeeding in preventing exploitation.

8.1 Security Analysis

To evaluate the security improvement of SeaK, we draw a comparison between it and existing security features, no matter whether they are enabled by default or not.

Dataset & Criteria. We built two datasets corresponding to the two illustrative scenarios. The first dataset is for the scenario of separating security-sensitive objects. It includes seven exploits collected by Google and Alphabet Vulnerability Reward Program [36]. We didn't include all cases because the remaining lacks publicly available, functional exploit code. We selected this program as the data source because the program report clearly specifies which security-sensitive object is used in each case, which saves our time and avoids inaccuracies in identifying which type of object for separation.

The second dataset is for the scenario of separating vulnerability corruption. This dataset is constructed using vulnerabilities reported by Syzkaller [15]. Every case from Syzkaller encompasses a report, a configuration file, and a PoC program, all aiding in the reproduction of the vulnerability. We randomly selected 50 vulnerabilities reported after kernel version v4.15 [2], successfully reproducing 46 of them. These include 30 reported by KASAN, 1 by KMSAN, 1 by UBSAN, 5 by BUG_ON macro, 2 by GPF, and 7 by WARNING macro. The diversity within these vulnerabilities ensures that the dataset is representative.

Separating Security-Sensitive Objects. Table 3 shows the results of separating security-sensitive objects against the first dataset. In general, SeaK outperforms C1 (*i.e.*, freelist randomization + freelist obfuscation + heap zeroing), C2 (*i.e.*, structure layout randomization + KFENCE), and also C3 (*i.e.*, slub_debug=UFPZ).

For C1, all exploits can bypass by-default enabled features in it. First, freelist randomization essentially cannot prevent temporal corruption based exploitation (*e.g.*, exp1, exp3, exp9, exp13, exp15) and is bypassed using heap grooming [39] in the exploitation of spatial corruption based exploitation (*e.g.*, exp4, exp6). Second, freelist obfuscation and heap zeroing are not activated because no exploit tampers with freelist pointer and relies on uninitialized value for KASLR bypassing. Instead, they use the read capability introduced by the vulnerability to leak kernel base address.

For C2, structure layout randomization fails to prevent all exploits because it only randomizes structures with every field as a function pointer. None of the sensitive objects misused in the exploit fall into this category. KFENCE samples objects for separation. During the experiment, we maximized the capability of KFENCE by minimizing its sampling interval to 1 ms and expanding its memory pool to its limit of 512 MB. We discovered that it can protect only 0.005% - 0.35% sensitive objects in Table 3.

For C3, slub_debug successfully thwarts most exploits except for exp15. This exploit performs DirtyCred attack [44] which overlaps two cred carrying different levels of privilege, thereby evading detection through red zone, poisoning, and user tracking.

In comparison, SeaK prevents all exploits, showcasing the strongest security improvement. On one hand, guard pages and offset randomization hinder exploitation utilizing spatial overlapping, achieving the same effect as slub_debug. On the other hand, SeaK only recycles objects allocated from the same site, having the same size, carrying the same privilege level, and in the same zone. Therefore, it can handle not only common temporal overlapping attacks but also the newest same-type exploitation like DirtyCred. Moreover, when new attacks emerge in the future, SeaK supports updating the recycling policy correspondingly to keep pace with the evolving threat landscape.

Separating Vulnerability Corruption. Table 4 shows the sampled results for separating vulnerability corruption against the second dataset. The results share a similarity with the first scenario: C1 cannot restrict corruption, so is structure layout randomization in C2; KFENCE in C2 works occasionally if the vulnerable object is sampled. slub_debug can separate all corruptions when proactive attacks are not present.

FP and FN. To separate vulnerability corruption, SeaK identifies the vulnerable object and its allocation and free sites. During this process, SeaK can have False Positive (FP) - identifying objects that are not vulnerable or sites that allocating irrelevant objects, and False Negative (FN) - missing vulnerable objects or sites that allocate vulnerable objects. SeaK can accommodate FP by separating more objects at the cost of a minor increase in overhead, thanks to its scalability (See Section 8.2). The elimination of FN relies on the soundness of the analysis. To achieve this, we used the state-of-the-art techniques in implementation (See Section 7). However, it is important to note that, to date, no whole-program analysis is sound when applied to real programming languages [46]. Therefore, thouogh empirically SeaK didn't overlook any vul-

[2] LLVM is unable to compile kernel earlier than this version, and many eBPF features used in SeaK did not exist at the time.

SYZ Title	C1	C2	C3	Type of Identified Vulnerable Object	SeaK
GPF-delayed_uprobe_remove	○	○/◑	●	**delayed_uprobe**	●
WARNING-call_rcu	○	○/◑	●	**route4_filter**	●
WARNING-ODEBUG bug-tcf_queue_work	○	○/◑	●	**route4_filter**, workqueue_struct	●
KASAN-uaf-read-route4_get	○	○/◑	●	Qdisc, route4_bucket, **route4_filter**, route4_head...	●
UBSAN-shift-oob-dummy_hub_control	○	○/◑	●	urb, usb_ctrlrequest, usb_device, **usb_hcd**	●
KASAN-uaf-read-hci_send_acl	○	○/◑	●	**hci_chan**, hci_conn, hci_dev, l2cap_conn, work_struct	●
BUG-corrupted list-kobject_add_internal	○	○/◑	●	**hci_conn**	●
KMSAN-uninit-value-geneve_xmit	○	○/◑	●	**netdev**, sk_buff	●
KASAN-slab-oob-write-decode_data	○	○/◑	●	tty_ldisc, **tty_struct**	●

Table 4: Sampled results of SeaK's security improvement for separating vulnerability corruption. More results can be found in Table 9. The "SYZ Title" column is the bug report title minus an uninformative preposition. Structures in bold in "Type of Identified Vulnerable Object" column are ground truth. The remainings are FPs. SeaK has no FN in all test cases. ○ indicates failing to prevent corruption from damaging other kernel objects, ◑ stands for working occasionally due to the sampling nature, ● means succeeding in separating corruption.

nerable object in all test cases (See Table 4 and 9), we must caution users about the risk of FNs - SeaK might occasionally fail to separate vulnerable objects.

Comparison with Other Works. Beyond C1, C2, C3, in Section 6.1, we mentioned several works also for object separation purpose. Among them, xMP [53] and kalloc_type [30] are comparable to SeaK security-wise. However, AutoSlab inadvertently simplifies cross-cache exploitation [40] because it separates kernel objects per slab cache which eases recycling at the buddy system level. Google slab_virtual's newest implementation [63] prevents temporal-corruption-based exploitation but partially works for spatial-corrution-based exploitation. Besides, it cannot handle DMA objects which must reside in DMA zone. On a different note, PET [66] prevents vulnerability triggering, focusing on a different stage of the exploitation chain from SeaK. It can miss vulnerabilities triggered through different paths or at different sites [43].

8.2 Overhead Measurement

Benchmarks and Settings. We employed identical benchmarks and settings including variances for measuring the overhead of SeaK as those utilized for the existing security features outlined in Section 3.2. In addition to the vanilla kernel image, we built a hardened kernel image incorporating the SeaK extensions. Both images have the BPF JIT engine enabled. Note that, we needn't build a different kernel for each individualAA. Once the SeaK extension is there, any number of AAs can be installed.

We used bpftrace to profile the lifespan of kernel objects for 20 minutes in two situations - no workload and running LMbench. The profiling revealed that most kernel objects are allocated and freed quickly while a small fraction of objects have long lifespan. No objects have both a long lifespan and frequent operations, as this combination would result in a substantial amount of objects lingering in memory, thus easily exhausting kernel memory.

Based on the profiling results, we consider three representative situations - Cold, Hot, and Durable. "Cold" refers to an object whose allocation and free rarely happen. We use KASAN-uaf-l2cap_chan_close [60] from the dataset for separating vulnerability corruption as an example of this situation, because its vulnerable object - `struct l2cap_chan` is allocated only in function `l2cap_chan_create` in the Bluetooth module. "Hot" indicates that the object is frequently allocated and freed, but has a short lifespan. Our profiling shows that `struct seq_operations` is the hottest in both no workload situation and running LMbench situation - allocated 32.7 times and 47.34 times per second respectively. "Durable" stands for an object that has a long lifespan and a moderate operation frequency. Our profiling suggests `struct cred`, `struct sk_filter`, `struct fdtable`, because they have 5s, 20s, and 30s lifespan. Further, we include `struct file` into our measurement because "Everything is a file".

Performance Overhead. Table 8.2 presents the performance overhead of the three representative situations. Intuitively, one would anticipate the overhead of "Hot" is the highest, given the correlation between performance and the frequency of allocation and free. However, the data illustrates that "Hot" doesn't show any significant increase compared to the other situations. Specifically, the overhead ranges from -2.42% to 2.70% for LMbench, and -1.73% to 1.64% for Phoronix, which is within reasonable margin of fluctuation and aligns with the data for "Cold" and "Durable". Therefore, we conclude that SeaK has negligible overhead, regardless of the benchmark and the use frequency of objects that are separated.

To further validate our conclusion and have a deeper understanding of SeaK's performance, we measured the latency caused by critical operations in SeaK. The potential overhead, if there is any, will come from eBPF program execution and

LMbench (ms)	Vanilla	Cold	Hot	Durable			File
Simple syscall	0.1942	-1.68%	-0.67%	0.06%	0.08%	-0.29%	-0.94%
Simple read	0.2946	0.20%	-0.58%	0.49%	-0.48%	0.03%	-0.45%
Simple write	0.2502	-2.67%	-2.42%	0.51%	0.15%	0.56%	-0.18%
Select on 100 fd's	1.0718	0.26%	0.20%	-0.16%	-0.49%	-0.10%	-0.01%
Signal handler install	0.2538	-1.28%	-1.32%	0.17%	-0.33%	0.11%	0.02%
Signal handler overhead	0.8815	-0.90%	-1.53%	0.12%	1.54%	0.35%	-0.33%
fork+exit	99.6357	0.83%	2.49%	-0.49%	-2.82%	-3.44%	-2.43%
fork+execve	283.2725	1.51%	0.23%	2.32%	1.82%	-1.76%	3.34%
fork+/bin/sh -c	678.1250	2.93%	2.70%	2.35%	0.23%	-1.16%	2.28%
UDP latency	5.8852	1.25%	-1.10%	0.07%	-0.73%	-1.37%	-0.32%
TCP/IP connection	10.1259	0.13%	0.78%	0.51%	-0.01%	2.04%	1.62%
AF_UNIX bandwidth	9460.5067	0.67%	-0.56%	0.71%	0.92%	-1.85%	-1.26%
Pipe bandwidth	4569.4767	0.87%	-1.37%	-1.03%	1.94%	0.56%	-3.02%
Phoronix	**Vanilla**	**Cold**	**Hot**	**Durable**			**File**
Sockperf (Msgs/sec)	739608	-0.04%	-1.73%	-1.30%	0.75%	0.63%	0.93%
OSBench (Ns/Event)	78.28	-0.92%	-0.30%	-0.23%	-1.18%	-0.15%	-2.23%
7-Zip Compress (MIPS)	29521	-1.31%	-0.95%	1.07%	0.60%	1.62%	0.97%
FFmpeg Live (FPS)	178.08	0.45%	-1.29%	1.63%	1.57%	0.86%	0.68%
OpenSSL SHA256 (B/s)	1225189783	0.28%	-0.31%	-0.05%	0.02%	0.23%	-0.08%
Redis SET (Reqs/sec)	1932771	1.49%	-1.21%	-3.36%	-0.28%	0.30%	1.03%
SQLite Speedtest (sec)	62.63	0.57%	1.64%	-1.41%	-0.88%	1.44%	-0.41%
Apache 100 (Reqs/sec)	48216	-0.63%	-0.95%	-0.40%	0.49%	0.68%	0.18%

Table 5: Performance overhead of `SeaK`. Vanilla indicates the baseline. The "Cold", "Hot", and "Durable" columns are three representative situations explained in Section 8.2. Especially, three columns in "Durable" indicate `cred`, `sk_filter`, and `fdtable` from left to right.

Operations	Time (ns)	Operations	Time (ns)
create region	13,320.93	idle handler	1,479.55
allocate from region	1,545.04	alloc handler	7,108.49
free to region	121.03	free handler	1,597.02
set guard page	280.64		

Table 6: Latencies of critical operations in `SeaK`. The idle handler represents an empty eBPF program; the alloc handler refers to the time in eBPF programs minus allocation helper function; the free handler is time in eBPF programs minus free helper function.

the creation of dedicated regions. Therefore, we inserted `rdtsc` instructions before and after these operations to collect the data. The results are presented in Table 6. As we can see, all latencies are below 0.02 ms, significantly smaller than the minimum time required for a system call - 0.1942 ms for simple syscall in Table 8.2. Among all operations, the most time-consuming one is "create region", taking 0.013 ms. It occurs only when a dedicated region is created - the AA performs allocation for the first time. Such a small overhead is unlikely to impact real-world applications - as doubly confirmed in Table 8.2.

Memory Overhead. The additional memory brought in by `SeaK` comprises the guard pages and the unused memory within the page(s) that hold the object. As two separated objects can share one guard page, the maximum memory wastage is 8KB for each 16B object. However, such extreme memory inflation is very rare, as our profiling indicates that 87.2% objects in the kernel are at least 64 bytes. Figure 5 presents the increased memory usage for various situations when running LMbench. Unlike existing features (Figure 2),

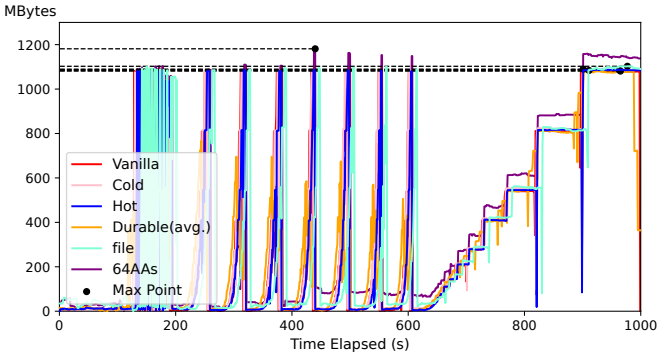

Figure 5: Increased memory usage of `SeaK` when running LMbench. The lines of all situations almost completely overlap with each other, except for 64 AAs (in purple). Durable (avg.) averages memory usage of `cred`, `sk_filter`, and `fdtable`.

the memory usage lines for "Cold", "Hot", and 'Durable (avg.)' almost completely overlap with Vanilla. On the one hand, it is because AA offers the most granular level of separation rather than indiscriminately protect every object all the time. Therefore, itself doesn't impose excessive memory overhead. On the other hand, no kernel object has a long lifespan while also being frequently allocated. Therefore, the memory consumed by `SeaK` is either quickly reclaimed, as seen in "Hot" and "Cold", or sufficiently small to remain unnoticeable, as in"Durable".

Comparison with Other Works. Besides C1, C2, C3, Section 6.1 mentioned several protections from academia and commercial products that are also for object separation. While the performance of kalloc_type [30] and AutoSlab [40] can not be evaluated due to their closed-source nature, a comparison can be made among `SeaK`, xMP [53], slab_virtual [63], and PET [66], based on publicly available results and our measurement in the same setting: xMP's average overhead for isolating `cred` using LMbench is 22.34% (calculated from paper's data), exceeding Seak's 1%. The Phoronix results of both are negligible. The average performance overhead of slab_virtual is 1.09% which is larger than SeaK-64 case's 0.4%. PET's performance overhead is 3% and memory overhead is 5.6% (915MB/16GB), as shown in the paper, while Seak's overhead is minimal in both.

8.3 Scalability & Stability Analysis

To deploy `SeaK` in the real world, it is essential to evaluate its scalability when multiple AAs are used. To measure this, we gathered all AAs from our security analysis and randomly selected additional security-sensitive objects identified in [65], to obtain in total of 64 unique AAs. We increased the number of installed AAs exponentially, without any specific order, and presented the averaged performance overhead in Table 7 (More complete results are in Table 8 in Appendix). The performance shows no noticeable degradation as the number

LMbench	2 AAs	4 AAs	8 AAs	16 AAs	32 AAs	64 AAs
Avg.	-0.32%	0.0%	-0.55%	0.01%	0.20%	0.04%
Phoronix	**2 AAs**	**4 AAs**	**8 AAs**	**16 AAs**	**32 AAs**	**64 AAs**
Avg.	-0.28%	-0.74%	-0.33%	-0.31%	0.71%	0.22%

Table 7: Performance of SeaK scaling up to 64 AAs. More detailed results are in Table 8.

of AAs grows, regardless of the benchmark used. This is because one single AA, no matter for cold or hot objects, has negligible overhead. Regarding memory overhead, Figure 5 shows that the memory usage of 64 AAs is on par with Vanilla, with occasional variances up to 100MB. Such intermittent overhead is negligible for modern OS which has access to the entire physical memory - often multiples of 4GB - owing to SDRAM technology. To conclude, SeaK is scalable when multiple AAs are present.

Regarding stability, we enabled all 64 AAs on the machine used for daily research and education activities, including Overleaf, Outlook, Zoom meetings, ChatGPT, Docker containers for CTF challenges, and plugining/unplugining peripherals. Over nearly 2.5 months, the machine maintained consistent stability without encountering any issues.

9 Discussion & Future Directions

eBPF Alternatives and Security Issues. SeaK employs the eBPF ecosystem because it is safe, expressive, efficient, and only privileged users can install eBPF programs. Without considering these advantages, there are alternatives of eBPF ecosystem like Kprobe. Kprobe stores raw data rather than structures data in kernel memory and relies on a userspace process to be in charge of data sharing between modules. Starting from Kprobe, we need to reinvent the wheels already provided by the eBPF ecosystem. This requires significant engineering efforts and inevitably incurs higher performance overhead due to additional kernel-user switches for data sharing.

A recent work EPF [34] reveals that attackers can misuse BPF code as gadgets for code reuse attacks. Furthermore, the eBPF ecosystem was previously reported to contain vulnerabilities [12–14]. Though addressing these security issues are orthogonal to SeaK, it remains worthwhile to pay attention to the downsides of eBPF in the deployment of SeaK.

Hardware Support. SeaK relies on randomization and guard pages to separate objects within dedicated regions. While the entropy stands strong at 43, it can be further strengthened through hardware features or hypervisor features if present. Some promising memory protection features include Intel MPK, Arm protection domain, and RISC-V Donky [58], and Xen. However, considering that many devices, especially embedded ones, lack these features, we choose not to integrate them into SeaK at this time to maintain its broad applicability. In the future, we will release new versions to leverage these features specific to different platforms.

Extending to stack. Though SeaK is designed as a kernel secure allocator, it can be extended to prevent stack attack. For example, we can attach eBPF programs to functions where stack overflow might happen, recording the value of the return address in BPF maps at the function entry and examining its integrity at the function exit. We will integrate this extension into SeaK in the future.

10 Related Works

Heap security features from the upstream kernel have been evaluated in Section 3 and those from academia and commercial products have been compared with SeaK in Section 8.

The remaining related works are secure allocator in user space, which employs safe design principles and randomized mechanisms to mitigates heap vulnerabilities. Typically, these allocators separate in-place metadata from the object and randomize both the allocation and reuse of memory. Notable examples include DieHard [19] and its successor DieHarder [48], which rely on abundant memory space to allocate more memory than required and randomize object locations. FreeGuard [59] enhances performance by integrating the security features of BIBOP-style allocators with the rapid allocation capabilities of freelist-style allocators. Guarder [57] refines FreeGuard by fine-tuning the level of entropy, while SlimGuard [45] improves memory efficiency through fine-grained memory class management. While SeaK benefits from its user-space counterparts, certain features are not applicable to kernel space. For example, the kernel doesn't have infinite memory which is commonly assumed in these works.

In addition to general-purposed secure allocators, there are specialized allocators focusing solely on mitigating use-after-free vulnerabilities. Cling [50] ensures that freed objects are reused only if their types match, identified through runtime call stack analysis. MarkUs [17] employs garbage collection principles to free objects when no dangling pointers reference them. Oscar [62] and FFmalloc [20] operate under the assumption of unlimited memory space, allocating objects once and never reusing them. Vik [23] assigns IDs to allocated objects and permits only pointers with matching IDs to access the object, optimizing overhead through ARM hardware features. These specialized allocators are generally unsuitable for kernel space because their assumptions do not apply there. Besides, they are overly specific to certain vulnerability type while SeaK generally works.

11 Conclusion

This work presents SeaK, the first practical secure allocator in the kernel that substantially strengthens heap security without

introducing noticeable overhead. It is anchored in a new strategy of designing a secure kernel allocator, centering around the "atomic alleviation" concept. This new strategy is derived from in-depth analyses of existing kernel heap security features and is further validated by comprehensive evaluation of SeaK in terms of security improvement, performance and memory overhead, scalability, and stability.

References

[1] Alexander Popov's blog. https://a13xp0p0v.github.io/.

[2] Analysis and Exploitation of a Linux Kernel Vulnerability. - Perception Point. https://perception-point.io/analysis-and-exploitation-of-a-linux-kernel-vulnerability-2/.

[3] bevx-talk. https://duasynt.com/slides/bevx-talk.pdf.

[4] CVE-2016-6187: Exploiting Linux kernel heap off-by-one - Vitaly Nikolenko. https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit.

[5] eBPF for Windows: Main Page — microsoft.github.io. https://microsoft.github.io/ebpf-for-windows/. [Accessed 06-Feb-2023].

[6] eBPF Implementation for FreeBSD :: FreeBSD Presentations and Papers — papers.freebsd.org. https://papers.freebsd.org/2018/bsdcan/hayakawa-ebpf_implementation_for_freebsd/. [Accessed 06-Feb-2023].

[7] Lexfo's security blog - CVE-2017-11176: A step-by-step Linux Kernel exploitation (part 1/4). https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html.

[8] Upstream bug lifetimes. https://syzkaller.appspot.com/upstream/graph/lifetimes.

[9] Vitaly Nikolenko's blog. https://duasynt.com/.

[10] ZDI-17-240 | Zero Day Initiative. https://www.zerodayinitiative.com/advisories/ZDI-17-240/.

[11] Phoronix Test Suite, 2015. http://www.phoronix-test-suite.com/.

[12] CVE - CVE-2021-4204 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4204, 2022.

[13] CVE - CVE-2022-0264 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0264, 2022.

[14] CVE - CVE-2022-23222 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23222, 2022.

[15] Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2023. https://github.com/google/syzkaller.

[16] Rajvardhan Agarwal. CVE-2021-22600, 2022. https://github.com/r4j0x00/exploits/tree/master/CVE-2021-22600.

[17] Ainsworth, Sam and Jones, Timothy M. MarkUs: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[18] Anonymous Author(s). Raw results of evaluation. https://tinyurl.com/3ytyevpb, October 2022.

[19] Berger, Emery D. and Zorn, Benjamin G. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[20] Brian Wickman and Hong Hu and Insu Yun and DaeHee Jang and JungWon Lim and Sanidhya Kashyap and Taesoo Kim. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *30th USENIX Security Symposium (USENIX Security)*, 2021.

[21] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A Systematic Study of Elastic Objects in Kernel Exploitation. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[22] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[23] Cho, Haehyun and Park, Jinbum and Oest, Adam and Bao, Tiffany and Wang, Ruoyu and Shoshitaishvili, Yan and Doupé, Adam and Ahn, Gail-Joon. ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[24] clubby789. CVE-2022-0185: A Case Study - A tale on discovering a Linux kernel privesc, 2022. https://www.hackthebox.com/blog/CVE-2022-0185:_A_case_study.

[25] Kees Cook. security things in linux v4.13, 2017. https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/.

[26] Kees Cook. security things in linux v4.14, 2017. https://outflux.net/blog/archives/2017/11/14/security-things-in-linux-v4-14/.

[27] Kees Cook. [v4] mm: Add SLUB free list pointer obfuscation. https://patchwork.kernel.org/project/linux-hardening/patch/20170726041250.GA76741@beast/, 2022.

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

[29] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[30] Pierre H. https://twitter.com/pedantcoder/status/1470585072361172993?lang=en, 2021.

[31] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *Proceedings of the 31th USENIX Security Symposium (USENIX Security)*, 2022.

[32] Hsin-Wei Hung, Yingtong Liu, and Ardalan Amiri Sani. Sifter: protecting security-critical kernel modules in android through attack surface reduction. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022.

[33] Nur Hussein. Randomizing structure layout, 2017. https://lwn.net/Articles/722293/.

[34] Di in, Vaggelis Atlidakis, and Vasileios P Kemerlis. EPF: Evil packet filter. In *USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.

[35] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[36] kCTF VRP. Kernel Exploits Recipes Notebook. https://docs.google.com/document/d/1a9uUAISBzw3ur1aLQqKc5JOQLaJYiOP5pe_B4xCT1KA/, October 2022.

[37] Imran Khan. Linux SLUB Allocator Internals and Debugging - SLUB Debugger, Part 2 of 4. https://blogs.oracle.com/linux/post/linux-slub-allocator-internals-and-debugging-2, 2022.

[38] Andrey Konovalov. Linux Kernel Exploitation, 2020. https://github.com/xairy/linux-kernel-exploitation.

[39] Azeria Labs. Grooming the ios kernel heap, 2020. https://azeria-labs.com/grooming-the-ios-kernel-heap/.

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

[41] Zhenpeng Lin. CVE-2021-4154, 2022. https://github.com/Markakd/CVE-2021-4154/blob/master/WRITEUP.md.

[42] Zhenpeng Lin. CVE-2022-20409, 2022. https://github.com/Markakd/bad_io_uring.

[43] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[44] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[45] Liu, Beichen and Olivier, Pierre and Ravindran, Binoy. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *Proceedings of the 20th International Middleware Conference(Middleware)*, 2019.

[46] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. In *Communications of the ACM (CACM)*, 2015.

[47] Larry McVoy and Carl Staelin. LMbench - Toos for Performance Analysis, 2015. http://lmbench.sourceforge.net/.

[48] Novark, Gene and Berger, Emery D. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[49] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[50] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *19th USENIX Security Symposium (USENIX Security)*, 2010.

[51] Jesse Polhemus. Vasileios kemerlis wins an nsf career award for adaptive hardening, debloating, and hardware-assisted protection. https://cs.brown.edu/news/2023/03/21/vasileios-kemerlis-wins-nsf-career-award-adaptive-hardening-debloating-and-hardware-assisted-protection/.

[52] Alexander Potapenko. mm: security: introduce init_on_alloc=1 and init_on_free=1 boot options. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6471384af2a6530696fc0203bafe4de41a23c9ef, 2022.

[53] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[54] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, 2022.

[55] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[56] Ruia-ruia. CVE-2022-29582, 2022. https://github.com/Ruia-ruia/CVE-2022-29582-Exploit.

[57] Sam Silvestro and Hongyu Liu and Tianyi Liu and Zhiqiang Lin and Tongping Liu. Guarder: A Tunable Secure Allocator. In *27th USENIX Security Symposium (USENIX Security)*, 2018.

[58] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security)*, 2020.

[59] Silvestro, Sam and Liu, Hongyu and Crosser, Corey and Lin, Zhiqiang and Liu, Tongping. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[60] syzbot. WARNING:refcount bug in l2cap_chan_put, 2020. https://syzkaller.appspot.com/bug?id=39d35c93d0856ca3134bf97f8bb3f249808c2751.

[61] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo Workarounds for Kernel Bugs. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[62] Thurston H.Y. Dang and Petros Maniatis and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *26th USENIX Security Symposium (USENIX Security)*, 2017.

[63] Torvalds. Slub virtual. https://github.com/thejh/linux/tree/slub-virtual/.

[64] Eduardo Vela. Making Linux Kernel Exploit Cooking Harder. https://security.googleblog.com/2022/08/making-linux-kernel-exploit-cooking.html, August 2022.

[65] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. AlphaEXP: An expert system for identifying Security-Sensitive kernel objects. In *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[66] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. PET: Prevent discovered errors from being triggered in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[67] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Wei Zou, and Xiaorui Gong. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[68] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[69] Kyle Zeng. CVE-2022-1786, 2022. https://blog.kylebot.net/2022/10/16/CVE-2022-1786.

[70] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K (H) eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, 2022.

[71] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, and Ryan Stutsman. XRP:In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[72] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[73] Xiaochen Zou. CVE-2022-27666, 2022. https://github.com/plummm/CVE-2022-27666.

LMBench	**2 cases**	**4 cases**	**8 cases**	**16 cases**	**32 cases**	**64 cases**
Simple syscall	0.70%	-0.01%	-1.52%	-1.20%	0.28%	1.43%
Simple read	0.06%	0.16%	-0.35%	0.16%	0.78%	0.05%
Simple write	0.55%	-2.28%	-2.58%	-2.28%	-0.21%	2.44%
Select on 100 fd's	-0.11%	-0.04%	0.11%	0.00%	-0.36%	0.01%
Signal handler install	-0.77%	-1.21%	-1.55%	-1.21%	-1.01%	-0.39%
Signal handler overhead	0.26%	-0.34%	-1.14%	-0.58%	1.55%	3.29%
fork+exit	-2.68%	3.26%	0.06%	3.26%	-2.04%	-3.30%
fork+execve	-1.95%	-0.90%	-0.50%	-0.90%	3.01%	-1.99%
fork+/bin/sh -c	-0.39%	0.06%	0.65%	0.06%	1.07%	-2.96%
UDP latency	0.95%	0.17%	-0.34%	0.17%	-0.23%	0.92%
TCP/IP connection	1.72%	0.64%	-0.15%	0.64%	1.20%	0.10%
AF_UNIX bandwidth	-1.09%	0.13%	0.26%	0.13%	-1.54%	-0.16%
Pipe bandwidth	-1.45%	1.00%	-0.16%	1.89%	0.13%	0.04%
Avg.	-0.32%	0.05%	-0.55%	0.01%	0.20%	0.04%
Phoronix	**2 cases**	**4 cases**	**8 cases**	**16 cases**	**32 cases**	**64 cases**
Sockperf (Msgs/sec)	0.48%	-1.33%	-1.65%	-1.30%	4.20%	3.75%
OSBench (Ns/Event)	-0.24%	-0.16%	-0.19%	-0.23%	1.45%	0.45%
7-Zip Compress (MIPS)	-1.88%	-1.22%	-0.50%	1.07%	-0.29%	0.41%
FFmpeg Live (FPS)	0.48%	-0.83%	-0.34%	1.63%	1.97%	0.87%
OpenSSL SHA256 (B/s)	-0.10%	-0.16%	-0.09%	-0.05%	-0.07%	0.04%
Redis SET (Reqs/sec)	0.94%	-3.30%	-3.06%	-3.36%	-1.06%	-2.99%
SQLite Speedtest (sec)	0.37%	-0.31%	0.57%	1.41%	0.00%	0.15%
Apache 100 (Reqs/sec)	-0.30%	-0.52%	-0.71%	-0.40%	-0.55%	-0.85%
Avg.	-0.28%	-0.74%	-0.33%	-0.31%	0.71%	0.22%

Table 8: Complete results for the performance of SeaK scaling up to 64 AAs. Corresponding sampled results are in Table 7.

A Additional Results

Due to the space limit, we only present sampled results in the text. In Google Sheet [18], we present more results regarding memory usage of existing features and SeaK when running Phoronix. In addition, Table 8 presents more complete results regarding SeaK's scalability in performance. Table 9 shows the FN and FP results of static analysis for separating vulnerability corruption. List 2 shows an example of synthesized eBPF program.

SYZ Title	C1	C2	C3	Type of Identified Vulnerable Object	SeaK
BUG__corrupted_list_in_kobject_add_internal	○	○/◑	●	**hci_conn**	●
KASAN__use-after-free_Read_in_sctp_auth_free	○	○/◑	●	crypto_shash, **sctp_endpoint**	●
KASAN__use-after-free_Write_in__sco_sock_close	○	○/◑	●	**hci_conn**	●
KMSAN__uninit-value_in_geneve_xmit	○	○/◑	●	sk_buff, **net_device**	●
WARNING__refcount_bug_in_l2cap_chan_put	○	○/◑	●	**l2cap_chan**	●
KASAN__slab-out-of-bounds_Read_in_tcf_exts_destroy	○	○/◑	●	tc_action, **tcindex_filter_result**	●
KASAN__use-after-free_Read_in_rdma_listen	○	○/◑	●	**rdma_id_private**	●
BUG__corrupted_list_in_nft_obj_del	○	○/◑	●	**nft_object**	●
BUG__corrupted_list_in_nf_tables_commit	○	○/◑	●	**nft_flowtable**,**nft_trans**	●
general_protection_fault_in_delayed_uprobe_remove	○	○/◑	●	**delayed_uprobe**	●
KASAN__use-after-free_Read_in_delayed_uprobe_remove	○	○/◑	●	**delayed_uprobe**	●
KASAN__use-after-free_Read_in_x25_device_event	○	○/◑	●	net_device, **x25_neigh**	●
WARNING__ODEBUG_bug_in_tcf_queue_work	○	○/◑	●	workqueue_struct,**route4_filter**	●
KASAN__use-after-free_Read_in_route4_get	○	○/◑	●	Qdisc, route4_bucket,route4_head, sk_buff, tcf_block, tcf_chain, tcf_proto,**route4_filter**	●
WARNING__ODEBUG_bug_in_route4_change	○	○/◑	●	**route4_filter**	●
WARNING_in_call_rcu	○	○/◑	●	**route4_filter**	●
KASAN__slab-out-of-bounds_Write_in_default_read_copy_kernel	○	○/◑	●	snd_pcm_oss_file, snd_pcm_plugin, snd_pcm_runtime, snd_pcm_substream,**snd_pcm_plugin_channel**	●
KASAN__slab-out-of-bounds_Read_in_default_write_copy_kernel	○	○/◑	●	snd_pcm_oss_file, snd_pcm_plugin, snd_pcm_runtime, snd_pcm_substream,**snd_pcm_plugin_channel**	●
general_protection_fault_in_vb2_mmap	○	○/◑	●	video_device,**vb2_buffer**	●
KASAN__use-after-free_Read_in_vb2_mmap	○	○/◑	●	video_device,**vb2_buffer**	●
KASAN__use-after-free_Read_in__list_add_valid	○	○/◑	●	sockaddr, ucma_context, ucma_file,**rdma_id_private**	●
KASAN__null-ptr-deref_Read_in_refcount_sub_and_test_checked	○	○/◑	●	**vb2_vmalloc_buf**	●
KASAN__use-after-free_Write_in__ext4_expand_extra_isize	○	○/◑	●	ext4_inode_info, ext4_sb_info, **inode**	●
KASAN__use-after-free_Read_in__nf_tables_abort	○	○/◑	●	list_head, net, nft_trans, sk_buff, sock, **nft_table**	●
BUG__corrupted_list_in__nf_tables_abort	○	○/◑	●	list_head, nft_flowtable, nft_rule, nft_set, **nft_table**	●
WARNING_in_snd_info_get_line	○	○/◑	●	pde_opener, proc_dir_entry, seq_file, snd_info_private_data, **snd_info_buffer**	●
KASAN__slab-out-of-bounds_Write_in_decode_data	○	○/◑	●	tty_ldisc, **tty_struct**	●
KASAN__use-after-free_Read_in_ip6_hold_safe	○	○/◑	●	**dst_entry**	●
KASAN__use-after-free_Write_in_ip6_hold_safe	○	○/◑	●	**dst_entry**	●
KASAN__use-after-free_Read_in_l2cap_chan_close	○	○/◑	●	**l2cap_chan**	●
KASAN__slab-out-of-bounds_Read_in_cap_inode_getsecurity	○	○/◑	●	**vfs_cap_data**	●
kernel_BUG_at_fs/userfaultfd.c	○	○/◑	●	**userfaultfd_ctx**	●
KASAN__use-after-free_Read_in_handle_userfault	○	○/◑	●	mm_struct, vm_area_struct,**userfaultfd_ctx**	●
KASAN__use-after-free_Read_in__rhashtable_lookup	○	○/◑	●	**rhashtable**	●
KASAN__use-after-free_Read_in_hci_send_acl	○	○/◑	●	hci_conn, hci_dev, l2cap_conn, work_struct, **hci_chan**	●
WARNING_in_snd_usbmidi_submit_urb/usb_submit_urb	○	○/◑	●	**urb**	●
UBSAN__shift-out-of-bounds_in_dummy_hub_control	○	○/◑	●	urb, usb_ctrlrequest, usb_device, **usb_hcd**	●
KASAN__use-after-free_Read_in_tcp_check_sack_reordering	○	○/◑	●	**sk_buff**	●
KASAN__use-after-free_Read_in__vb2_perform_fileio	○	○/◑	●	vb2_fileio_data, **video_device**	●
KASAN__slab-out-of-bounds_Read_in_bitmap_ip_add	○	○/◑	●	**bitmap_ip->members**	●
KASAN__slab-out-of-bounds_Read_in_bitmap_ip_ext_cleanup	○	○/◑	●	net, nlattr, sk_buff, sock, **bitmap_ip->members**	●
KASAN__slab-out-of-bounds_Write_in_bitmap_ip_del	○	○/◑	●	**bitmap_ip->members**	●
KASAN__use-after-free_Read_in__queue_work	○	○/◑	●	atomic64_t, **work_struct**	●
KASAN__use-after-free_Read_in_ep_scan_ready_list	○	○/◑	●	atomic64_t, **work_struct**	●
KASAN__use-after-free_Read_in_p9_fd_poll	○	○/◑	●	list_head, p9_client, **p9_trans_fd**	●
WARNING__ODEBUG_bug_in_p9_fd_close	○	○/◑	●	p9_client, **p9_trans_fd**	●

Table 9: More results of SeaK's security improvement for separating vulnerability corruption. Structures in bold in "Type of Identified Vulnerable Object" are ground truth. The remaining is FP. SeaK has no FN in all test cases. ○ indicates failing to prevent exploitation, ◑ stands for working occasionally due to the sampling nature, ● means succeeding in preventing exploitation.

```
// essential utilities
// define a map where an alloc_addr corresponds to a key
struct {
    ...
} addr2key SEC(".maps");
// define a map where a key corresponds to a cache
struct {
    ...
} key2cache SEC(".maps");
int alloc_handler(struct pt_regs* ctx,
    u64 kpi_type) {
        // one key is related to one kind of object
        key = generate_key(ctx);
        //for kmem_cache_alloc,
        //we should get the kmem_cache first.
        //Then we can read the alloc_size through kmem_cache.
        if(kpi_type == kmem_cache_alloc){
            cache = (struct kmem_cache*) PT_REGS_PARM1(ctx);
            alloc_size = BPF_CORE_READ(cache, size);
            alloc_size = get_size(alloc_size);
        }else if(kpi_type == kmalloc){
            alloc_size = PT_REGS_PARM1(ctx);
            alloc_size = get_size(alloc_size);
        }
        //use key to find the related cache in key2cache
        cache = bpf_look_up_cache(key2cache, key)
        if(!cache){
            //create a cache if first allocted
            cache = generate_cache(key,alloc_size);
        }
        //allocate an object
        alloc_addr = bpf_cache_alloc(cache)
        update_map(addr2key,alloc_addr,key);
    }
int free_handler(struct pt_regs* ctx,
    u64 kpi_type) {
    //for kmem_cache_free
    //the second parameter is alloc_addr
        if(kpi_type == kmem_cache_free){
            alloc_addr = PT_REGS_PARM2(ctx);
        }else if (kpi_type == kfree){
            alloc_addr = PT_REGS_PARM1(ctx);
        }
        key = bpf_look_up_key(addr2key,alloc_addr)
        //locate the object's cache
        cache = bpf_look_up_cache(key2cache,key)
        alloc_size = BPF_CORE_READ(cache,size)
        // get the alloc_size of the object
        bpf_delete_object(alloc_size,alloc_addr)
    }
// attach to allocation site
SEC("kprobe/single_open+0x2a")
int probe_alloc_seq (struct pt_regs* ctx) {
    return alloc_handler(ctx, kmem_cache_alloc);
}
// attach to free site
SEC("kprobe/single_release+0x34")
int probe_free_seq (struct pt_regs* ctx) {
    return free_handler(ctx, kfree);
}
```

Listing 2: A simplified example of a synthesized ebpf program. The to-be-protected object is `seq_operations` which is allocated via `kmem_cache_alloc` in `single_open()` and freed via `kfree` in `single_release`. Note that comments in the code are not synthesized but for illustration purpose.