BUDAlloc: Defeating Use-After-Free Bugs by Decoupling Virtual Address Management from Kernel

Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, and Youngjin Kwon, *KAIST*

https://www.usenix.org/conference/usenixsecurity24/presentation/ahn

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

BUDAlloc: Defeating Use-After-Free Bugs by Decoupling Virtual Address Management from Kernel

Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, Youngjin Kwon

School of Computing, KAIST

Abstract

Use-after-free bugs are an important class of vulnerabilities that often pose serious security threats. To prevent or detect use-after-free bugs, one-time allocators have recently gained attention for their better performance scalability and immediate detection of use-after-free bugs compared to garbage collection approaches. This paper introduces BUDAlloc, a one-time-allocator for detecting and protecting use-after-free bugs in unmodified binaries. The core idea is co-designing a user-level allocator and kernel by separating virtual and physical address management. The user-level allocator manages virtual address layout, eliminating the need for system calls when creating virtual alias, which is essential for reducing internal fragmentation caused by the one-time-allocator. BUDAlloc customizes the kernel page fault handler with eBPF for batching unmap requests when freeing objects. In SPEC CPU 2017, BUDAlloc achieves a 15% performance improvement over DangZero and reduces memory overhead by 61% compared to FFmalloc.

1 Introduction

Memory bugs persist as enduring security vulnerabilities in software systems. Arising from developers' oversights and the use of memory-unsafe languages, these bugs are prime targets for exploitation by attackers. Among the myriad of memory bugs, use-after-free (UAF) vulnerabilities stand out as particularly notorious. Recent research evaluates its significance, ranking UAF bugs seventh among the top 25 most prevalent and impactful vulnerabilities in CWE [25].

Numerous strategies have been proposed to prevent or detect use-after-free (UAF) bugs, including explicit labeling [16, 31, 33, 38], reference counting [29, 39], pointer nullification [28, 43, 46], garbage collection (GC) [17, 24], and one-time-allocator (OTA) [25, 42, 44, 45]. The explicit labeling method involves tagging memory chunks as used or freed, with runtime checks verifying the tags on each access. However, this incurs significant performance overhead. Pointer nullification nullifies the freed objects and their associated virtual addresses. However, tracking dangling pointers also incurs performance overhead. Reference counting, while effective in some cases, is not universally applicable to all C/C++ applications and also suffers from high performance overhead due to runtime checks. GC-based approaches have gained popularity due to their modest performance and memory overhead. Nevertheless, they also come with several limitations.

Firstly, GC-based approaches suffer from poor scalability in multi-threaded applications. For example, MarkUs [17] necessitates concurrent threads to scan the application's memory while causing pauses in the application threads. Although Minesweeper [17] attempts to optimize this issue with linear memory scans, they still encounter stop-the-world delays in concurrent applications. Secondly, applying GC-based approaches to applications using obfuscated pointers, such as pointers with flags or reference counters, is challenging and often results in excessive memory overhead due to false positive detections [17]. Thirdly, the intricate design of GC-based approaches frequently involves severe security flaws, either by design or implementation [44]. Lastly, most garbage collection-based methods quarantine freed pointers and repurpose them once dangling pointers are eliminated. However, this approach is not suitable for accurate detection.

Recently, researchers have gained interest in rethinking the OTA approach to overcome the limitations of the GC-based approach. The fundamental concept behind the one-time-allocator (OTA) is never reusing allocated virtual addresses for subsequent allocations, ensuring each allocation request receives a distinct memory chunk. While OTA doesn't eliminate UAF bugs, it makes them unexploitable. In a basic OTA implementation, each object is allocated on a separate page, which is unmapped when the object is freed. However, this object-per-page allocation leads to significant memory bloat due to internal fragmentation. To address this, OTA incorporates the notion of virtual aliasing [23], where multiple aliases of objects are mapped to a single page [42]. However, the aliased-based OTA introduces a notable performance overhead due to frequent system calls (one syscall for each allocation or free). Recognizing these limitations, recent efforts have revisited OTA to overcome the performance challenges posed by the original approach.

FFmalloc [44] abandons virtual aliasing and uses a group-based memory allocator to aggregate objects of similar sizes and deploy batching when freeing objects. While FFmalloc provides good performance across various workloads, it faces

significant memory bloat, up to 800%, at the cost of forgoing virtual aliasing. Additionally, FFmalloc sacrifices the detectability of UAF bugs, a key advantage of OTA over garbage-collection (GC) based approaches. DangZero [25] revives virtual aliasing to enable precise UAF bug detection. To mitigate system call overhead, DangZero utilizes library OS (LibOS) where the library kernel and an application run in the same protection domain. For protection, the LibOS instances execute within a virtualization. While DangZero shows low memory overhead, it has several drawbacks. Virtualization inherently incurs significant IO overhead due to a complex IO stack. Moreover, DangZero requires substantial engineering effort as it must reimplement kernel features such as demand paging, copy-on-write, and NUMA migration. Notably, the current DangZero prototype lacks support for copy-on-write on fork, necessitating the copying of all allocated memory during fork operations.

This paper introduces BUDAlloc, presenting a new design approach for the practical implementation of OTA. BUDAlloc aims to strike a balance between performance and memory overhead while providing strong UAF bug detectability. BUDAlloc does not compromise compatibility, allowing unmodified binaries to use BUDAlloc by simply replacing the default allocator using `LD_PRELOAD`.

The core idea of BUDAlloc consists of two parts: separating virtual address from in-kernel memory management, and co-designing a one-time-allocator and virtual address management at the user-level. This co-design provides several performance benefits for implementing virtual aliasing. Firstly, creating alias pages in BUDAlloc does not necessitate system calls, as the user-level OTA manages the virtual address layout. Second, BUDAlloc provides good scalability in multi-threaded applications. The Linux kernel employs coarse-grained locks (`mmap_sem`) to protect virtual and physical memory operations, exhibiting scalability bottlenecks when memory system calls are frequently invoked, as is the case in OTA. In BUDAlloc, virtual address operation is independently managed at the user-level, thereby significantly reducing lock-holding times during memory system calls.

The mechanism enabling the collaborative design of user and kernel levels is eBPF. When allocating an object, the user-level OTA generates alias information (policy) as user-level metadata and shares it with the kernel. Using eBPF, BUDAlloc customizes the in-kernel page fault handler to perform virtual to physical mapping (mechanism) according to the shared alias information. This design minimizes the interference with existing kernel implementation, allowing BUDAlloc to reuse the existing kernel implementations such as demand-paging, copy-on-write, and NUMA migration.

Furthermore, BUDAlloc offers an option for selecting protection and precise detection. When freeing an object, BUDAlloc defers unmapping a page containing the object and batches unmapping operations when handling the next page fault. While this introduces a potential window for a dangling reference, our observations indicate that this window is sufficiently small to detect most UAF bugs in practice (Table 3). However, for precise detection, BUDAlloc provides a method to immediately unmap a page upon releasing an object, albeit incurring a performance penalty.

We implement BUDAlloc in both the Linux kernel and a user-level memory allocator from scratch, with 2,870 and 8,259 lines of code respectively. At first, we evaluate the performance and memory overhead of BUDAlloc compared to recent OTAs, FFmalloc [44], and DangZero [25], as well as GC-based MarkUs [17]. In SPEC CPU 2006, compared to DangZero, BUDAlloc demonstrates faster performance than DangZero by 5% even in the full detection mode and 15% in the prevention mode. BUDAlloc shows acceptable memory overhead (30%) and significantly better bug detectability, while providing 13% lower performance. We evaluate the scalability using multithreaded PARSEC 3.0 and show the most scalable performance improvement, even faster performance when using more than 8 threads compared to FFmalloc. We then conduct real-world performance tests using Nginx [11] and Apache [8], demonstrating similar performance and memory overhead to GLIBC without scalability issues, unlike other work.

Finally, we test BUDAlloc's resilience against use-after-free vulnerabilities using six CVEs commonly found in state-of-the-art OTA researches. Even in prevention mode, BUDAlloc instantly detected five out of six UAF bugs due to its high bug-detection precision. Additionally, we employ the Fuzzer and unit tests to assess the robustness of BUDAlloc, discovering no reports exceeding 24 hours in HardsHeap [47] and passing all suites in NIST Juliet tests [12].

This paper makes the following contributions.

- We propose a new design idea to implement OTA by co-designing user and kernel level, reducing system call overheads.
- We present a practical approach to detect many UAF bugs without compromising performance.
- We showcase that the co-design strikes the balance among performance, scalability, memory use, and bug detectability in real workloads.

2 Use-After-Free Bug

Memory-unsafe languages, such as C and C++, are extensively used in the development of software systems, including Redis, operating systems, and web or mobile applications. However, they are easily affected by a series of memory safety issues, such as use-after-free, double-free, and invalid-free errors. Among these, the use-after-free (UAF) bug stands out as one of the most frequently reported and exploited security vulnerabilities. UAF occurs when a program attempts to access a dangling pointer that points to a previously freed object. In severe cases, attackers can hijack the freed object, altering its content with malicious data.

Various research efforts have proposed methods to detect or mitigate use-after-free bugs. One approach involves labeling each memory block to track its allocation and deallocation, and checking the metadata when the pointer is dereferenced [22,31,33,38,46]. However, dynamically checking the pointer references introduces significant overhead. Another strategy is to actively invalidate or block dangling pointers when an object is freed, utilizing garbage collection or dynamic tracking [17,24,28,43]. However, these methods suffer from high CPU utilization and performance degradation, low scalabilities, and occasionally stop the entire program from sweeping the memory space. Finally, approaches using randomized patterns for allocation [34] offer probabilistic detection only.

Detection and protection. Defending against UAF bugs involves both immediately detecting the bugs and protecting against their malicious behavior. Detecting encompasses protection and can be applied for online [9] and offline sanitizer [16], and bug triaging. Usually, protection can operate much faster than detection. Detecting UAF bugs at runtime or through program analysis poses challenges due to the temporal nature of these bugs (i.e. the part where the bug occurs is different from where it actually happens). As a result, existing UAF detection work exhibits unacceptable performance and resource overheads. To the best of our knowledge, no OTA system has simultaneously achieved high bug detectability, good performance, and minimal memory overhead. To address this, we introduce an efficient UAF detection mechanism by co-designing the secure allocator with the kernel using eBPF.

One Time Allocator. One-time-allocator (OTA) [23] has regained popularity, particularly with the extensive address space offered by modern 64-bit architectures. OTA ensures that allocated virtual addresses are never reused, preventing attackers from dereferencing and manipulating a freed object with a dangling pointer. OTA has to allocate a distinct virtual address at every memory allocation and remove accesses to deallocated memory.

Figure 1 shows an overview of the virtual aliasing of OTA. OTA divides the virtual address space into two sections; canonical and alias. OTA uses the canonical address internally, while the application uses the alias address. OTA first allocates a canonical address using the internal memory allocator when the user application requests memory. Subsequently, OTA assigns one or more alias addresses and maps the alias addresses (*VA2, VA3, and VA4* in Figure 1) to the canonical address (*VA1*). Each alias address has a different offset within the same canonical page. OTA can detect UAF bugs immediately after unmapping the alias page from the page table since alias pages are not duplicated or reused. On allocation, OTA returns the alias address to the user application. Upon free, OTA reverses these steps by unmapping the alias address from the page table and freeing the canonical address using the original free operation in the internal memory allocator (e.g., `free`).

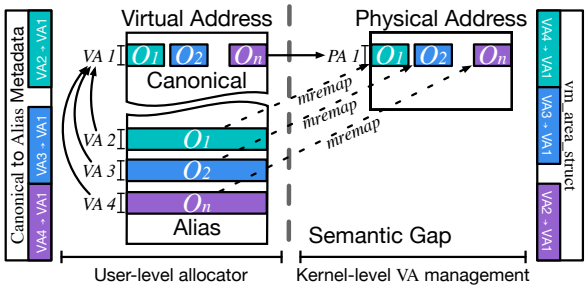

Figure 1: Overview of virtual aliasing.

3 OTA Design Problem

This section analyzes challenges when implementing OTA (§3.1) and discusses the limitations of the current OTA designs: trade-offs in performance, memory footprint, and bug-detect precision (§ 3.2) as well as compatibility problems (§ 3.3). These limitations motivate us to safely co-design the memory allocator with a custom page fault handler.

3.1 Challenge: Semantic Gap

To implement OTA, the widely adopted approach is virtual aliasing [23]. As illustrated in Figure 1, each object has its own virtual alias, with at most one virtual alias accommodated in a virtual page. The *alias-per-page* allocation makes it easy to prevent use-after-free bugs. When an object is freed, OTA disconnects the mapping of the virtual page containing the freed alias object, preventing access to the freed object. However, the alias-per-page allocators suffer from high internal memory fragmentation. Therefore, virtual aliasing makes multiple aliases to be mapped to the same physical page.

OTA records the mappings of alias to canonical page in user-level metadata. According to the alias to canonical mapping, kernel maps alias to physical pages on the page fault. However, this information is not visible to the kernel. We call this problem *semantic gap* in OTA. How to address the semantic gap significantly affects the performance, the size of memory footprint, and bug-detect precision of OTA.

Trade-off 1: Performance and memory footprint. To synchronize the user-level information to the kernel, OTA allocators have to perform system calls (e.g., `mremap` and `unmap`) to bridge the semantic gap. However, frequent system calls cause significant performance overhead. Previous work addressed this problem by deferring `unmap` system calls and batching them, yet they faced notable memory overhead stemming from memory fragmentation [44].

Trade-off 2: Performance and bug-detect precision. Accurate detection of use-after-free bugs closely related with how to handle the `free` operation in OTA. When freeing an object, to detect the reuse of an alias address precisely, OTA has to instantly unmap the alias address from the page table. Unfortunately, this operation requires at least one system call per `free` operations, which incurs significant overhead. Therefore, previous use-after-free works typically favored the prevention

over detection for the performance, despite the feasibility of the detection [17, 24, 44].

3.2 Limitations of Previous OTAs

In this section, we show how the previous OTAs address the trade-off caused by the semantic gap and discuss their limitations. Table 1 summarizes the discussion.

3.2.1 No alias mapping

The most straightforward approach is abandoning alias to canonical mapping, so the semantic gap does not exist. All objects are allocated in canonical address space (i.e., virtual address) directly without having aliases and a canonical page is mapped to a physical page. Therefore, this approach does not need to remap multiple alias addresses to a single canonical page, showing less performance overhead. To prevent use-after-free bugs, this approach must not reuse the allocated canonical address and free a physical page when all objects in a canonical page are freed. Consequently, this approach suffers from high internal fragmentation in physical pages because it cannot reuse empty space in a canonical page (and a physical page). Only when all objects in a canonical page are freed, this OTA frees (unmaps) the physical page mapped to the canonical page, bloating memory footprint significantly when the lifetime of objects varies.

FFmalloc [44] is an example of this approach. It achieves performance close to native memory allocators but introduces significant memory overheads up to 800% in real-world applications [45]. Also, before a physical page is unmapped, FFmalloc cannot detect the use-after-free bug, having the lowest detecting precision.

3.2.2 Syscall-based approach

This approach uses system calls to synchronize the mapping with the kernel, addressing the semantic gap. For allocating an object, syscall-based OTA uses `mremap` to establish alias mapping, and for freeing an object, it issues `mprotect` or `munmap`.

However, while this approach is convenient to implement, syscall-based approach suffers from several limitations. Firstly, this approach incurs high performance overhead due to frequent system calls as `malloc` and `free` require a system call for each. Secondly, even with attempts to serialize and remove locks from the user-level allocator, the frequent mapping and unmapping through the system call leads to high serialization overheads to global kernel locks (e.g., `mmap_sem`) [42, 44]. Thirdly, it results in virtual address fragmentation caused by frequent `mmap` and `munmap`, which in turn fragments the kernel's memory data structure, taking a longer time to find empty address space. We observe that this approach often crashes a process by hitting the Linux kernel's limit of the address fragments (65536) [42].

The state-of-the-art syscall-based OTA is Oscar [42]. To mitigate the overhead of frequent system calls, Oscar speculatively combines freeing and mapping of same-sized objects in one single `mremap` based on the observation that recently freed objects have a high chance of being reused after freeing. Despite that, Oscar suffers from the highest overhead compared to other approaches because it still necessitates at least one system call per `malloc` or `free` operations.

System call batching. Tuning the kernel to support batching system calls could mitigate system call overheads [27, 41]. However, solely batching system calls is not the definitive solution. Firstly, complex system calls like `mmap` or `mremap` typically consume a significant portion of execution time for the kernel software, not the context change. Secondly, the complexity of in-kernel memory metadata also incurs a `vm_area_struct` fragmentation on the OTA. Lastly, even with system call batching, it is inevitable to encounter overheads from the synchronization of kernel metadata such as `mmap_sem` lock. Oscar tried to batch system calls using custom `IOCTL`, but they could not optimize it successfully. They eventually abandoned this approach as, in certain benchmarks, batching system calls led to a slowdown. Oscar analyzes the failure to the challenge of accurately predicting which canonical address will be needed in the future and the side effect of disrupting the favorable memory access patterns of `malloc` by reusing recently freed slots in the canonical address.

3.2.3 LibOS-based approach

Using Library OS, this approach accesses the page table directly while virtualization ensures the protection. The library kernel and OTA run at the same privilege level. OTA controls the virtual to physical mappings without using system calls, thus eliminating the semantic gap problem.

DangZero [25] implements OTA using a Library OS. It manages aliases by directly updating page table mappings and precisely detects any occurrences of use-after-free bugs. However, to safely grant access to page tables, it relies on mechanisms such as Dune [20] or Kernel-Mode-Linux (KML) [30], which introduce fundamental overheads for running on a virtual machine. For instance, Dune shows approximately a 40% performance degradation due to an increased system call overhead [25]. Furthermore, virtualization incurs significant overhead when it comes to performing IO through virtualized storage and network stack [3, 36] as well as address translation overhead due to nested page table translation. The address translation cost is exacerbated by the higher TLB miss rate in OTA where each alias occupies one TLB entry. Another drawback is compatibility because this approach completely bypasses the kernel. Therefore, to be compatible with existing POSIX semantics and APIs, DangZero has to implement complex in-kernel features like fork, demand paging, and copy-on-write for the alias pages, which requires significant efforts to work them completely. Unfortunately, DangZero does not support copy-on-write on fork, exhibiting significant overhead when creating a process. We discuss details in the next section (§3.3).

	Memory Bloat	Syscall Overhead	Scalability	Bug-detect Precision	Compatibility
No alias mapping [44]	Very High	Low	Very High	Very Low	Fully Compatible
Syscall-based [42]	Moderate	Very High	Low	Detector	No COW
LibOS-based [25]	Low	VM overhead	Single thread only	Detector	No COW, proc fs, etc
BUDAlloc-detection	Low	Low	Very High	Detector	Fully Compatible
BUDAlloc-prevention	Low	Very Low	Very High	High	Fully Compatible

Table 1: Comparisons of previous OTAs.

3.3 Compatibility Problem

Fork compatibility. Syscall-based and LibOS-based approaches cannot use copy-on-write. Syscall-based approach necessitates using `MAP_SHARED` for canonical allocations, which changes the semantics of memory to be shared in case of the `fork`. LibOS-based approaches bypass the kernel when creating alias mappings, thereby the kernel is not aware of these pages during `fork`. To ensure `fork` compatibility they have to copy the entire parent address before `fork` without copy-on-write, which increases the latency of `fork` sensitive workloads [35]. In 1GB workloads, the LibOS-based approach takes 1,724ms whereas both our system and a normal Linux process take only 520ms. Fork is not only used for process spawning but also for large databases for consistency snapshot [14], fuzzers, and lambda functions for avoiding initialization overheads [7, 48]. Ensuring scalable performance of `fork()` is crucial for supporting such applications.

Other issues. The LibOS approaches also necessitate the application developers to build their physical address management policies and mechanisms such as `proc fs`, resident set accounting, NUMA-aware placement, or swapping, as the kernel is not aware of allocated alias pages in the page table. For instance, to measure the maximum resident set size of the benchmarks, we have to use the internal accounting mechanisms of DangZero, despite other allocators can conveniently use utility `time` which uses the `proc` filesystem of the kernel.

3.4 Threat Model

We assume a program contains one or more use-after-free bugs, and the attacker can use at least one use-after-free bug, for information leaks, privilege escalation, corrupting data, etc. However, we do not assume that the attacker has the ability to corrupt the OTA allocator. Preventing other classes of memory bugs, such as out-of-bound access or side-channel attacks, are beyond the scope of this work. In prevention mode, we allow crashing at UAF or accessing the old data. In detection mode, we only allow crashing at UAF, rejecting any other class of bugs from UAF. These threat models are compatible with the previous OTA approaches that aim to detect or prevent use-after-free bugs [17, 25, 42, 44].

4 BUDAlloc

BUDAlloc seeks to strike the balance between performance, scalability, memory bloat, and bug-detection precision while providing full compatibility with existing API semantics.

4.1 BUDAlloc Design Overview

Separating address management. At the core design of BUDAlloc lies separating the address management of the kernel into virtual address management and physical address management. The user-level part of BUDAlloc has full control over managing the virtual address space while the kernel controls the physical address management such as page fault handling and the actual page mapping of the designated virtual address. This design allows full compatibility because BUDAlloc reuses well-designed kernel implementation to support existing compatible functions, such as copy-on-write on fork or on-demand paging. Additionally, developers can use the existing interfaces, such as the Linux `proc` file system, to manage applications.

Co-design user-level allocator and kernel. With the separation of address management, BUDAlloc co-designs the user-level allocator and kernel, which removes the duplicated metadata management for the semantic gap. The user-level OTA manages the virtual address, customizing the in-kernel page fault handler safely using BUDAlloc. The custom page fault handler and user-level OTA share the alias to canonical mapping metadata, addressing the semantic gap in OTA. The user-level OTA specifies alias to canonical mapping at the user-level, without the need for expensive exceptions like system calls (policy). Kernel establishes page table mapping according to the metadata while handling page faults (mechanism).

Fine-grained locking. To enhance scalability, BUDAlloc reduces lock contention for managing virtual and physical addresses, which are protected by global locks in the kernel, by shifting virtual address management to the user-level. In addition, BUDAlloc uses fine-grained locking to protect the shared metadata. These design choices are critical for OTA, which incurs intensive lock contentions due to frequent system calls.

4.2 User-level Components of BUDAlloc

As the outcome of co-designing user and kernel space, BUDAlloc consists of user and kernel components. This section introduces the design and optimization of user-level components of BUDAlloc. Figure 2 shows the overall design of BUDAlloc. The user-level allocator incorporates two functions: user-level OTA and virtual address manager (`LibMM`). As an OTA, BUDAlloc uses virtual aliasing. After creating an alias page, BUDAlloc inserts the alias to canonical mapping to the metadata, implemented by trie. The metadata is shared with

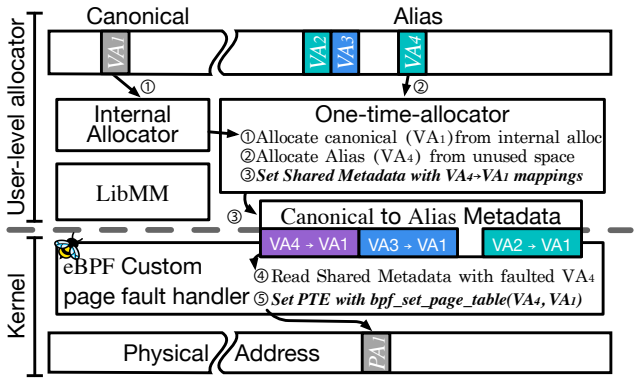

Figure 2: Overview of the BUDAlloc one-time-allocator.

kernel components (§4.2.1). The in-kernel custom page fault handler examines the metadata to establish a mapping from an alias page to a physical page (§4.3). As a virtual address manager, `LibMM` manages canonical address and the virtual address layout of the internal allocator. (§4.2.2).

4.2.1 One Time Allocator

BUDAlloc replaces the existing memory allocators. When a user application requests allocation, BUDAlloc transparently intercepts memory APIs using `LD_PRELOAD`, and relays memory requests to the internal memory allocator.

Allocating an object. The internal memory allocator allocates a new canonical address, which is not exposed to the application. Then, the user-level OTA creates a new virtual alias page for the object. As an OTA, BUDAlloc avoids reusing the alias page in subsequent allocations. Notably, unlike the syscall-based approaches, this step does not require the `mremap` system call because BUDAlloc manages the virtual address space directly in user-level. BUDAlloc inserts the alias-to-canonical mapping into the trie metadata and returns the alias address to the application. Unlike the Linux kernel, which uses a linked list for metadata, the trie metadata does not incur virtual address fragmentation. Upon an alias page fault, BUDAlloc custom page fault handler searches the trie metadata for the canonical address associated with the faulted alias address. If the entry exists and is valid, the custom page fault handler maps the alias page with the same physical address as the canonical address.

Freeing an object. When a program releases the alias object, BUDAlloc scans the trie metadata to find the corresponding canonical address and notifies the internal memory allocator to reuse the canonical. This design is crucial to mitigate internal fragmentation. FFmalloc cannot reuse it because it forgoes virtual aliasing for performance.

As discussed, the degree of bug-detection precision is closely related to how to design freeing objects. To immediately detect UAF bugs, OTA should instantly remove the mapping of freed alias pages. Unfortunately, this requires at least one system call to execute an `unmap` operation. Therefore, BUDAlloc supports two modes for immediate detection and prevention of UAF bugs.

Deferred alias free. BUDAlloc offers a prevention mode, allowing users to defer the freeing operation at the expense of relaxing UAF detection. In this mode, BUDAlloc postpones the unmapping of alias pages from the page table until the occurrence of the next page fault. More precisely, BUDAlloc inserts freed alias addresses into a per-thread ring buffer. When a newly allocated alias address is accessed, a page fault occurs, and the BUDAlloc custom page fault handler unmaps all addresses stored in the ring buffer while handling the page fault. To ensure correctness in deferred freeing, the internal allocator must reuse freed slots of the canonical address after the associated alias pages are removed from the page table. BUDAlloc tracks the states per-thread ring buffer and ensures that there exists no alias to canonical mappings to the freed canonical address before the canonical address is reused by the internal memory allocator.

Due to the deferring, the prevention mode allows a potential window for a dangling reference. However, unlike the previous work [17, 24, 44], the time window is small, which is *min*(time taken to the next page fault, batching delay). We set the batching delay to 10 ms, and dynamically adjust the ring buffer length based on the deallocation rate (e.g., `free`). Given that UAF bugs tend to manifest after a long-term period rather than a short-term [28], BUDAlloc detects almost all UAF bugs in prevention mode as demonstrated in various CVEs (§5.1).

Detecting memory bugs. At the time of freeing an alias address, if the trie entry corresponding to the alias address in the user-level allocator is empty, BUDAlloc regards this as an invalid free or a double free. If successful, the alias-to-canonical mapping is removed from the trie, and during the next page fault, if the custom page fault handler finds no corresponding canonical address in the trie, BUDAlloc reports a use-after-free bug to the user.

4.2.2 Virtual address manager (`LibMM`)

We use `mimalloc` [10] as the internal allocator. The internal allocator uses system calls (e.g., `mmap`, `munmap`, and `madvise`) to allocate and reclaim virtual addresses from the operating systems. BUDAlloc intercepts the memory system calls from the internal allocator and forwards them to `LibMM`. `LibMM` is a compatible layer for managing the internal allocator as the canonical allocator. The canonical allocator remains unaffected by virtual address fragmentation. Thus we manage the `LibMM` using a linked list like the Linux kernel. Upon a page fault, the custom page fault handler determines whether the faulted address is canonical or alias. If it is a canonical page, it maps a physical address to the faulted address. If it is an alias fault, it searches the shared metadata to handle the page fault by policies of the BUDAlloc.

4.2.3 Optimizations

Prefaulting alias pages. OTA allocates the objects to distinct alias pages. Thus, in theory, there should be at least one page

fault per object if batching is not used. However, using BUDAlloc user-defined page fault handler, BUDAlloc can prefault alias pages anticipating upcoming memory access patterns based on the semantics of alias pages. We store the object size along with a canonical address in the shared metadata (trie). For prefaulting, BUDAlloc uses two algorithms based on the object size in the trie entry.

For objects smaller than the page size, BUDAlloc prefaults consecutive alias pages. BUDAlloc manages a reservation pool that stores alias pages of the same size and inserts these alias pages into the trie entries. Upon a page fault, BUDAlloc prefaults the next consecutive alias pages until it reaches the maximum limit (128 in our implementation).

For objects greater than a page size, BUDAlloc takes a heuristic based on the window-based algorithm. The window size determines the prefaulting size of the faulted address. Initially, the window size is a page size, and BUDAlloc adaptively increases the prefaulting window by a factor of two. This reduces the overall page fault rates on a logarithmic scale for the newly allocated objects.

4.3 Kernel Components of BUDAlloc

BUDAlloc extends the kernel to collaborate with the user-level OTA and `LibMM` using eBPF. The design goal is as follows: Firstly, BUDAlloc must be secure and isolated from both the kernel and the other user processes. Secondly, BUDAlloc ensures seamless compatibility with various kernel features, such as fork, clone, on-demand paging, swapping, or NUMA. Thirdly, BUDAlloc facilitates fine-grained locking for scalability.

4.3.1 Decoupling address space

BUDAlloc allows the user-level OTA to establish virtual-to-physical mapping for virtual aliasing. However, directly exposing the layout of physical addresses to users bleaches the security guarantees of the kernel. Instead, BUDAlloc constructs pseudo-physical address, and allows the user-level OTA to use it. The user-level OTA provides a virtual address and pseudo-physical address for aliasing, and the kernel translates the pseudo-physical address to the physical address and finally establishes the mapping between the virtual address and physical address. BUDAlloc reuses the process's page table to convert a pseudo-physical address into a physical address. Pseudo-physical address restricts BUDAlloc to managing only the virtual addresses of the owner process. Therefore, even if the owner process is compromised, it cannot access or modify other processes' memory.

This level of indirection allows decoupling address space between what user-level OTA uses and what kernel manages. Decoupling address space hides the physical address information such as address space layout or page table policies from BUDAlloc. The amount of information accessible to BUDAlloc is identical to what is shown in the Linux kernel (i.e. `/proc/pid/maps`). In addition, by decoupling, the kernel has complete control over the physical address. It transparently applies kernel-specific tasks hidden from the user-level OTA, such as resident-set-size counting or NUMA migration policies. Note that, as BUDAlloc reuses the process's page table, there is no additional space overhead when mapping from pseudo-physical to physical addresses.

Type	API
Page Table	bpf_set_page_table(void *vaddr, size_t len, void *pindx, u64 vm_flags, u64 prot)
	bpf_unset_page_table(void *vaddr, size_t len)
Shared Memory	bpf_uaddr_to_kaddr(void *uaddr, size_t len)

Table 2: A summary of BUDAlloc helper functions used for page table modification and shared memory.

4.3.2 User-Defined Page Fault Handler

As previously explained, the user-level OTA involves sharing alias-to-canonical mapping metadata with the kernel (policy). The customized page fault handler then updates page tables based on this metadata (mechanism), bridging the semantic gap. At its essence, we expand the Linux kernel page fault handler using extended Berkeley Packet Filter (eBPF). eBPF enables users to execute a simple program in the Linux kernel without modifying the source code [6].

Reasons for using eBPF. There are several reasons for our choice of using eBPF for the custom page fault handler. Firstly, eBPF ensures the safety of the eBPF program by rejecting invalid API usages, removing denial-of-service attacks, and ensuring memory and control-flow safety [5]. Secondly, eBPF is efficient, as it statically checks the eBPF program and compiles it into native code using a Just-In-Time Compiler (JIT). Thirdly, eBPF can load custom kernel logic without requiring privileged permission (e.g., `sudo`). Linux offers `CAP_BPF` capability, allowing eBPF programs to be loaded with normal user privilege [1]. Lastly, eBPF supports various toolchains, including C, Python, or even Rust, utilizing the LLVM compiler backend [5, 6].

Custom page fault handler. We implement two helper functions to control the page table and one helper function to share the memory between the kernel and the user. Also, we extended POSIX `mmap` system call to register a virtual address range used for `LibMM` and OTA. Table 2 shows the summary of BUDAlloc helper functions. The user-level OTA can specify the mapping of virtual pages with size `len` to a physical page for aliasing using `bpf_set_page_table`. The specified physical address, `pindx`, serves as a pseudo-physical address. If `pindx` is unused, BUDAlloc maps it to a new zeroed physical page and associates the virtual address with the pseudo-physical address. Upon invoking a custom page fault handler, it populates the faulted address by referencing the shared metadata associated with the handler. Upon completion of the page fault handling, the customized page fault handler returns 0 for success or 1 for failure. In case of failure, the kernel page fault handler sends a `SIGSEGV` to the process.

eBPF programming restrictions. Although eBPF ensures safety through static verification, eBPF restricts the maximum number of instructions (1M for `CAP_BPF`) for ensuring termination of an eBPF program. Despite that, we are able to implement the required functionalities within the restrictions. For looping, we leverage the recent eBPF features, `bpf_loop` [2] to implement iterations with unknown count or complex branches. However, we need to address two implementation challenges. Firstly, eBPF lacks features for flexible shared mapping between the user and the kernel. To overcome this limitation, we introduce new helper functions (Table 2). Specifically, `bpf_uaddr_to_addr` is utilized to create a shared map between the kernel and the user, ensuring the validity of the resulting kernel address. Shared memory between the user and the kernel shares the same physical address but has different virtual addresses to maintain separation between the kernel and the user. BUDAlloc transparently ensures the safety and validity of the shared mappings in case of `fork` or `unmap`. BUDAlloc utilizes the helper function for sharing the metadata between the custom page fault handler and the user-level allocator. Secondly, eBPF supports only atomic operations for the synchronization. Therefore, we co-design the custom page fault handler to control the synchronization only using atomic operations (§4.4).

4.3.3 Confining logic errors

BUDAlloc ensures the ordering and atomicity of the helper functions to protect race conditions on the page table modifications by a reader-writer lock. While BUDAlloc cannot prevent logic errors in eBPF, such as specifying an incorrect page when creating an alias, or denial of service attack by refusing to handle page fault, these errors are isolated from other processes by the safety guarantee of eBPF. In BUDAlloc, an eBPF program cannot modify page tables of other processes because BUDAlloc helper functions verify the current pseudo-physical address belongs to the process. Also, we ensure the Translation Lookaside Buffer (TLB) are flushed before and after page table modifications in an eBPF program. When a helper function carries out page table modifications kernel flushes the TLB without requiring explicit TLB flush operations from an eBPF program. While this may be a conservative approach that rules out optimizations of coalescing TLB flushes, we prioritize the correctness and safety of page table updates by leveraging well-debugged kernel code.

4.4 Scalability

The BUDAlloc user-level allocator is optimized for multi-threading scalability, using fine-grained locking and lock-free data structure. This is particularly important in the OTA, as it frequently incurs metadata update operations, stressing the scalability bottleneck due to coarse-grained kernel locks (e.g., the `mmap_sem` in the Linux `mm_struct`) [4].

Opportunity. By decoupling virtual address and physical memory management, BUDAlloc reduces the contention window of coarse-grained kernel locks because virtual address operations such as finding empty virtual address and returning freed virtual address range are moved to the user-level, reducing lock contentions. Furthermore, BUDAlloc allows the user to implement fine-grained locking using the semantics of the user-level allocator. This user-centric approach allows for greater flexibility in deploying user-provided data structures instead of using the existing kernel ones, enabling optimized co-designing locks to safeguard critical sections.

Shared trie metadata. We optimize the shared trie data structure between the in-kernel custom page fault handler and the user-level OTA. Given that both frequently access this metadata, ensuring scalability is essential. The trie entry comprises four levels, each using 48 bits of virtual addresses as an index. We experiment both using global locks and reference-counting-based lockless mechanisms for the trie, and discover that combining the two mechanisms yield the best performance due to the locality of the alias addresses. We assign the last level of the trie to each thread to hold per thread pool of consecutive alias addresses, typically accessed by the same thread due to the spatial locality. Unlike the last-level entry, which stores the canonical address, upper-level entries store the next-level entry, which remains unmodified during operations. Consequently, we adopt different locking mechanisms based on the level of the trie entry.

The shared trie dynamically allocates and reclaims each level, maintaining consistency through reference counting for upper-level nodes (L1-L3) and spinlock for leaf nodes (L4). A thread increases the reference count (using atomic operations) for upper-level nodes before accessing them and decreases the count upon completing trie access. BUDAlloc frees a node if its reference count reaches zero, ensuring that a thread doesn't access a dangling (freed) node. Since leaf nodes' contents can be modified, BUDAlloc uses spinlocks to serialize accesses to a leaf node. These spinlocks are shared among user threads and eBPF code.

However, the BUDAlloc custom page fault handler cannot acquire spinlock, since eBPF does not allow unbounded spinning. Therefore, we use `trylock` inside the BUDAlloc custom page fault handler and reattempt the page fault after a failed lock acquisition.

Deferred free list. In prevention mode, for scalability, BUDAlloc uses a per-thread ring buffer to list deferred alias pages, protected by per-thread spinlock. The ring buffer is shared with the kernel. To ensure the safety of deferred freeing, when a thread frees an alias and reuses the associated canonical address, the page fault handler must unmap the freed alias page before the thread uses the new object using the reused canonical address. To do this, BUDAlloc marks the trie entry of the new alias with the thread index of the per-thread ring buffer. In the rare scenario where a BUDAlloc custom page fault handler cannot acquire the lock using `trylock` and trie entry is marked, it retries the page fault to guarantee the previous

mapping is invalidated. If it is not marked, it skips the deferred free and proceeds with processing the page fault. In detection mode, this logic is not required because detection mode does not defer freeing an alias page.

Supporting fork. When a thread initiates a fork system call, other threads may interact concurrently with the user-level OTA. Therefore, it may end up with inconsistent lock states before and after the fork. Therefore, we implement a global read-write lock to protect against invalid fork states. Before fork operation, the user-level OTA releases all holding spin locks and awaits the completion of the fork using the read-write lock. Similar mechanisms are commonly used by other allocators to support fork operations safely [19].

4.5 Compatibility

While BUDAlloc moves virtual address management to the user-level, BUDAlloc maintains compatibility of existing kernel features such as copy-on-write, demand paging, and the `proc` file system. Therefore, BUDAlloc runs unmodified binaries by replacing its default memory allocator using `LD_PRELOAD`. Nevertheless, some parts of the Linux kernel are closely integrated with both physical and virtual management. We outline two specific cases of such scenarios and describe how to tackle these challenges.

VA-Chain reverse map. The operating system traditionally maintains a reverse mapping that locates a virtual address (address of a PTE) of a physical page, crucial for physical address management tasks like swapping, migration, compaction, and copy-on-write. Linux uses an object-based reverse map, tightly coupled with the metadata for virtual address management, `vm_area_struct`. However, BUDAlloc does not use the metadata because it bypasses the virtual address management in kernel. To address this, we revise a VA-chain-based reverse map, utilizing the unused fields in Linux's `struct page`. Unlike Linux's PTE-chain, which stores references to PTEs in the page table from a page, the VA-chain stores virtual addresses of a page using double-linked lists. The reverse mapping information is directly stored in the struct page, offering a more flexible solution for BUDAlloc. It consumes 16 bytes per mapping. If virtual addresses are allocated contiguously, the memory overhead for the reverse map is only 16 bytes. In our evaluation, the VA-based reverse map shows acceptable memory overhead in real workloads (§5.3).

PTE reference count. If the last entries in the page table are released, the kernel deallocates the upper level of the page table entry to remove unused page table entries. The Linux kernel achieves this by utilizing the `vm_area_struct` to selectively clear the last-level page table entries using the virtual address allocation information. To clear these last-level entries without explicit knowledge of `vm_area_struct`, we employ reference counting. BUDAlloc reuses the `struct page` field to keep track of the references of the PTE entries. BUDAlloc helper function increases the reference count (using atomic operations) upon setting a PTE entry and decreases the count upon unsetting a PTE entry. When the reference count reaches zero, BUDAlloc clears the upper-level page table entry. By leveraging the reference count, BUDAlloc effectively manages the page table, ensuring efficient memory usage.

4.5.1 Compatibility Study

On demand paging. By the design of decoupled address space (§ 4.3.1), BUDAlloc easily supports demand paging. The user-level OTA specifies alias and canonical mapping in the pseudo-physical address, so it does not give any restrictions for physical memory management. Therefore, BUDAlloc populates the page table entry at the page fault.

Copy on write. Both syscall-based and LibOS-based approaches face challenges when it comes to supporting copy-on-write, as discussed in §3.3. Therefore, they copy the entire address space during a fork, which causes significant overhead in an application with a large memory footprints. In contrast, BUDAlloc doesn't encounter these issues. Like the demand paging, decoupled address management allows transparent kernel physical management during fork operations. Note that, users can implement their fork logic using eBPF. By default, BUDAlloc allows the users to reuse the existing copy-on-write implementation in the kernel.

Proc file system. BUDAlloc supports `proc` file system to monitor memory states. In contrast, DangZero requires implementing supplemental mechanisms for replacing `proc` system. For instance, while DangZero necessitates custom logic to monitor memory usage, BUDAlloc transparently supports utility `time`, which internally use `proc` file system.

4.6 Discussion

BUDAlloc shares the fundamental limitation of OTA, increased Translation Lookaside Buffer (TLB) pressure and cache line misses. Unlike non-OTA, which reuses freed objects in subsequent allocations, BUDAlloc necessitates the allocation of new alias pages for each allocation, resulting in reduced cache line hits and increased TLB pressure. Furthermore, since it refrains from recycling freed virtual addresses, BUDAlloc may eventually exhaust virtual addresses. FFmalloc also suffers from the same problem. DangZero uses garbage collection for reusing alias addresses with significant performance overhead. BUDAlloc can incorporate and optimize the same mechanism. We leave this as future work.

4.7 Security Analysis

Attackers who compromise processes using BUDAlloc cannot attack other processes or kernel. Firstly, BUDAlloc isolates pseudo-physical addresses between processes. This ensures that BUDAlloc can manage the virtual address of the owner process only. Even if the owner process is compromised, it cannot access or modify virtual addresses of other processes or kernel space. Additionally, a pseudo-physical address hides the secure information of the physical address, revealing only

what the Linux kernel exposes to the user space. This ensures that no kernel address layout information or sensitive policies, such as Kernel Address Space Layout Randomization (KASLR), are exposed to user space program.

Secondly, BUDAlloc uses eBPF [5] for isolating custom page fault handler from the kernel or other user processes. BUDAlloc requires `CAP_BPF` capability as a normal user, which prevents access to arbitrary kernel memory and leakage of kernel pointers to users, and allows isolation of user space processes [1]. eBPF ensures the safety of BUDAlloc through static analysis using a static verifier. We disable the eBPF *bypass_spec_v1* feature, which incorrectly rejects BUDAlloc due to false positives during verification. Recent CPU mitigates speculative side-channel attacks [26], and fixing these false positives in the verifier is beyond the scope of this paper. However, bugs in the eBPF verifier or helper functions, including additional helper functions in Table 2, could undermine isolation. BUDAlloc trusts the eBPF verifier and helper functions, and these bugs are not part of the BUDAlloc threat model.

5 Evaluation

We evaluate BUDAlloc in terms of security and performance. For security evaluation, we perform CVE analysis, NIST Juliet test suite [12] and HardsHeap [47]. For performance evaluation, we evaluate BUDAlloc with SPEC CPU 2006, SPEC CPU 2017, PARSEC 3.0, Apache, and Nginx Webserver. We implement BUDAlloc on Linux 6.4.0 by 2870 lines of code, and testing with Intel(R) Xeon(R) Gold 5220R CPU at 2.2GHz with 24 cores, 172GB DRAM - 2666 MHZ, 512 GB SSD, and 10-Gigabit Network Connection. In all the experiments, we disable hyper-threading, CPU power-saving states, and frequency scaling to reduce the variance. We use Non-Uniform Memory Access (NUMA) in the PARSEC 3.0 benchmarks (§ 5.5) to fully utilize all 48 cores in the motherboard. We use `time` to get the resident set size (RSS) and total execution time except DangZero. We set the default configuration for the other memory allocators for all evaluations. We use a KVM virtual machine to evaluate test cases on DangZero [25], as it is the default option for running Kernel-Mode-Linux in DangZero.

5.1 Security Evaluation

CVE Analysis. To evaluate the robustness of BUDAlloc compared to other OTA systems, we evaluate a set of Common Vulnerabilities and Exposures (CVEs) commonly referenced in the recent OTA papers [25, 44, 45]. These CVEs address real-world applications' use-after-free bugs in PHP, Python, mruby, and libmimedir. Additionally, we expand our test sets to include the latest real-world exploits of PHP, Python, mruby, our own UAF real-world corpus, and UAFBench [15]. Table 3 shows our protection results: BUDAlloc successfully defends against all attempted attacks.

As predicted, in the case of FFmalloc, it is successful in preventing use-after-free (UAF) bugs, but fails to detect most of the UAF bugs, since FFmalloc delays the unmapping of canonical pages until consecutive pages are freed. However, in contrast, BUDAlloc, even with deferred free enabled (BUDAlloc-p for prevention), successfully detected all UAF bugs in 29 out of 30 separate tests. This capability comes from the detection time interval being short enough to identify the bugs. Furthermore, a user can easily switch to full detection by deactivating deferred free. In this configuration, BUDAlloc successfully detects attempted attacks (BUDAlloc-d for detection). In contrast to other OTA research [25, 44], BUDAlloc achieves near-complete detection even in prevention mode.

Vulnerability	Program	BUDAlloc-p	BUDAlloc-d	FFmalloc	DangZero
UAFBench					
CVE-2016-3189	bzip2	●	●	◑	●
*CVE-2016-4487	cxxfilt	●	●	●	●
CVE-2017-10686	nasm	●	●	◑	●
CVE-2018-10685	lrzip	●	●	◑	●
CVE-2018-11496	lrzip	●	●	◑	●
*CVE-2018-11416	jpegoptim	●	●	●	●
CVE-2018-20623	readelf	●	●	◑	●
*CVE-2019-20633	patch	●	●	●	●
*CVE-2019-6455	rec2csv	●	●	●	●
Issue 74	giflib	●	●	◑	●
*Issue 122	gifsicle	●	●	●	●
Issue 73	mjs	●	●	◑	●
Issue 78	mjs	●	●	◑	●
Issue 91	yasm	●	●	◑	●
ffmalloc & DangZero					
CVE-2015-2787	PHP	●	●	◑	●
*CVE-2015-3205	libmimedir	●	●	●	●
CVE-2015-6835	PHP	●	●	◑	●
CVE-2016-5773	PHP	●	●	◑	●
Issue 3515	mruby	●	●	◑	●
Issue 24613	Python	●	●	◑	●
Exploit Database					
CVE-2019-6076	Lua	●	●	◑	●
CVE-2019-7703	Binaryen	●	●	◑	●
CVE-2019-8343	nasm	●	●	◑	●
CVE-2019-17582	libzip	●	●	◑	●
CVE-2020-24346	nginx	●	●	◑	●
CVE-2022-1934	mruby	●	●	◑	●
CVE-2022-1106	mruby	◑	●	◑	●
CVE-2022-35164	LibreDWG	●	●	◑	●
*BUG-66783	PHP	●	●	●	●
BUG-80927	PHP	●	●	◑	●
●: Detect UAF bug			◑: Prevent UAF bug		

Table 3: Summary for BUDAlloc-d (detection), BUDAlloc-p (prevention), DangZero, and FFmalloc. We use default settings for the previous works. Benchmarks with a star(*) involve a precondition leading to a UAF bug, which is detected and aborted in FFmalloc and BUDAlloc during arbitrary or double `free()`.

NIST Juliet Test Suite. We evaluate the robustness of BUDAlloc using the NIST Juliet Test Suite v1.3 [12]. This test suite comprises over one hundred unit tests for use-after-free (CWE 416) and double-free (CWE 415) bugs, including cases classified as false positives and false negatives. BUDAlloc successfully detects all bugs in detection mode and prevents all bugs without encountering any false positives.

HardsHeap. HardsHeap [47] is an automated tool designed for fuzzing secure allocators. We apply HardsHeap to check the resilience of BUDAlloc against UAF bugs. After running for more than 168 hours, HardsHeap detected no issues or errors. This empirical experiment shows the hardness of the BUDAlloc one-time-allocator.

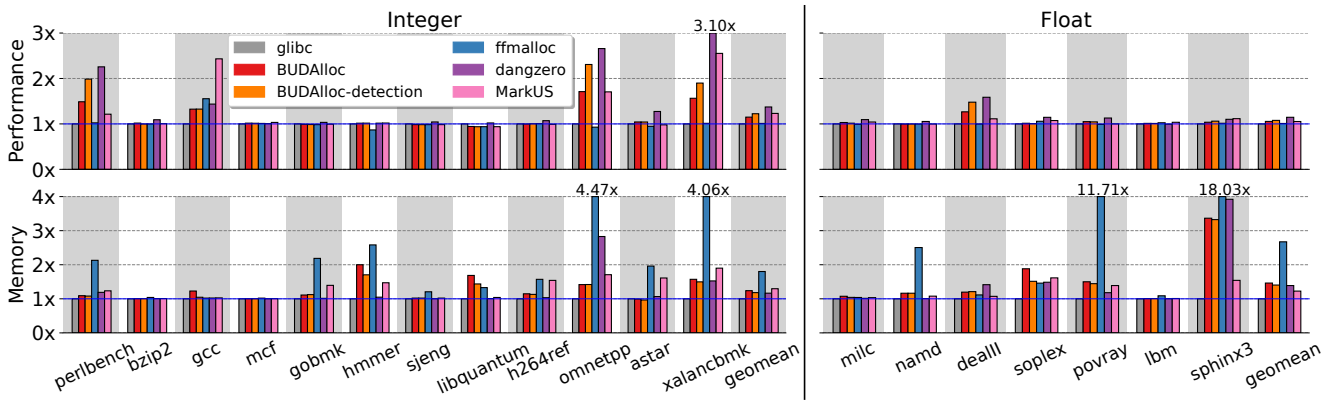

Figure 3: Overhead of SPEC CPU2006 compared with the GLIBC memory allocator. 1x means no overhead. Lower is better.

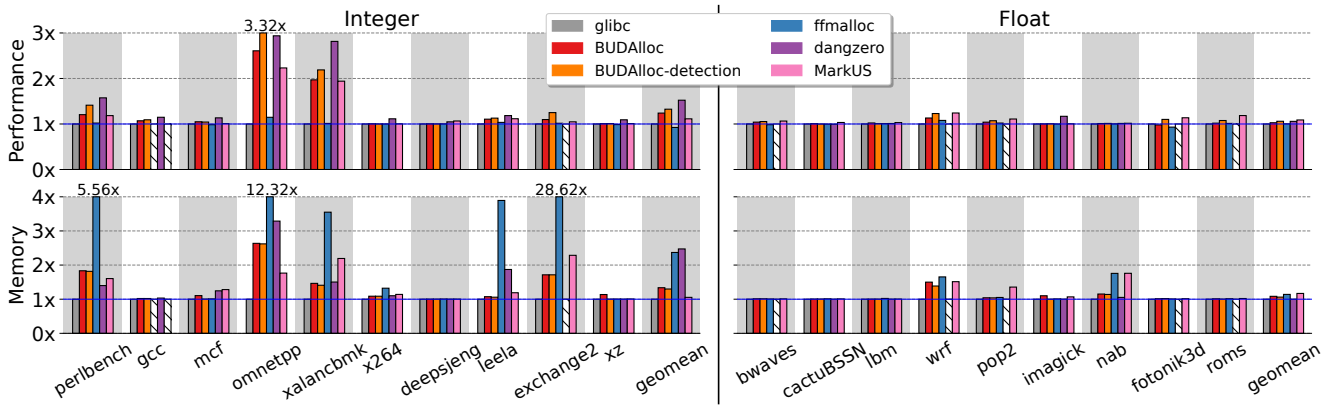

Figure 4: Overhead of SPEC CPU2017 compared with the GLIBC memory allocator. 1x means no overhead. A white bar indicates that a specific allocator did not run. Lower is better.

5.2 Performance Evaluation

We evaluate BUDAlloc using widely adopted benchmarks in OTA research and real-world programs to analyze and compare BUDAlloc performance overheads. We select four sets of benchmarks; SPEC CPU 2006 and SPEC CPU 2017 for single-threaded applications (§5.3), PARSEC 3.0 for multi-threaded applications (§5.4), Apache and Nginx Web server for real-world applications (§5.5). We compare the overhead of BUDAlloc on SPEC CPU 2006 and SPEC CPU 2017 with MarkUs [17], DangZero [25], and FFmalloc [44]. To achieve optimal performance in DangZero, we choose to disable an alias reclaim for a fair comparison. SPEC CPU 2017 contains workloads using a Fortran library, which requires multi-threading. Unfortunately, DangZero lacks multithreading support, causing crashes in several SPEC CPU 2017 benchmarks, specifically `bwaves`, `wrf`, `pop2`, `exchange2`, `fotonik3d`, and `roms`. Also, FFmalloc and MarkUs failed to execute `gcc`.

5.3 Single-threaded Benchmarks

Table 4 summarizes the average performance and memory overhead for five iterations of the commonly executable SPEC CPU benchmarks. We use `glibc` as a performance baseline.

5.3.1 SPEC CPU 2006

System	SPEC CPU 2006		SPEC CPU 2017	
	Perf.	Mem	Perf.	Mem
BUDAlloc-p	1.11×	1.31×	1.18×	1.24×
BUDAlloc-d	1.16×	1.25×	1.23×	1.20×
DangZero	1.28×	1.24×	1.31×	1.27×
FFmalloc	1.01×	2.08×	1.01×	1.90×
MarkUs	1.16×	1.27×	1.17×	1.28×

Table 4: Normalized execution time (perf) and memory usage (mem) in SPEC CPU 2006 and 2017 except `gcc` and benchmarks using Fortran library. Lower is better.

Performance overhead. BUDAlloc shows a moderate performance overhead compared to other systems, with a geometric mean of 1.11× for BUDAlloc in prevention mode, 1.16× for BUDAlloc in detection mode, 1.28× for DangZero, 1.01× for FFmalloc, and 1.16× for MarkUs in SPEC CPU 2006. DangZero reports a higher geometric mean overhead compared to BUDAlloc. This is because, unlike BUDAlloc, DangZero requires virtualization for direct page table access, introducing consistent virtualization overhead across all workloads. Importantly, the performance overhead of DangZero is slower than BUDAlloc in the detection mode (BUDAlloc-d) which detects

every UAF bug like DangZero. While FFmalloc reports almost no overhead in SPEC CPU 2006, it exhibits significantly high memory overhead, as mentioned later. Fundamentally, FFmalloc offers faster performance than detectors like BUDAlloc. FFmalloc does not manage alias-to-canonical mappings, groups small-sized objects into pages to reduce TLB misses, and does not unset a page table after freeing. However, this strategy in FFmalloc leads to delayed detection of UAF bugs and higher memory overheads compared to BUDAlloc. Notably, BUDAlloc is faster than FFmalloc in the `gcc` workload, which has the highest memory operations per second, causing frequent system calls in FFmalloc. Unlike FFmalloc, BUDAlloc manages system calls in the user space, resulting in almost zero system call overhead during high-frequency allocation. It shows comparable results to DangZero's direct page table modification with virtualization. Also, when we compare BUDAlloc with the syscall-based approach Oscar [42], which reports a geometric mean of 1.4× overhead, BUDAlloc demonstrates faster performance.

Memory overhead. Figure 3 shows the memory overhead with default configurations in previous works. BUDAlloc demonstrates an acceptable overhead of geometric mean 1.31× in prevention mode and 1.25× in detection mode. In contrast, FFmalloc exhibits approximately 2.08× memory overhead in SPEC CPU 2006. Notably, FFmalloc experiences significant peaks in malloc-intensive benchmarks such as `omnetpp`, `xalacbmk`, `povray`, and `sphinx3`. This is attributed to high memory fragmentation. To mitigate system call overhead, FFmalloc allocates memory in chunks and delays memory release until at least 8 consecutive chunks can be freed. MarkUs and DangZero exhibit similar memory overhead to BUDAlloc. We find that `sphinx3` incurs the highest memory overhead among all workloads due to its highly fragmented alias address spaces from frequently allocated small chunks with mostly unused pages. However, among the three OTA-based approaches, BUDAlloc demonstrates the smallest memory overhead in `sphinx3`.

Performance breakdown. In Figure 5, we breakdown the performance overhead of SPEC CPU 2006 to discover the impact of BUDAlloc on both user and kernel stack. Notably, allocating aliases incurs less overhead than freeing them, thanks to the user and kernel co-design that avoids system calls for alias allocation. The performance of user-level parts is sensitive to the application memory allocation patterns. For example, in the `gcc` benchmark, the application frequently allocates objects, which puts pressure on the internal memory allocator when allocating canonical address. However, due to the semantic-aware prefaulting, overall kernel time is relatively small in the `gcc`. In the kernel, we observed that the most overhead comes from modifying page table entries. Flushing the TLB incurs a small geomean overhead of 0.09%, less than 2.7% in our implementation for the challenging `omnetpp` workload. This workload puts high TLB pressure on the OTA

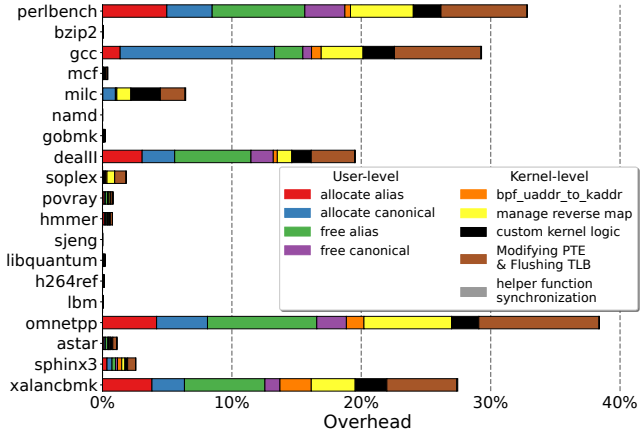

Figure 5: SPEC CPU 2006 performance breakdown.

due to its memory usage pattern. To modify a page table entry, BUDAlloc helper functions traverse the page table iteratively, requiring one iteration per object. Reverse mapping incurs the second-largest overhead, taking up to 6.7% in `omnetpp` and 0.11% on average, mainly due to traversing list entries.

5.3.2 SPEC CPU 2017

BUDAlloc exhibits a moderate performance overhead and efficient memory use compared to previous approaches in SPEC CPU 2017, which aligns with the result from SPEC CPU 2006. The performance overhead of SPEC CPU 2017 is 1.18× for BUDAlloc in prevention mode and 1.23× for BUDAlloc in detection mode while DangZero and MarkUs suffer from 1.31 × and 1.17× slowdown respectively. In terms of memory overhead, BUDAlloc prevention incurs 1.24×, BUDAlloc detection 1.20×, DangZero 1.27×, FFmalloc 1.90×, and MarkUs 1.28×. In the `gcc` benchmark, BUDAlloc shows faster performance (1.07× in prevention and 1.09× in detection) compared to DangZero (1.14×). For benchmarks that use the multithreading Fortran library, the performance overhead is as follows: BUDAlloc in prevention mode is 1.05×, BUDAlloc in detection mode 1.10×, FFmalloc 1.0×, and MarkUs 1.11×. The memory overhead is as follows: BUDAlloc in prevention mode 1.16×, BUDAlloc in detection mode 1.14×, FFmalloc 1.75×, and MarkUs 1.51×. We assume that in the multithreading Fortran library, MarkUs shows the worst performance overhead due to garbage collection. In `omnetpp_s`, BUDAlloc experiences higher performance overhead (2.61×) compared to SPEC CPU 2006, mainly due to the frequent deallocation of small objects in `omnetpp_s`. This increases TLB pressure and unmapping page table overhead, which is fundamentally unavoidable in OTA. In such workloads, BUDAlloc shows distinct trade-offs in the performance and memory overhead; FFmalloc shows a smaller overhead than BUDAlloc in performance, but memory use is 12.32× while BUDAlloc shows 2.8×. Compared to MarkUs, BUDAlloc has a more aggressive memory deallocation policy for small memory objects than MarkUs, resulting in lower performance but less

memory overhead in SPEC CPU 2017.

5.4 Multi-threaded Benchmarks

The PARSEC workload contains a suite of parallel programs for evaluating the performance of multiprocessor systems. Despite its widespread use in literature, the PARSEC 3.0 benchmarks are no longer actively maintained. Consequently, we had to exclude several workloads from PARSEC 3.0 due to issues such as compiled binaries hanging on our system when using GLIBC, as reported in previous studies [44]. Our experiments involved running 15 benchmarks across six different core counts: 1, 2, 4, 8, 16, 32. We evaluate the performance using BUDAlloc, GLIBC, FFmalloc, and MarkUs. We are unable to include DangZero in our evaluations because it lacks support for multithreading applications. Additionally, we omit any failed executions and represent them as white bars in the results. For `canneal`, to ensure an accurate comparison of scalability, we adjust the region of interest (ROI) to exclude the single-threaded parts, which accounted for nearly 85% of the execution time in GLIBC with 32 threads.

To analyze the data, we partition the workloads into CPU-intensive and memory-intensive workloads based on the number of allocation frequency. The highest allocation rate in CPU-intensive workloads is `facesim`, 4,619 per second, while the lowest rate in Allocator intensive workloads is `vips`, 23,343 per second, about 5× higher than `facesim`.

CPU Intensive workloads. In the `blacksholes`, `bodytrack`, `facesim`, `ferret`, `fluidanimate`, `freqmine`, `netferret`, `netstreamcluster`, `streamcluster`, and `x264` workloads, BUDAlloc, FFmalloc, and MarkUs show similar results since these workloads do not heavily utilize memory allocators.

Allocator Intensive workloads. In allocator-intensive workloads, such as `canneal`, `dedup`, `netdedup`, `swaptions`, and `vips`, BUDAlloc shows scalable results, even outperforming FFmalloc in some workloads. Figure 7 illustrates the overall geometric mean of performance improvement normalized by the glibc single thread. FFmalloc exhibits optimal performance with fewer than 4 threads, but BUDAlloc surpasses FFmalloc with more than 8 threads. We observe that FFmalloc eventually encounters bottlenecks due to frequent kernel system calls, which causes the in-kernel global lock contention in the multi-threaded workload. As shown in Figure 6, with more than 32 threads, FFmalloc experiences a significant performance drop in `dedup`, `netdedup`, and `vips`. These benchmarks heavily stress the memory allocator and invoke FFmalloc frequently call `mmap` and `munmap`, which hold a global lock in the kernel.

MarkUs shows the worst performance in these workloads despite similar performance overhead in single-threaded benchmarks with BUDAlloc. These outcomes are unsurprising, as GC-based systems like MarkUs typically involve synchronization between the GC thread and the main thread, which can be severed when there is no dedicated core for offloading. However, BUDAlloc bridges the semantics of user space to manage the metadata lock, enhancing OTA performance in multithreading applications despite the increased bug-detection precision which results in higher page table modification rates.

In `swaptions`, despite the scalability advantages of the BUDAlloc design, BUDAlloc shows worse performance than FFmalloc. This is due to `swaptions` frequently allocating and freeing large objects, placing significant stress on the alias to canonical mapping in OTA. In BUDAlloc, this results in page faults whenever a memory object is accessed and frees the previous alias page for precise bug detection. FFmalloc mitigates these overheads by using large chunks of memory, at the cost of losing bug-detect precision. However, this approach could be configured differently. We conduct tests on `swaptions` using the same logic as FFmalloc for handling large objects. With this configuration, BUDAlloc demonstrates improved performance by approximately 38%, surpassing FFmalloc's performance by about 13% while maintaining a similar memory overhead as BUDAlloc. However, adopting this configuration may lead to significant memory overhead in other applications, similar to FFmalloc. Consequently, we adhere to our original BUDAlloc model for managing large objects.

5.5 Apache and Nginx

We evaluate the performance and memory overhead using the real-world workload of the Apache Web Browser (version 2.4.58) [8] and Nginx [11]. Apache and Nginx Web servers are widely adopted for serving HTTP servers. We connected two Linux machines via an internal 10G network, directly linked by the network interface cards. For the evaluation, we use 16 maximum processes with a 64 Kbytes test set for the server. We tested the benchmark on the GLIBC, BUDAlloc, FFmalloc, MarkUs, and DangZero. Apache adopts a thread-centric approach for the concurrent connections, whereas Nginx adopts an event-driven model using a process. We check the scalability of BUDAlloc in Apache, and memory overhead in Nginx. Unfortunately, we failed to execute DangZero under vhost-net due to kernel panic. We assume that the old kernel version Linux 4.0.0 has some issues with vhost-net which is used by DangZero. Additionally, we encountered null pointer dereference bugs that prevented the execution of Apache, even when using the `pre-fork` mode. As a result, we opted to exclude DangZero from the Apache evaluation.

Apache. As shown in Figure 8, BUDAlloc demonstrates comparable performance, latency, and memory overhead compared to GLIBC. However, FFmalloc shows approximately 8× higher memory usage because Apache frequently allocates and deallocates small objects to handle incoming packets, leading internal allocator fragmentation. MarkUs experiences a significant performance drop under heavy loads after 200 concurrent client connections, as Apache stresses worker threads to serve concurrent client connections, resulting in contention between mark-and-sweep threads and application threads.

Nginx. Unlike Apache, Nginx uses a worker process model

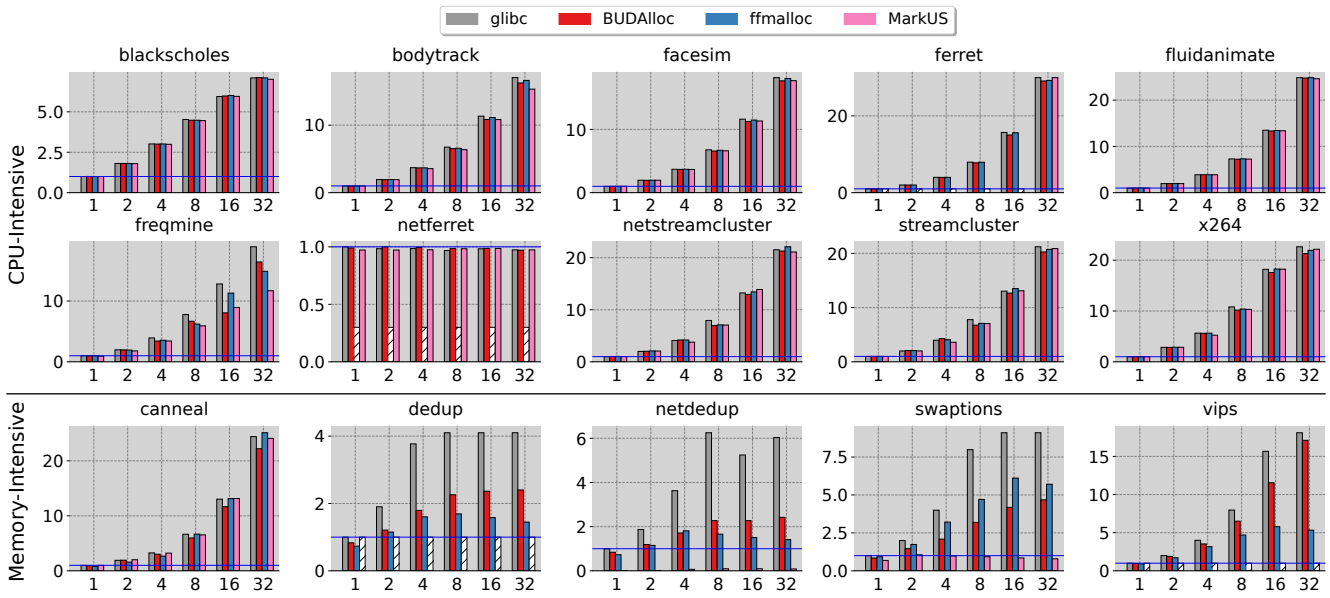

Figure 6: Speedups of PARSEC 3.0 based on the number of threads (higher is better). Performance is normalized to the GLIBC single thread. A white bar indicates that a specific allocator did not run.

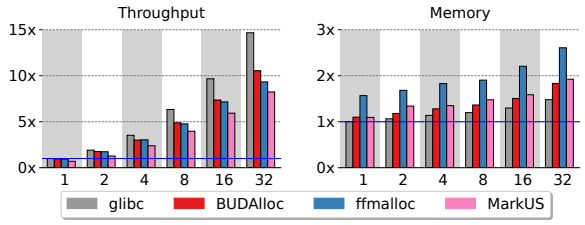

Figure 7: Normalized overall geometric mean for all executable workloads in PARSEC 3.0 with GLIBC single thread. BUDAlloc shows the least memory overhead, and highest scalability, overtaking FFmalloc in more than 8 threads. We replace crashed workload performance with the one of GLIBC to compute the overall geometric mean.

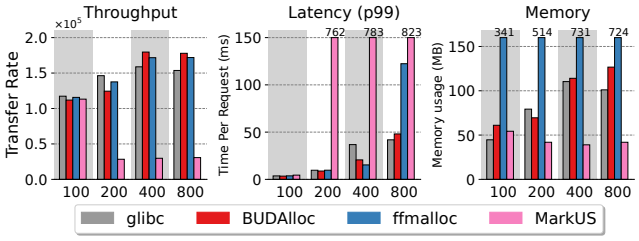

Figure 8: Apache throughput (Requests/sec) and average latency (p99) with varying concurrent connections.

by spawning processes. For Nginx benchmarks, we employed default settings with 400 concurrent connections. BUDAlloc, MarkUs, and FFmalloc exhibited comparable results with GLIBC at 0.98×, 1.04×, and 0.95×, respectively, without significant contention in MarkUs. However, we observed high memory overhead in FFmalloc (7×) and MarkUs (154×), unlike BUDAlloc (1.1×). Unsurprisingly, FFmalloc showed high memory overhead in the malloc-stress settings. Interestingly, MarkUs exhibited the highest peak in memory usage on the Nginx web server. Upon investigation, we believe this may be attributed to memory leaks resulting from false positive detection in MarkUs. This finding aligns with the prior research [25, 45]. DangZero shows around 8.9× performance overhead compared with GLIBC, due to virtualization over the network I/O stack using `virtio` [37].

In summary, only BUDAlloc demonstrates similar performance, memory consumption, and scalability compared to the baselines GLIBC, even with high bug-detecting precision.

6 Related work

Secure heap allocator. Existing work on creating secure heap allocators, such as OpenBSD [13], Cling [18], and DieHarder [34], typically focuses on protecting the metadata of heap allocators by maintaining a bitmap to store allocation states called BIBOP allocator (Big Bag of Pages) [40]. However, BIBOP allocators introduce significant performance overheads from bitmap tracking and synchronization overheads for multithreading applications. FreeGuard [40] solves the performance problem of the BIBOP allocator using a novel combination of both free list (sequential) allocator and BIBOP allocator. However, it fails to protect against use-after-free bugs and still incurs unacceptable memory overhead in certain applications. In contrast to bitmap-based allocators, BUDAlloc maintains free lists for better performance while effectively preventing metadata corruption, by deferring canonical frees until the next page fault handler.

Use-after-free prevention. There are trains of work for preventing use-after-free bugs. These works cannot guarantee detection, but prevent the bugs from malicious attacks. DangNull [28], FreeSentry [46], DangSan [43] and pSweeper [29]

track pointers to all allocated objects and explicitly nullify pointers when the referenced objects are freed. MarkUs [17], and Minesweeper [24] attempt to eliminate UAF bugs using a garbage collector, quarantining freed objects until all references to them vanish from the applications heap, stack, or registers. However, they encounter significant CPU and memory overhead due to traversing dangling pointers.

Use-after-free detection. Compared with the prevention, detecting UAF bugs is more challenging due to the complexity of the mechanisms. Approaches like CETS [32] and Undangle [21] utilize dynamic runtime for detecting UAF bugs. However, they have to inject guards on every memory access or employ heavy runtime taint analysis, resulting in significant performance and memory overhead.

In contrast, tools like Valgrind [16] and AddressSanitizer [38] focus on detecting use-after-free bugs for debugging purposes rather than security. Unfortunately, their mechanisms can be easily detoured by attackers using simple exploitable programs that trigger use-after-free bugs [28].

7 Conclusion

This paper presents a practical OTA designed by separating virtual and physical address management and by co-designing one-time-allocator and virtual address management at the user-level. BUDAlloc showcases well-rounded results in terms of performance, memory use, scalability, and bug detectability compared to FFmalloc, DangZero, and MarkUs.

Acknowledgements

We greatly appreciate the anonymous reviewers for their valuable insights and suggestions. We thank Jiyong Park for his helpful discussions and feedback. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT2201-06. This work was also supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-00871, Development of DRAM-Processing-In-Memory Chip for DNN Computing) and Googler-initiated Grant (GiG) founded by Google.

References

[1] Introduce cap_bpf, 2020. https://lore.kernel.org/bpf/20200513230355.7858-1-alexei.starovoitov@gmail.com/.

[2] bpf_loop, 2021. https://lore.kernel.org/bpf/87tuft7ff7.fsf@toke.dk/T/.

[3] Kvm performance improvements and optimizations, 2022. https://www.linux-kvm.org/images/5/59/Kvm-forum-2011-performance-improvements-optimizations-D.pdf.

[4] Zone-lock and mmap_sem scalability, 2022. https://lwn.net/Articles/753269/.

[5] BPF and XDP Reference Guide, 2023. https://docs.cilium.io/en/stable/bpf/.

[6] ebpf documentation, 2023. https://www.ebpf.io/.

[7] american fuzzy lop, 2024. https://github.com/google/AFL.

[8] Apache HTTP server project, 2024. https://httpd.apache.org/.

[9] GWP-ASan: Sampling heap memory error detection in-the-wild - The Chromium Projects, 2024. https://sites.google.com/a/chromium.org/dev/Home/chromium-security/articles/gwp-asan.

[10] mimalloc GitHub page, 2024. https://github.com/microsoft/mimalloc.

[11] NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy, 2024. https://www.nginx.com/.

[12] NIST Juliet C/C++ version 1.3, 2024. https://samate.nist.gov/SARD/test-suites/112.

[13] OpenBSD version 7.4, 2024. https://www.openbsd.org/.

[14] Redis persistence, 2024. https://redis.io/docs/management/persistence/.

[15] Uafbench, 2024. https://github.com/strongcourage/uafbench.

[16] Valgrind, 2024. https://valgrind.org/.

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

[18] Periklis Akritidis, Niometrics, Singapore, and University of Cambridge. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium (USENIX Security)*, 2010.

[19] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[20] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[21] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[22] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1977.

[23] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *International Conference on Dependable Systems and Networks (DSN)*, 2006.

[24] Márton Erdős, Sam Ainsworth, and Timothy M. Jones. MineSweeper: a "clean sweep" for drop-in use-after-free prevention. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

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

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[27] Dmitry Kuznetsov and Adam Morrison. Privbox: Faster System Calls Through Sandboxed Privileged Execution. In *USENIX Annual Technical Conference (ATC)*, 2022.

[28] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[29] Daiping Liu, Mingwei Zhang, and Haining Wang. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *ACM SIGSAC Conference on Computer and Communications Security (CSS)*, 2018.

[30] Toshiyuki Maeda and Akinori Yonezawa. Kernel Mode Linux: Toward an Operating System Protected by a Type Theory. In Vijay A. Saraswat, editor, *Advances in Computing Science – ASIAN. Progamming Languages and Distributed Computation*, 2003.

[31] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. *ACM SIGARCH Computer Architecture News*, 40(3), 2012.

[32] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[33] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *2010 international symposium on Memory management (ISMM)*, 2010.

[34] Gene Novark and Emery D. Berger. DieHarder: securing the heap. In *17th ACM conference on Computer and communications security (CCS)*, 2010.

[35] Pu Pang, Gang Deng, Kaihao Bai, Quan Chen, Shixuan Sun, Bo Liu, Yu Xu, Hongbo Yao, Zhengheng Wang, Xiyu Wang, Zheng Liu, Zhuo Song, Yong Yang, Tao Ma, and Minyi Guo. Async-fork: Mitigating query latency spikes incurred by the fork-based snapshot mechanism from the os level. In *International Conference on Very Large Data Bases (VLDB)*, 2023.

[36] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. MDev-NVMe: A NVMe storage virtualization solution with mediated Pass-Through. In *USENIX Annual Technical Conference (ATC)*, 2018.

[37] Rusty Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *Operating Systems Review*, 42(5), 2008.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (ATC)*, 2012.

[39] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *2019 Network and Distributed System Security Symposium (NDSS)*, 2019.

[40] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[41] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

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

[43] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Twelfth European Conference on Computer Systems (CCS)*, 2017.

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

[45] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. PUMM: Preventing Use-After-Free using execution unit partitioning. In *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[46] Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[47] Insu Yun, Woosun Song, Seunggi Min, and Taesoo Kim. HardsHeap: A Universal and Extensible Framework for Evaluating Secure Allocators. In *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[48] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Sixteenth European Conference on Computer Systems (EuroSys)*, 2021.