DangZero: Efficient Use-After-Free Detection via Direct Page Table Access

Floris Gorter
f.c.gorter@vu.nl
Vrije Universiteit Amsterdam

Koen Koning
koen.koning@vu.nl
Vrije Universiteit Amsterdam

Herbert Bos
herbertb@cs.vu.nl
Vrije Universiteit Amsterdam

Cristiano Giuffrida
giuffrida@cs.vu.nl
Vrije Universiteit Amsterdam

ABSTRACT

Use-after-free vulnerabilities remain difficult to detect and mitigate, making them a popular source of exploitation. Existing solutions incur impractical performance/memory overhead, require specialized hardware, and/or guarantee only protection, but not detection.

In this paper, we propose DangZero, a new solution to detect use-after-free vulnerabilities as they occur. DangZero builds on a traditional page protection and aliasing scheme, where objects are made inaccessible after a `free`, and subsequent accesses are immediately detected. In contrast to prior solutions using alias-based detection, DangZero relies on *direct page table access* in *ring 0* to provide a much more efficient implementation. The key idea is that, by giving the program's allocator direct access to the page tables, we can efficiently manage and invalidate vulnerable objects. To safely implement this, we build upon a unikernel-like design, where virtualization provides ring-0 (guest-mode) access, isolation, as well as compatibility with existing Linux programs. Moreover, we show direct page table access serves as an efficient building block for garbage collection-style *alias reclaiming*. Doing so provides the ability to safely reuse freed areas and address the scalability issues plaguing state-of-the-art alias-based solutions. Our experimental results confirm that DangZero provides accurate detection guarantees with significantly lower overhead than competing state-of-the-art solutions (e.g., 18% saturated throughput degradation on long-running programs such as the Nginx web server).

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Memory safety, Use-after-free detection, Page permissions

ACM Reference Format:
Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2022. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560625

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560625

1 INTRODUCTION

Temporal memory errors remain an important concern in the protection of computer systems against bugs and exploits. Use-after-free (UAF) bugs were ranked #7 in the CWE top 25 of the most common and impactful issues in software [40]. Additionally, Microsoft reports that UAF bugs are the second most common root cause of vulnerabilities and continue to be a preferred target for exploitation [39]. Approaches to defend against such threats can be classified as offering immediate detection or (merely) protection against exploitation. Providing detection of bugs is important in both offline (e.g., testing) and online (e.g., sampling [51]) deployment scenarios, as well as for bug triaging. Unfortunately, existing solutions in either category are problematic.

Guaranteeing UAF *protection* is typically more efficient than immediate *detection* and existing protection systems attempt to minimize their performance impact by means of a variety of techniques: type-safe memory reuse [5, 52] (which, however, can only preserve type safety), reference counting [50] (which, however, is not applicable to arbitrary C/C++ programs), one-time allocation [54] (which, however, cannot bound memory usage), and garbage collection-style (GC) solutions [4, 19, 23, 34]. While GC-style solutions have been gaining momentum for their reported efficiency, recent studies evidence nontrivial, fundamental costs with GC-style techniques—often hiding behind concurrency and generous provisioning of memory/computational power [14]. Further drawbacks are that many solutions cannot protect against exploits that do not rely on memory reuse [5], while most of the compiler-based solutions (with exceptions [4, 5, 19, 54]) cannot handle unmodified binaries. Most importantly, none of the solutions in this category can provide strong UAF detection guarantees.

Most UAF detection-focused systems rely on compiler instrumentation to track and invalidate pointers to freed objects [30, 49, 53, 55]. Despite dedicated optimizations [53], such solutions still incur nontrivial performance overhead. Less costly solutions rely on special hardware support [22, 57] (limiting deployability) or on object IDs [9, 13, 15, 22, 25, 41] (or poison values [47]) to detect UAFs (only) until a predetermined number of memory reuse events occurs (limiting security guarantees). Here also, most compiler-based solutions cannot handle unmodified binaries.

Nonetheless, binary-compatible UAF detection systems are described in literature [17, 18]. Such solutions create a new virtual page (*alias*) for each memory allocation and map it to the same physical page as the original object. As a result, every object receives a unique (unused) pointer, and the object (and its pointers) can easily be invalidated upon `free` by revoking the page mapping. Unfortunately, such alias-based solutions rely on the kernel for page protection and aliasing, and incur high overhead due to the extra syscalls and kernel administration costs. Moreover, state-of-the-art solutions [17] still suffer from impractical scalability issues due to virtual memory address space exhaustion—as we will show, this occurs in a matter of days on a heavily loaded web server.

In this paper, we introduce DangZero, an efficient, scalable, and binary-compatible UAF detection system. The key idea is to rely on direct page table access in *ring 0* (i.e., the highest privilege level normally only running OS kernels) to implement a traditional alias-based scheme in a much more efficient way. Drawing inspiration from modern unikernel-like designs [29], DangZero relies on virtualization extensions and a privilege backend such as Kernel Mode Linux (KML) [36] to provide direct access to the page tables. This strategy allows us to transparently run (and isolate) arbitrary user-space programs in ring 0 guest mode, while safely providing them with direct access to their own (guest) page tables.

We show that directly accessing page tables can crucially make alias-based UAF detection systems practical in two ways. First, by granting the program's memory allocator page table access, it can efficiently manage aliases by directly updating page table mappings. Doing so eliminates the need for operating system involvement and the corresponding (syscall and kernel administration) overheads.

Second, page tables already track important metadata about the virtual memory address space of the program and can also accommodate extra application-specific metadata. We use this observation to design an efficient *alias reclaiming* system and address the virtual memory address space exhaustion issues of prior alias-based solutions [17]. The goal is to allow *safe* reuse of virtual addresses, once we confirm that dangling pointers to the object (alias) no longer exist. Our design is similar, in spirit, to that of prior GC-style solutions [4, 19], but with two crucial differences. First, DangZero's metadata management is uniquely efficient, since it can piggyback and expand on the metadata already present in the page tables (e.g., the present bit pinpointing the resident pages to scan for dangling pointers). Moreover, since DangZero reclaims virtual aliases rather than objects in physical memory, our reclaiming strategy is not prone to the typical performance/memory tradeoff of GC-style techniques [14]. Indeed, as we shall see, our alias reclaiming strategy is very efficient, allowing DangZero (a *detection* system) to outperform even state-of-the-art GC-style *protection* systems [4, 19] on long-running benchmarks (which commonly feature frequent, short-lived allocations), without having to resort to memory over-provisioning or concurrent reclaiming on spare CPU cores.

We have evaluated DangZero on standard benchmarks (SPEC CPU 2006 and 2017) and long-running application benchmarks (the Nginx web server in particular). On SPEC CPU 2006, DangZero reported a geomean performance overhead of only 16% (and 22% on SPEC CPU 2017) compared to 40% for the state-the-art alias-based UAF detection system [17]. On Nginx, DangZero reported saturated overheads as low as 11-18%, significantly lower than state-of-the-art UAF protection/detection systems, with consistently modest (and bounded) memory overhead.

To summarize, we make the following contributions:

- A new approach to detect use-after-free bugs based on alias allocation with virtualization-based direct page table access.
- A novel solution for alias reclaiming.
- A prototype of DangZero using KML as a privilege backend.

```
int* ptr = malloc(sizeof(int));
free(ptr);
int b = *ptr;
```

Listing 1: Use-after-free example

- An evaluation to show that DangZero significantly outperforms prior detection systems and even state-of-the-art GC-style protection systems on long-running benchmarks.
- Code available at: https://github.com/vusec/dangzero

2 BACKGROUND

2.1 Use-after-free

Use-after-free (UAF) bugs are temporal memory errors present in unsafe languages such as C and C++, which arise due to heap allocated objects being dereferenced after already being freed. These bugs are possible since (so-called *dangling*) pointers to freed objects remain intact even if the pointed memory location is no longer valid. Attackers typically exploit UAF bugs and the corresponding dangling pointers by forcing memory reuse after the free, but before the use. However, depending on the allocator design, exploitation without memory reuse (with allocator metadata playing the role of the target object) is possible [5]. Listing 1 shows a trivial example of a UAF bug. The temporal nature of these bugs makes them hard to detect, both visually in the code as well as through program analysis, and many mitigation designs aimed to neutralize UAF bugs suffer from significant (runtime/memory) overhead. In this paper, we show such cost is not fundamental and direct page table access can unlock an efficient and scalable alias-based solution.

2.2 Page tables

Page tables are a software-maintained data structure that is used by the memory management unit (MMU) of the CPU to describe how to map *virtual* to *physical* memory. On most common architectures, page tables are stored as a hierarchical tree, where certain bits of the virtual address are used to select the entry in the respective level of the page table. A page table entry (*PTE*) stores the address to the next level of the tree, or (for the last level) the result of the address translation. Additionally, PTEs store a limited number of metadata bits, such as permissions of that mapping and whether the entry is valid ("present"). Finally, each PTE contains a number of bits that are ignored by hardware, and thus can be used by the operating system for additional information. Most 64-bit architectures use 4-level page tables, each table consisting of 512 entries, yielding a 48-bit (256 TB) virtual address space. Some modern CPUs also feature 5-level page tables, but for the remainder of this paper we assume a 4-level page table structure for simplicity. Many different names exist for referring to the different levels of these structures; for this paper we simply refer to them as L4 through L1 (with L4 the root/first table, and L1 the leaves/last level).

Typically, each process has its own set of page tables, describing the address space of that process. Linux splits the available address space in half, giving the bottom half to user space and keeping the top half for its own data. This means each user process has 128 TB of virtual addresses available. To request new mappings, or change existing mappings, the process (and its memory allocator) issues system calls such as `brk`, `mmap`, and `mremap`. On top of the page tables, Linux also maintains its own data structures, containing information for each consecutive virtual memory area (*VMA*).

When running a virtual machine (VM) using hardware virtualization extensions, there are two levels of page tables: the guest page tables, and the extended page tables (*EPT)* on the host. The former behave exactly as described above, and give the guest the illusion of running directly on the hardware. The EPT is managed by the hypervisor and is similar to normal page tables, except it translates every guest-physical address to a host-physical address.

2.3 Access to privileged CPU features

To achieve direct page table access, DangZero requires access to privileged features normally reserved for ring 0. The Dune [6] project presented a practical implementation through the use of a lightweight virtual environment. In particular, the application runs in ring 0 (guest mode) of a specialized "virtual process" environment. This provides the application access to all privileged features (e.g., guest page tables), while still being isolated from the rest of the (host) system by the hypervisor.

Dune used a small library operating system (libOS) running in the guest alongside the application, to manage basic kernel tasks so that unmodified Linux binaries could run. Additionally, a specialized (KVM-based) hypervisor mapped system calls issued by the guest via *VM exits* to Linux syscalls on the host.

Of similar spirit is the Kernel Mode Linux (KML) [36] project, which allows programs to run in ring 0 alongside the Linux kernel. KML has the advantage of not requiring expensive VM exits for every system call a la Dune. Similar to Dune, KML still requires a virtual environment for isolation, that is to protect the rest of the system. The resulting design effectively transforms Linux into a libOS and the process into a unikernel—and recent application optimization work has shown KML can be efficiently used as such [29].

3 THREAT MODEL

We assume a standard threat model, with an attacker seeking to exploit arbitrary use-after-free vulnerabilities in a victim binary program (written in an unsafe language), for the purpose of information disclosure, privilege escalation, etc. We consider arbitrary use-after-free exploits regardless of whether memory reuse and other exploitation techniques (e.g., memory massaging) are involved. We assume the program is free from other vulnerabilities (e.g., buffer overflows) or otherwise hardened against them with orthogonal mitigations.

4 DANGZERO

DangZero protects against UAF bugs immediately as they occur, by creating *distinct* virtual addresses for each heap-allocated memory object. Whenever an object is freed, its memory is rendered inaccessible by invalidating the virtual memory mapping. This is done by means of the page protection flags and renders further (UAF) accesses invalid. To reduce physical memory overhead, we allow objects to still share *physical* pages. Since the smallest granularity of mappings is at the page level, we thus create new and unique *aliases* for every allocated object, which can be invalidated on `free`.

Figure 1 provides an overview of the main components of DangZero and their interactions. At the core of DangZero lies an overlay

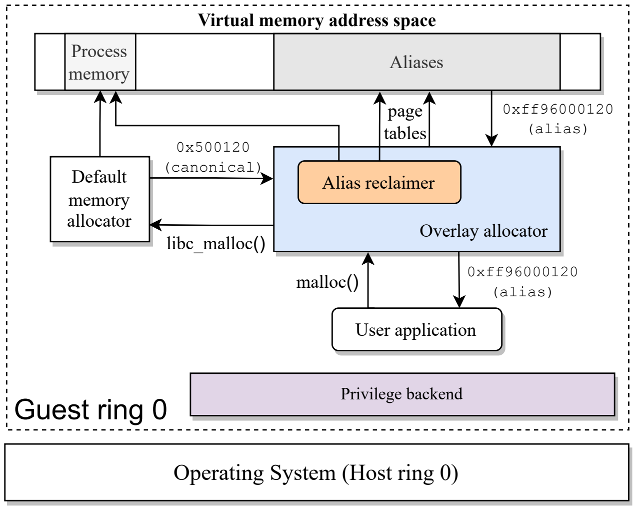

Figure 1: Overview of DangZero's components.

allocator, which serves as an extension to the default memory allocator (e.g., `glibc`) and has direct access to the page tables of the running process. Heap allocation requests are intercepted such that alias pages can be created and returned to the user. Heap deallocation requests, in turn, are intercepted to invalidate the alias mappings. The privilege backend provides access to restricted features, direct page table access in particular. DangZero does not require instrumentation of the program, nor of the underlying allocator and thus works on off-the-shelf binaries.

Overlay allocator. DangZero's overlay memory allocator can be viewed as a proxy between a user-space program and the default memory allocator. Upon request of a memory allocation (e.g., `malloc`), DangZero first relays this request to the default memory allocator. The default allocator may request virtual (and physical) memory from the operating system (typically via `brk` or `mmap`) and ultimately returns the virtual address of the allocated heap slot (which we refer to as the *canonical* address of the object) to our overlay allocator. Then, our allocator creates one or more alias pages in an unused area of the virtual memory address space, computing what we refer to as the *alias* address of the object. Since our allocator has direct access to the process page tables, it can directly and efficiently create these alias mappings. Finally, we transparently return the corresponding alias address to the user program, which uses this address to refer to the allocated object as usual, unaware this concerns an alias.

When the program frees the object, we perform similar instrumentation. Upon interception of a `free` (or equivalent) call, we first invalidate the alias mapping directly in the page tables. We then compute and pass the canonical address of the object to the original `free` so its physical memory can be reused by the default allocator.

Alias reclaiming. Providing every object with its own unique alias address would eventually lead to virtual memory exhaustion. Even on the large virtual address spaces of modern systems, exhaustion can occur within days of operation on heavily loaded server programs. To address this challenge, DangZero relies on an

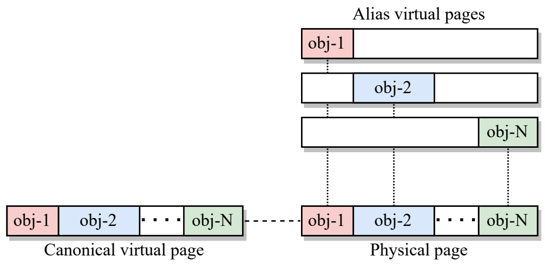

Figure 2: Page aliasing: one virtual alias page per object.

alias reclaimer to enable *safe* reuse of alias addresses. The reclaimer periodically marks invalidated (i.e., previously freed) areas of the alias space as reusable for future alias creation, if it determines (dangling) pointers to it no longer exist. This component is similar, in spirit, to a (conservative) garbage collector, but only reclaims virtual (rather than physical) memory and is tightly integrated with the overlay allocator as well as the page table data structure.

In the remainder of this section, we describe the operation and components of DangZero in greater detail.

4.1 Temporal safety through page protection

At the core of DangZero lies the idea of using page protection as a mechanism to prevent and detect use-after-free errors. By removing access from the application to deallocated memory, the memory management unit (MMU) in hardware will automatically raise an exception when invalid accesses are detected. The MMU, however, operates on a page (typically, 4096 bytes) granularity. Placing every object on its own (physically backed) page would lead to significant memory overhead caused by fragmentation.

Therefore, DangZero builds on the idea of *aliasing* [18], where each object has its own *virtual alias*, but multiple objects share the same underlying physical page. Figure 2 visualizes how the different pages relate to each other. We refer to the virtual page returned by the default memory allocator as the *canonical* page. These pages can contain multiple memory objects by design (up to 128 objects on a 4K page). The physical page backing the canonical page matches the objects. Then, for each individual object, there is a distinct alias page, with the in-page offset matching the canonical page. If the original allocation spans multiple pages, the resulting object will have the same number of corresponding alias pages.

Upon deallocation of an alias, we require the corresponding canonical address, so that we can initiate the deallocation process of the default allocator and free up the physical memory. For this, we pad every memory allocation with the canonical address pointer. In practice, this means that we change every `malloc(n)` call to `malloc(n + sizeof(void*))`.

4.2 Creating and invalidating aliases

Virtual address mappings are maintained in the process page tables, which are managed by the operating system kernel. Normally, for a process to update mappings, it thus needs to go through system calls, which is an expensive operation. Operating systems such as Linux are also not built for creating (large amounts of) custom mappings, and thus are restrictive in what can be set up. Moreover, kernel memory usage often explodes because of the metadata kept for each alias mapping, such as the VMA data structures.

With DangZero, we can directly modify the page tables from our overlay allocator instead, completely outside of the control of the operating system. DangZero takes over an unused area of the virtual address space, that is never touched by the kernel, and directly writes into the page table page entries corresponding to that area. To facilitate this access to (normally) restricted resources, our *privilege backend* provides safe access. The design of DangZero is agnostic to the exact mechanism used, but in general it must support direct read and write access to page tables and allow for TLB flushing, while at the same time ensuring safety and isolation to the rest of the system. Our primary backend uses a guest running Kernel Mode Linux (KML) for transforming Linux into a libOS; Section 5.1 provides more details on these underlying mechanisms.

In order to create an alias page upon request of a memory allocation, we have to consider the effects of demand paging. Since Linux applies demand paging on its memory system, heap allocations do not have physical backing until they are actually used. However, we cannot create alias mappings without knowing the physical address of the allocation. Therefore, the allocator touches the canonical page in order to force physical backing for the memory object, after which we can set up the page tables to ensure the alias page points to the same physical page. We do not observe overhead penalties from the forced population of the pages in practice, since the pages are normally used by the default allocator or shortly after the allocation anyway. If this approach were to introduce difficulties for certain workloads, an alternative design would use a custom page fault handler, such that alias pages are created as soon as the heap allocations receive physical backing.

4.3 Alias reclaimer

To support long-running applications that may exhaust the alias space, DangZero introduces the alias reclaimer component. The main goal of this component is to allow *safe* reuse of previously freed alias pages. We deem an alias page safe for reuse if no references (i.e., pointers) exist to the *object* associated to the alias page. Conversely, if there once existed a large object of 10 pages, that is now freed, and a (dangling) pointer exists anywhere in the program to *any* of these 10 pages, *none* are considered safe for reuse.

The main design goal of the reclaimer, besides safe reuse, is to be lightweight. The reclaimer runs infrequently, and as such, its normal operation (when a cleanup is not needed) should be minimal in terms of performance and memory overhead. Our design is therefore inspired by that of conservative garbage collectors (GCs) [10], but is tailor-made for the environment of DangZero. Unlike a traditional GC, the reclaimer is *not* critical for the physical memory overhead of the system. The alias reclaimer automatically runs when a predefined *watermark* on the number of invalidated pages is reached. Since we can never reclaim memory that is still in use (i.e., not freed), the watermark excludes any such pages.

Similar to a conservative GC, the reclaimer consists of a *marking* and a *sweeping* phase. During marking, all memory and registers of the program are scanned for possible pointers to the alias space. For any pointer that is found pointing to a freed alias object, we *mark*

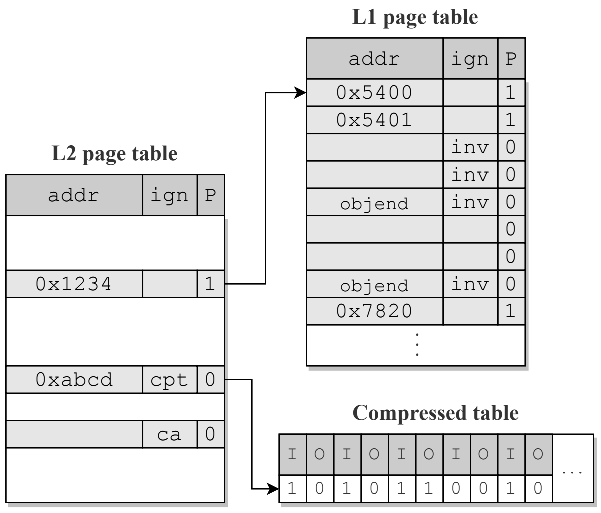

Figure 3: Alias reclaimer metadata in the page tables.

that object as still referenced. Afterwards, during the *sweep*, we iterate over all freed objects in the alias space, and return all objects that are *not marked* to the alias freelist of our overlay allocator. The reclaimer ignores all the nonfreed objects, and objects are zeroed on free to avoid cyclic dependencies when marking [19]. Zeroing objects also provides protection against leaking data from uninitialized reads, when a physical page is reused across allocations.

Currently the mark and sweep phases require temporarily *stopping the world*, which means the target application is essentially paused. This is a common paradigm in garbage collection to obtain a consistent view of the program's memory [10]. There are further possible optimizations, such as concurrent and parallel marking and sweeping. In particular, direct page table access enables optimizations such as fast dirty bit scanning [6]. However, we opted for the much simpler stop-the-world approach, as the overhead of reclaiming was low enough that we did not feel more complex concurrent reclaiming was a feature to prioritize.

The reclaimer must keep track of certain metadata in order to perform its mark and sweep operations correctly. First, it needs knowledge on the state of each alias page: whether it is *available* (available to create new aliases), *in-use* (in use by a live object), or *invalidated* (part of a dead object that was explicitly deallocated with `free()`). Additionally, the reclaimer needs to know where object boundaries were for invalidated objects, since a pointer to anywhere in the object excludes reuse of any of its pages. Finally, during the marking phase, the reclaimer needs to remember which objects were marked for the sweep phase.

The reclaimer stores all of this information in the page tables themselves, resulting in virtually no memory overhead, as shown in Figure 3. We obtain this storage for free, because invalidated alias pages (i.e., after a call to `free()`) are set to nonpresent in their respective page table entries (PTEs), meaning all other bits in the PTE are unused and ignored by the hardware. Therefore, we use one of the PTE bits to distinguish between available and invalidated entries. Additionally, we use another bit to mark the *end* of each object. Thereby, object boundaries can be reconstructed by looking for consecutive PTEs with this bit not set, followed by one where it is. Figure 3 shows on the right an L1 page table with several PTEs, where the present bit is indicated by the P column, the invalidated bit is stored in one of the ignored bits, and so is the bit indicating the end of objects. The figure depicts a mix of in-use objects (e.g., the first two entries), invalidated objects (entries 3-5 and entry 8), and available entries (entries 6-7).

Page table reclaiming and compression. If any of the 512 entries of a page table is still a valid mapping (i.e., the present bit is set), the whole page table needs to be kept around, and thus it provides free storage space for our metadata bits. However, the moment all entries are nonpresent (i.e., available or invalidated), keeping it around only for the metadata bits wastes a lot of space. A 4K page table has 512 64-bit entries (each describing a page), but our metadata only requires 2 bits per entry. As a result, we *compress* page tables into 128 bytes of data the moment all entries are nonpresent and free the original page table. We can place 32 of these compressed page tables into a single 4K page and manage these chunks of memory with a slab allocator [11]. We then point the L2 page table to this compressed entry, and set a special bit in the page table entry to indicate its presence (`cpt` in Figure 3).

On top of this, we identify two common states a page table can be in, which allow for further compression. Often, a (compressed) page table is completely filled with invalidated objects. Additionally, a page table often only contains entries where every page is a separate object or every page belonged to the same large object (i.e., all or none of the object-end bits are set, respectively). In all of these cases, the entire compressed page table (128 bytes) is compressed into a single bit that is stored in the higher-level page table (`ca` in the last L2 entry in Figure 3). As we will show in Section 6, compression drastically reduces the memory overhead of DangZero.

5 IMPLEMENTATION

We implement DangZero as a shared library that overlays the default memory allocator via `LD_PRELOAD`. Additionally, DangZero requires a backend to be available for direct page table access, which we describe in detail in the following section.

5.1 Privilege backend

One of the critical components of DangZero is its privilege backend, which is responsible for providing direct access to privileged kernel features from the (user-space) allocator. The ability to directly modify page tables (and issue corresponding TLB flushes) is essential for the performance of our system.

For this, DangZero's primary backend builds on top of Kernel Mode Linux (KML) [36], a Linux kernel modification that enables running user-space applications in kernel mode (ring 0). This effectively transforms Linux into an efficient unikernel [29]. It also provides the application with all privileges on the system, allowing it to call into kernel code (e.g., `alloc_page` for allocating physical pages) and write to any memory. However, KML still remains Linux under the hood, thus benefiting from all existing Linux drivers and supporting all existing Linux binaries.

However, providing arbitrary user-space programs with kernel privileges is inherently unsafe: a bug or vulnerability in the program can affect or corrupt any other program and user on the system.

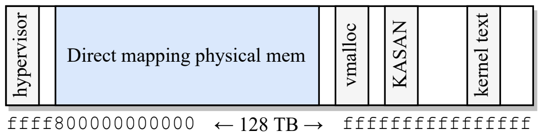

Figure 4: Kernel-space virtual memory layout.

Like other unikernel systems, we therefore run our system inside a (lightweight) virtual machine (VM), which isolates the guest from the rest of the system. By dedicating the guest system to an individual process, we consider a single security domain, and hence an attacker abusing orthogonal vulnerabilities cannot exploit page table access to compromise other security domains (e.g., other applications or the host). Additionally, this setup can provide higher performance than a bare-metal baseline, since the KML kernel can be tweaked specifically for running a single application on virtualized hardware [29]. DangZero does not include such optimizations, to fairly evaluate the overhead introduced by its design.

For DangZero, we used the latest Linux kernel with an available KML patch, i.e., v4.0. We applied two patches to the KML kernel such that it can properly operate with the latest LTS version of Ubuntu at the time (20.04). The KML project introduced a patch to `glibc` 2.11 to change system calls into direct calls, which we port to `glibc` 2.31 (the default for the used OS). The modern version of `glibc` allows us to use the default `gcc` version on our OS (9.4.0). For some features, we require access to kernel data structures (e.g., iterating through VMA structs). This is conveniently implemented in a kernel module, which, thanks to KML, we can call directly from our allocator via a regular function call.

5.1.1 *Alternative backend: Dune.*

Next to KML, we have also implemented an alternative backend for DangZero to demonstrate the wide applicability of its design. For this purpose, we chose Dune [6], a hypervisor and libOS that is aimed at granting user applications access to privileged hardware features. Our Dune-based DangZero prototype showed almost identical performance as KML for the SPEC benchmarks. However, on system call intensive applications (e.g., Nginx), the Dune baseline itself reports a significant performance overhead (up to 60%), because Dune has to translate system calls into far more expensive VM exits, resulting in higher overall overhead for DangZero/Dune. As such, the remainder of this paper is solely evaluated using DangZero's KML backend.

5.2 Alias page tables

While KML grants our allocator direct access to kernel memory such as the page table data structures, it cannot easily take full control of them. In particular, Linux itself is unaware of DangZero, and will overwrite any changes our allocator would make in user-space mappings. Instead, DangZero operates in a normally reserved area of the Linux kernel address space (untouched by the kernel by design). This strategy has two advantages: it does not require kernel modifications to allow for page table access sharing and it does not reduce the amount of virtual memory available to userland.

Figure 4 shows a (simplified) kernel-space virtual memory layout. The kernel reserves a virtual memory area of 64 TB for the direct mapping of all physical memory. Since most machines have at most a few hundred gigabytes of RAM, most of this area is not in use. Therefore, we can reserve a large part of this unused area to host our alias page tables. The 64 TB of direct mapping corresponds to the highest level (L4) page table entries 273 to 400. Assuming that the system uses up to 27 x 512 GB physical memory at most (a generous overapproximation for the foreseeable future), this leaves entries 300 to 400 available for our virtual alias pages. This area can host up to 50 TB of virtual alias space, which corresponds to a maximum of 12.5 billion concurrent 4K alias pages.

5.3 Supporting fork

The downside of bypassing the kernel when creating alias mappings directly in the page tables of a process is that the kernel is not aware of these pages when executing operations such as `fork`. The `fork` system call creates a child process where the pages of the parent and the child are shared. Upon modification of a shared page, the copy-on-write (CoW) technique ensures a copy of the page is used. Unfortunately, the alias pages are not copied over to the page tables of the child process, and even if they were (e.g., by being present in the normal kernel data structures such as VMAs), they would wrongfully remain aliased to the physical pages of the *parent*.

The intuition to solve this problem is that we forcefully trigger CoW on all of the canonical pages in the child process, causing the pages to receive new physical backing. After this, we can create the appropriate alias pages and make them point to these new physical pages in the child. However, recreating the alias mappings in the child to point to the new physical pages is difficult, since we lack both a canonical-to-shadow map, or a parent-physical to child-physical map. Maintaining such a data structure would be prohibitively expensive for an uncommon operation such as `fork`.

To overcome this problem, we apply the following algorithm as an epilogue to the `fork` library call (also shown in Listing 2). We create a temporary map of parent-physical to canonical addresses, which we generate by walking the page tables for each canonical address in the parent and reading the physical page in the resulting PTE. Using this map of physical-canonical address pairs, we reconstruct the alias page tables in the child process. For each alias page in the parent, we obtain its physical address by doing a page walk and look up the corresponding canonical address in our map, using the physical address as key. Then, we touch the virtual address in order to trigger CoW, resulting in new physical backing in the child. Next, we look up the new child-physical address with a page walk of the canonical address in the child. Finally, we create the corresponding page table entries in the alias space of the child, such that the new alias page points to the new physical backing. Throughout this algorithm, we sync the parent to wait for the child to finish these operations, such that the memory state remains consistent.

5.4 Optimizing page entry lookups

Direct access to the page tables allows us to optimize our design specifically for managing the alias pages. Since we can often grow the alias space in a linear fashion, we can make assumptions surrounding the details of the page walks. More specifically, we can cache the pointers to intermediate page table levels, to avoid repeating these lookups unnecessarily. This optimization benefits from the fact that, in a linear design, the higher-level page table

```
// construct physical -> canonical map (pre-CoW)
for canon_addr in heap:
    phys_addr = page_walk(canon_addr)
    map[phys_addr] = canon_addr

// reconstruct alias mappings in child
for alias_addr in alias_space:
    phys_addr = page_walk(alias_addr)
    canon_addr = map[phys_addr]
    *canon_addr; // trigger CoW to force new backing
    child_phys_addr = page_walk(canon_addr)
    pt_map(alias_addr, child_phys_addr)
```

Listing 2: Recreating alias mappings in the child after `fork`.

(L4 and L3) entries change infrequently. Even the lowest page table level lookup (L2 to L1) only changes once every 512 alias pages. In practice, every time we create a new alias we increment the L1 page table index. If this index reaches 512, we wrap around, perform a new lookup for L2 to L1, and increment the previous levels in a cascading manner when necessary. This optimization significantly reduces the number of lookups and exchanges them with much cheaper increment operations.

6 EVALUATION

We evaluated the security, performance, and memory characteristics of DangZero using different benchmarks. For our security evaluation, we used the Juliet Test Suite and confirm detection of known CVEs. For our performance/memory overhead evaluation, we used the SPEC CPU2006 and CPU2017 benchmarks (for their mix of CPU and memory intensive real-world programs) as well as the Nginx and Apache web servers (representative of long-running, system-call and allocator-heavy programs). All reported numbers are the median of 5 runs to reduce noise (unless otherwise stated). For the SPEC benchmarking suites, we averaged the results using the geometric mean (geomean) over all the benchmarks.

6.1 Security evaluation

We empirically confirmed that DangZero can accurately detect and mitigate use-after-free bugs by running the NIST Juliet Test Suite v1.3 [26]. This test suite contains hundreds of test cases, categorized by vulnerability type (CWE), and tests for both false positives and false negatives. DangZero successfully detected all the use-after-free and double free bugs in the test suite (CWE416 and CWE415, respectively) with (or without) alias reclaiming and with no false positives, no unbounded memory usage, and regardless of memory reuse. Besides the Juliet test suite, DangZero also correctly identified a use-after-realloc bug contained in the test workset of 400.perlbench (SPEC CPU2006) [47].

Furthermore, we successfully confirm that DangZero is able to detect six distinct use-after-frees from CVEs and issues reported in programs such as PHP and Python [1–3, 12, 24, 32]. By means of a signal handler we verify that the memory access errors triggered by the use-after-frees originate from page table entries that were invalidated by DangZero. Finally, DangZero has a relatively small codebase of 3,100 LOC, which shows the proposed design can be implemented with a small trusted computing base.

6.2 Performance baseline

In our experiments, we used the virtualized environment without KML enabled as baseline—see later for the overhead (and speedup opportunities) of virtualization itself. Therefore, the measured performance overhead is the cost of running KML together with the slowdown incurred by the overlay allocator. Although we originally expected KML to provide a consistent speedup (since it removes the need for mode switching at the syscall interface), in practice we observed an overall marginal impact. In fact, for SPEC CPU2006, we observed a geometric mean change of 0% in runtime, with the extremes being one binary experiencing a speedup of 8%, while another binary suffers from a slowdown of 4%. We observed similar effects of KML for the other benchmarks.

Additionally, while we did use a KML-patched `glibc` to perform direct calls instead of system calls, we observed that this does not provide a speedup in practice. Although we confirmed that the `glibc` patch does reduce the number of CPU cycles required for a system call in microbenchmarks, under more complex workloads this speedup degrades, matching the observations seen in previous work [29]. With all of this in mind, we believe that comparing to an in-guest no-KML baseline accurately represents the overhead of our system. For all the competing systems, we used a bare-metal baseline with the default system allocator (except MineSweeper's [19], which uses the intended `jemalloc` [21] baseline), matching their primary deployment scenario.

6.3 SPEC CPU2006

We ran the SPEC CPU2006 benchmarks on a machine with the following configuration: Intel i7-6700 (Skylake) CPU @ 3.40GHz, inside a virtual machine using QEMU/KVM, where the VM receives 24 GB of RAM backed by hugepages, and runs Ubuntu 20.04 on the Linux v4.0-KML kernel. We provided hugepage backing to QEMU/KVM to avoid high variance due to transparent hugepages. We also disabled all optional CPU mitigations to avoid interference. The configured watermark for the alias reclaimer was never reached, since SPEC CPU2006 consists of only short-lived applications.

6.3.1 Runtime overhead. The individual overhead of each SPEC CPU2006 binary is displayed in Figure 5. Excluding alias reclaiming, DangZero reported a 14% geometric mean runtime overhead on SPEC CPU2006. Forcing a single alias reclaim operation (mark and sweep) at the end of each program puts the geomean at 16%. For the relatively short-lived SPEC applications, running without the alias reclaimer is sustainable. Nonetheless, the mere 2 percentage-point geomean increase confirms alias reclaim management is efficient.

6.3.2 Components buildup. Next, we look at the sources of our overhead by measuring each component of DangZero separately, as shown in Figure 5. We divide our design into five core elements: (1) using Kernel Mode Linux, (2) creating aliases, (3) using aliases, (4) disabling aliases, and (5) reclaiming aliases.

Enabling Kernel Mode Linux has a varying effect on the performance of the binaries. While 429.mcf experiences a speedup of 8%, the runtime of 458.sjeng slows down by 4%. However, as mentioned earlier, overall the geometric mean of enabling KML is 0%. Next, creating aliases entails that we request pages from the kernel, insert them into the alias address space, and make them

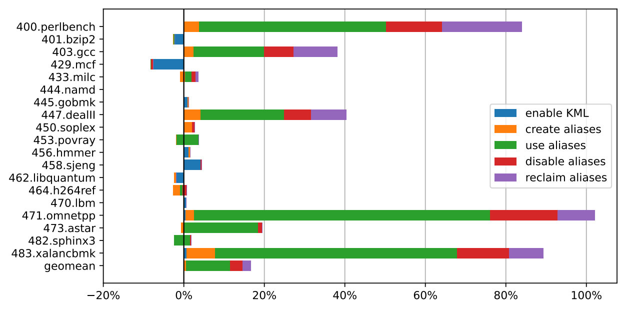

Figure 5: SPEC CPU2006 runtime overhead for DangZero.

System	Reported	Measured
DangZero-base	-	14.0%
DangZero-alias-reclaim	-	16.0%
Oscar	40.0%	40.0%
MineSweeper	5.4%	10.0%
MarkUs	10.0%	14.0%
FFmalloc	2.3%	1.2%

Table 1: SPEC CPU2006 runtime overhead compared.

point to the physical pages of the corresponding memory objects. This step includes the addition of the 8-byte padding for memory allocations, to contain the canonical pointer normally required during free. Including this ensures the right number of alias pages are created for our measurements. As seen in the figure, creating the alias pages only incurs a small overhead, of which the efficiency can be attributed to the direct access to the page tables and our page lookup caching optimization.

Actually *using* the alias pages incurs the largest overhead of all the components. Using the pages means the alias addresses are returned to the application, and the canonical addresses are stored in the padding. This component results in relatively high overhead due to the significant increase in pressure on the translation lookaside buffer (TLB). After creating the aliases, the next logical step is to disable the aliases when the memory is freed. Disabling (i.e., unmapping) the alias pages provides the actual security benefit of detecting and mitigating use-after-free bugs, and incurs some overhead, primarily due the the required TLB invalidation.

The last component concerns the overhead imposed by the need to gather metadata for the alias reclaiming feature. Most notably, this includes maintaining object boundary information, zeroing out memory upon free, and dealing with compressed page tables during page walks (i.e., uncompressing).

6.3.3 Comparison to other systems. Compared to Oscar [17], which has the same core functionality as DangZero (without alias reclaiming), DangZero reported significantly better performance. Specifically, Oscar results in a geomean runtime overhead of 40% on SPEC CPU2006, whereas our Oscar-like (i.e., no-reclaim) version is at 14%. Even with alias reclaiming and the corresponding compression enabled, our overhead (16%) is far less in comparison. Our highest runtime overheads come from: 471.omnetpp (2.03x), 483.xalancbmk (1.90x), and 400.perlbench (1.84x). In comparison, Oscar reports the highest overheads on the same binaries, with 4.5x for 471.omnetpp, 4.0x for 483.xalancbmk, and 4.3x for 400.perlbench.

Table 1 displays the geomean runtime overhead on SPEC CPU2006 for various use-after-free protection systems, along with our results. We include both the overhead reported in the respective papers, as well as the overhead we measured on our setup. Specifically for MineSweeper [19] and MarkUs [4], there is some discrepancy between the reported and measured numbers, since we pin the SPEC execution to a single core for a fair comparison—whereas the numbers reported in the original papers do not include the overhead of the garbage collection threads offloaded to other cores (which is important to factor in [14]). The measured number for MineSweeper/403.gcc is simply the originally reported value, since this binary crashes on our machine.

As shown in the table, MineSweeper and MarkUs, albeit not offering detection guarantees, result in comparable (measured) runtime overhead on SPEC CPU2006. Furthermore, the need for garbage collection sweeps in the short-lived SPEC binaries is limited and thus such GC-style solutions could experience far more stress in other settings. In fact, in our evaluation of Nginx in Section 6.5, we see that these systems impose a significant runtime overhead in order to contain physical memory usage. DangZero does not suffer from this limitation, as the increase in physical memory usage is less of a concern due to our virtual aliasing design.

FFmalloc [54] reports a very low geomean overhead of 2.3% and we measured an even lower overhead of 1.2%. However, FFmalloc trades memory consumption for runtime overhead, as we will show later. Another concern is that FFmalloc's fast forwarding allocation design implies that virtual addresses can *never* be reused. Since FFmalloc does not contain any form of garbage collection or reclaiming, the instrumentation results in scalability issues due to virtual address space exhaustion when considering long-running applications, an issue we also show later on Nginx.

6.3.4 Memory overhead. Figure 6 displays the memory overhead of DangZero for each of the SPEC CPU2006 binaries, in comparison to the overhead of Oscar (as reported in the paper [17]). These overhead numbers concern the full (most practical) configuration of DangZero, which includes alias reclaiming and compression. While Oscar reaches a maximum memory overhead of 5.2x on 447.dealII, DangZero incurs an overhead of only 1.4x on the same binary, even though the systems share the same core aliasing principle. This discrepancy is due to our page table compression feature.

Since our alias page tables are created using kernel space memory, we cannot simply report the resident set size (RSS) of the application to express the memory overhead. Instead, we calculate the memory consumption of DangZero by taking the sum of the maximum RSS and the maximum number of concurrent page tables (one 4K page per table). The number of page tables pages varies throughout the execution due to the compression feature having the ability to clean up pages. Our memory consumption is an upper bound, since the max RSS and the max number of pages do not necessarily need to align in time, although this is likely.

Table 2 displays the reported and measured memory overheads of all the considered systems. DangZero results in a geomean memory overhead of 75% without compression, which decreases to 25%

System	Reported	Measured
DangZero-no-compression	-	75.0%
DangZero-with-compression	-	25.0%
Oscar	60.0%	nontrivial
MineSweeper	11.1%	22.3%
MarkUs	16.0%	26.9%
FFmalloc	61.0%	115.8%

Table 2: SPEC CPU2006 memory overhead compared.

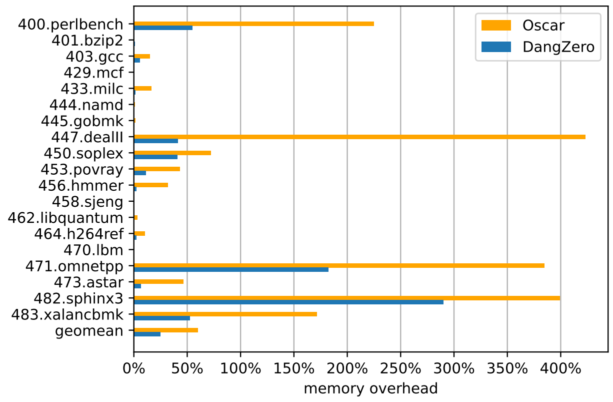

Figure 6: SPEC CPU2006 memory overhead for DangZero.

when applying compression. In comparison, Oscar reports a geomean memory overhead of 60%, which we could not easily reproduce due to the complexity of estimating the RSS when the VMAs contain alias pages. The memory overhead of FFmalloc appears much higher when measured on our machine, which aligns with the observations of the MineSweeper project [19]. The authors of MineSweeper measured a 244% geomean overhead for FFmalloc on SPEC CPU2006, which appears to be a consequence of memory fragmentation preventing FFmalloc from freeing old pages, for example if only one small object remains alive on a page.

Overall, the memory overhead of DangZero is comparable to the state-of-the-art UAF mitigations that focus on memory efficiency, and is far better than most other UAF mitigations. Combined with the competitive runtime overhead and its detection guarantees, this demonstrates the efficacy of our design.

6.4 SPECspeed2017

We evaluated DangZero on SPECspeed2017 (pure C/C++, no Fortran) with the same setup as the previous experiments on SPEC CPU2006, except taking the median of 3 runs (ensuring standard deviations are minimal). For these experiments, we kept the optional OpenMP extensions disabled. On SPECspeed2017, we observed a geomean runtime overhead of 22%, accompanied with a memory overhead of 30% (Figure 7). DangZero imposes the highest runtime overhead on the counterparts of the same benchmarks as SPEC CPU2006, with 2.56x on 623.xalancbmk and 2.08x on 620.omnetpp.

We attempted to run Oscar on SPEC CPU2017, since the numbers were not reported in the original paper. Unfortunately, this is nontrivial due to dependencies on the old `glibc` version that

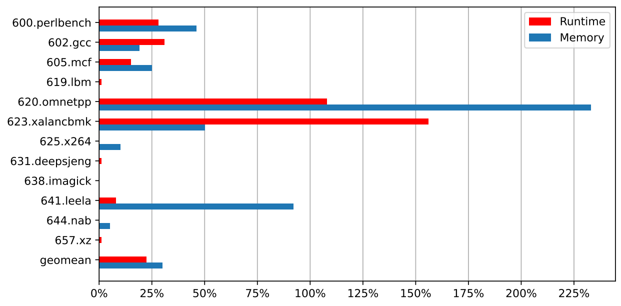

Figure 7: SPEC CPU2017 overhead for DangZero.

Oscar uses. MineSweeper reports a 10.8% geomean runtime overhead, however this is including the Fortran binaries. We tried to extract the geomean for the pure C/C++ benchmarks from the MineSweeper paper, but this proved problematic due to 620.omnetpp not being present. The partial geomean (excluding omnetpp) of MineSweeper is 12%, compared to our partial geomean of 17%. Moreover, the reported numbers for MineSweeper do not incorporate overhead offloaded to other cores, although SPEC CPU2017 contains some parallel programs. We measured a geomean of 14.7% for MineSweeper, where we pin SPEC CPU2017 (without OpenMP) to one core, and use the reported values for 602.gcc and 644.nab, because both binaries segfault on our system.

Similarly, MarkUs reports a geomean of 13% runtime overhead. Again, this includes Fortran binaries, and offloads some overhead to other cores. We extract the pure C/C++ geomean overhead from their paper, which is 15.5%. We then measure the overhead of MarkUs pinned to one core on our setup, which results in a geomean overhead of 15.9%. We used the reported value for 602.gcc, since it segfaults on our system. Overall, the runtime overhead of DangZero is again comparable to MineSweeper and MarkUs, while DangZero provides use-after-free detection guarantees.

For FFmalloc we could not produce conclusive results on our machine. From the results of SPEC CPU2006 in the FFmalloc paper, we speculate that 602.gcc is the one binary in SPEC CPU2017 that mostly impacts the geomean runtime. Unfortunately, this specific binary crashes on our setup, where FFmalloc reports it cannot allocate memory after a couple of minutes—even on systems with 64 GB of RAM. Without including 602.gcc, the partial geomean runtime overhead is negligible—around 0%. However, the partial geomean for the memory overhead is 103%.

Only MineSweeper reports their memory overhead on SPEC CPU2017, and they also include their measurements of MarkUs and FFmalloc. Since the reported geomeans include the Fortran binaries, we extract the pure C/C++ overheads for each of the systems, and identify the following geomean memory overheads: 19% for MarkUs, 29.5% for FFmalloc, and 8.1% for MineSweeper. The memory overhead of DangZero is comparable with 30%.

6.5 Nginx

In order to evaluate DangZero on more realistic long-running applications, we measured the performance of our system on the popular Nginx web server. We used Nginx version 1.20.2 in combination with the `wrk` HTTP benchmarking tool. We set up two machines

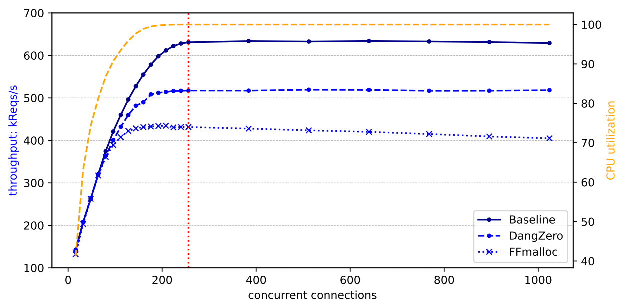

Figure 8: Nginx throughput for DangZero.

and connect them using Mellanox ConnectX 100G NICs, such that we can fully saturate all the CPUs of the server machine. The two machines are identical, containing an Intel Xeon Silver 4110 CPU, 32 GB RAM, and running Ubuntu 20.04 LTS. Similar to our other experiments, we ran Nginx inside a virtual machine for its privilege backend. We assigned 16 GB of RAM to the guest and gave it direct access to the high-speed NIC via PCI passthrough using `vfio`. We configured the `wrk` benchmark to execute 5 iterations of 30 seconds per run, sending a 64-byte file. Since each machine has 16 CPU cores, there are 16 server workers, and 16 client threads. Watermarking is set to 5 million invalidated pages, causing it not to run in 30-second benchmarks. However, in Section 6.5.2 we show the impact of the alias reclaimer on Nginx in detail.

6.5.1 Performance. Figure 8 displays the throughput in (kilo) requests per second of Nginx for the baseline, DangZero, and FFmalloc. The figure shows the buildup in throughput for an increasing number of concurrent connections, as well as the CPU saturation, which reaches 100% for all three configurations around 256 connections. At the point of saturation, where the throughput of Nginx peaks on the baseline, DangZero imposes a throughput degradation of 18%, while FFmalloc's is at 31%.

We also ran a more naive version of DangZero, which does not perform any form of reclaiming. We recorded a throughput degradation of 11%. However, this naive setup is not sustainable for longer runtimes due to alias space exhaustion. We further investigate the scalability of our system and of others in Section 6.5.2.

Unfortunately, most of the other related systems are unable to run Nginx for varying reasons. At its core, the web server relies on the `fork` system call to create worker processes. Oscar and MineSweeper do not support forking of processes. Although support for fork is described in the paper of Oscar, the version the authors shared with us was missing this component. MineSweeper does not take the effects of `fork` into account for their threads and synchronization primitives, and therefore deadlocks. MarkUs runs out of memory (with 32 GB RAM available) while running Nginx at full load. Luckily, Nginx provides settings (intended for development) to run without the need for `fork` [43]. However, the downside of this configuration is that it is more synthetic and cannot reach proper saturation for all the CPUs (i.e., saturating only one core), since the server runs as a single process. We managed to run all of the systems using the forkless setup, with one worker thread, and one client thread.

System	Setup	Throughput degradation
DangZero	Production	18%
FFmalloc	Production	31%
DangZero	No fork	23%
Oscar	No fork	40%
MineSweeper	No fork	56%
MarkUs	No fork	43%

Table 3: Nginx throughput degradation compared.

Table 3 displays the throughput degradations that we obtained in our experiments. For the forkless setup, we pinned the Nginx server to a single core, since in a production environment with full load there are also no other spare cores available to offload threads towards. We confirmed that the single core is fully saturated for all the forkless experiments. The overhead of DangZero shifts slightly, from 18% on the production setup to 23% for the forkless variant.

DangZero introduces the least overhead on Nginx, for both setups, by a significant margin. The difference between DangZero and Oscar is in line with our observations for the SPEC CPU benchmarks, where our design is more efficient at creating and managing alias pages. However, MineSweeper and MarkUs, the two GC-style protection systems, perform significantly worse on Nginx than on the SPEC benchmarks. This is not unexpected, since, unlike SPEC, Nginx features the frequent, short-lived memory object allocations of common real-world, long-running applications. Such allocation patterns result in more aggressive garbage collection and thus a significant overhead—especially evident if the collector threads are not offloaded to other cores. Next, we also see that FFmalloc experiences a large shift in overhead: 31%, whereas the overhead on SPEC was minimal. The high overhead stems from FFmalloc's no-memory-reuse design, which causes access locality to rapidly degrade, combined with frequent kernel interaction for the (de)allocation of objects. DangZero, on the other hand, can efficiently create and invalidate aliases via direct page table access, and does not suffer from slowdown due to TLB pressure, as was the case with some SPEC CPU benchmarks.

6.5.2 Scalability. Nginx is designed to withstand long runtimes without requiring restarts. Therefore, we need to evaluate the capabilities of DangZero to scale to theoretically indefinite runtimes, without suffering from resource exhaustion. This scenario evidences the importance of alias reclaiming or other forms of garbage collection for both the virtual and physical memory footprint.

During the peak throughput of Nginx (at 256 concurrent connections), DangZero imposes a memory overhead of 15% (i.e., the increase in max RSS). This overhead stems from the additional pages required to store the alias page tables. However, if we do not apply compression, this memory overhead raises to 2,400% (25x). Since Nginx frequently allocates memory, without compression the alias page tables host a large number of unused entries. This shows our compression design is effective at drastically reducing physical memory overhead. In comparison, FFmalloc results in a 2,100% (22x) memory overhead on Nginx.

Next, we show that virtual memory address space exhaustion is effectively remedied by our alias reclaimer, while other systems

		Latency (μs)			
Config	**Req/s**	**50p**	**75p**	**90p**	**99p**
Baseline	623,571	395	439	521	663
WM: None	516,605	502	538	603	752
WM: Low	510,668	505	540	616	837
WM: High	507,412	509*	542*	621*	806*
FFmalloc	426,015	532	655	980	2,650

Table 4: Nginx performance vs. different watermarks (WMs).

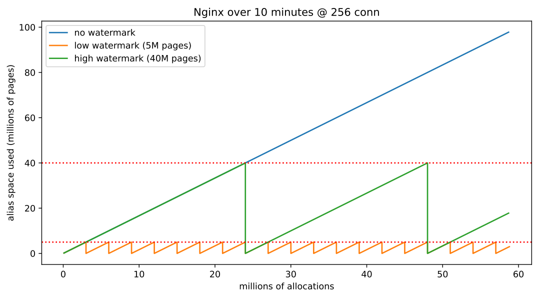

Figure 9: Nginx alias size increase vs. different watermarks.

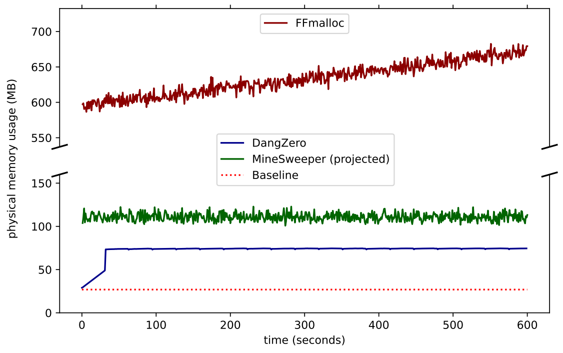

Figure 10: Nginx physical memory usage over time.

like Oscar and FFmalloc can reach exhaustion in a matter of days. Figure 9 displays the growth of DangZero's alias space over the number of memory allocations from saturated Nginx over a period of 10 minutes. We plot the size of the alias space under three different watermark configurations, where the watermark serves as a threshold for when to run the alias reclaimer. Without a watermark in place, that is, by never reclaiming aliases, we see that alias space grows rapidly: reaching nearly 100 million pages in just 10 minutes. With DangZero currently claiming up to 50 TB of kernel address space, extrapolating this graph estimates we run out of alias space within 21 hours. Since Oscar creates the same number of virtual pages as DangZero, and has a theoretical 128 TB of alias (i.e., user) space available, it will reach exhaustion in 53 hours. This matches the ("several days") estimate of the Oscar paper [17].

Since Nginx does most allocations at the page granularity, the number of created alias pages by DangZero corresponds closely to the number of canonical pages. Therefore, FFmalloc exhausts the virtual address space at the same rate per allocation. With an alias space of 128 TB, this again results in a maximal runtime of 53 hours before FFmalloc reaches exhaustion. Note that DangZero could extend its available alias space by also reserving page table entries in user space. We also recognize that the approximations for Oscar and FFmalloc should be stretched slightly due to the additional overhead they impose, effectively reducing the possible number of allocations per second. Nonetheless, our calculations highlight that unbounded address space usage does not scale for allocation-heavy applications with long runtimes in practice.

Some modern systems now feature 5-level paging [16], providing 9 extra bits of usable virtual address space. This could extend the maximum runtime of nonreclaiming systems like Oscar and FFmalloc to around 3 years. The same of course applies to DangZero: in environments where alias reclaiming is not crucial, it could be disabled for extra performance. Also note that this setup is no worst-case scenario for exhaustion, even on Nginx. For example, instead of using worker processes, Nginx can also be configured to use threads (one per core). Since all threads share the same virtual address space, this would decrease the maximum lifetime of Nginx by, in our case, a factor of 16—about 3 hours on most systems and up to 70 days on systems with 5-level paging enabled.

By applying a watermark, DangZero can effectively contain the size of the alias space, as seen in the figure for a watermark of 5 (low) and 40 (high) million pages. As soon as the number of invalidated pages hits the threshold, the alias reclaimer is executed, which on Nginx frees up essentially the entire alias space, since the allocations are short-lived. Fortunately, the repeated alias reclaiming runs come at a marginal cost, as shown in Table 4. The throughput (requests per second) barely decreases as a result of the watermark triggering alias reclaiming, compared to no watermark. We do see an increase in latency, since the alias reclaimer temporarily stops the world. For the high watermark, we actually see that the alias reclaiming stops the world for too long, resulting in 135 read errors and 31 timeouts (0.00005% of requests out of 304 million total). The latency numbers of the high watermark are therefore not completely accurate, since the dropped requests are not included. The low watermark does not experience this issue, and is our recommended setting.

Lastly, we evaluated the physical memory overhead of the default (low) watermark configuration of DangZero on Nginx—again, using a 10-minute execution. Figure 10 displays the physical memory usage over time, when running Nginx with 256 connections. Starting from the baseline, we initially see an increase in memory usage, which is the result of alias space page tables being allocated. Then, after about 30 seconds, the first alias reclaiming run is triggered by the low watermark. This results in an increase of physical memory by certain data structures being created, such as the alias freelist and the compressed (slab) pages freelist. After this initial alias reclaiming run the physical memory remains stable, since DangZero is in a steady state, sustaining on the existing alias and compressed pages freelists. Subsequent runs of the alias reclaimer are triggered by the alias freelist being exhausted, causing the sweeping phase to free all reclaimable alias and compressed pages.

We see that, with a low watermark of 5M pages, DangZero stabilizes at a physical memory usage of roughly 76 MB, which is an increase of 176% compared to the baseline. This is a higher increase than we observed in the 30-second runs, since the triggered watermark causes (roughly constant-size) alias reclaiming data to be created. Additionally, we suspect that the demand for memory causes the heap to double (from `brk`). In comparison, FFmalloc experiences an ever-increasing memory overhead, starting at 596 MB (22x) and growing up to 678 MB (25x) in 10 minutes (at a rate of 12 GB/day). While FFmalloc should be able to reclaim all memory after it is freed, we expect there to be a memory leak in the metadata of the allocator. To bound such increasing physical memory usage, GC-style solutions such as MarkUs and MineSweeper need to ramp up their garbage collection efforts, degrading performance.

For instance, during the 10-minute Nginx run, we observed that MineSweeper had to perform as many as 11,929 sweeps (vs. only 19 for DangZero), despite yielding a projected steady-state physical memory usage of ≈110 MB (notably higher than DangZero's ≈80 MB, as shown in Figure 10). Note that, without fork support, we could not directly measure physical memory usage of MineSweeper (ditto for MarkUs and Oscar) in our production configuration with 16 workers. As such, we measured ≈7 MB of usage for MineSweeper (vs. MarkUs quickly running out of 32 GB memory and Oscar being again nontrivial to measure) for a single worker and projected the usage to 16 workers (accounting for shared pages).

6.6 Apache

In addition to Nginx, we also evaluate DangZero on the Apache HTTP server. We configured Apache to run in its *prefork* (i.e., non-threaded) setting to ensure compatibilty with the non-thread-safe DangZero instrumentation. In order to accurately measure the runtime overhead, we disable the internal caching of the custom memory allocator within Apache. This ensures that Apache does not bypass the instrumentation of DangZero and the other related systems. We confirm that disabling the internal caching functionality does not incur a negative performance penalty, which is attributed to the underlying allocator (`glibc`) being sufficiently optimized (e.g., by batching memory allocations from the kernel).

We used Apache version 2.4.54 in combination with the `wrk` HTTP benchmarking tool. Unfortunately, we do not manage to reach full CPU saturation using the two connected machines used for Nginx. Therefore, we instead run Apache in a loopback fashion on the machine described in Section 6.3, again inside a virtual machine. With this setup, we manage to reach CPU saturation by running the `wrk` benchmark with 7 client threads, and Apache with 256 worker processes. The remaining settings are identical to the ones described for Nginx in the previous section.

We attempt to run Apache with all of the related systems in order to compare with DangZero, however only FFmalloc results in successful execution. As seen with Nginx, both Oscar and MineSweeper cannot overcome the forking behavior of Apache, and MarkUs runs out of memory again. We manage to reach CPU saturation for the baseline, DangZero, and FFmalloc at 176 concurrent connections, with a throughput degradation of 19.1% for DangZero, and 28.7% for FFmalloc. These runtime overhead results closely correspond to the ones observed for Nginx, which suggests that our results reflect a general trend on real-world applications.

6.7 Virtualization

Since our overlay allocator requires isolation from other processes to execute in a safe manner, some form of virtualization is a requirement for most use cases. While the cost of virtualization may greatly vary across applications and is generally considered negligible [46], we also investigated its impact in our setup—compared to the bare-metal host as baseline. For SPEC CPU2006, we observed a geomean runtime overhead of 4.2%, which can mostly be attributed to the 464.h264ref binary experiencing a 33% slowdown. For Nginx, we measured an approximate throughput degradation of 8%.

However, the cost of virtualization can be more than compensated by benefiting from the unikernel-like design of our system, for example with unikernel optimizations proposed by Lupine Linux [29]. For instance, Lupine Linux shows that a 33% throughput increase can be achieved on Nginx by merely stripping features off the Linux kernel that are not needed in a unikernel setup. For a fair comparison with the state of the art, we did not enable these and other unikernel optimizations in DangZero (which would have likely resulted in an overall speedup of DangZero-instrumented Nginx running in virtualized KML compared to a bare-metal baseline). Furthermore, if the target environment on which to apply DangZero is already virtualized (e.g., in the cloud), then the overhead of virtualization is not applicable.

7 RELATED WORK

In this section, we review prior work on UAF defenses.

7.1 Secure allocators

Much of the literature focuses on the mitigation of UAF by means of secure allocators [4, 19, 20, 31, 33, 48, 54, 56]. Early solutions [7, 44] provide probabilistic protection through randomized allocation—making it difficult (not impossible), for the attacker to target specific objects. Moreover, some also re-initialize the free memory [44].

Solutions such as Cling [5] and TAT [52] do not prevent UAF *per se*, but ensure that all reuse is type safe. UAF is still possible, but limited to using dangling pointers to objects of the same type.

Another common technique builds on garbage collection (GC) [4, 10, 19, 23]. For instance, the system may cause deallocations to be quarantined until the GC establishes that no references to them remain. A drawback is that a GC running concurrently is not cheap, especially since strong security requires regular stop-the-world synchronization. As an alternative, CRCount tracks the number of references to each object and releases freed memory for reuse when the reference count reaches zero [50]. An extreme solution, used in FFmalloc [54], is never to reuse and unmap all freed memory.

7.2 Use-after-free detectors

On the detection side, we find various sanitizers. Some explicitly track pointers to each object and invalidate them when it is freed [30, 49, 53, 55, 57]. All subsequent dereferences of the pointers result in a crash. However, tracking all pointers to objects throughout the execution requires expensive metadata management.

Instead of pointer tracking, it is also possible to focus on the allocations, although such approaches are generally less accurate. A simple technique, used by ASan [47], protects allocations with poison values (where each access terminates the process). ASan additionally places freed allocations in quarantine to reduce the probability of temporal memory errors—a simple measure, vulnerable to massaging through continuous allocation and deallocation. Moreover, ASan has high performance/memory overhead and offers limited security guarantees against advanced attacks. Other approaches, such as xTag [9], Vik [15] and others [13, 22, 25, 41] tag the allocated memory and the pointer with an identifier and verify, upon dereference, that the tags of pointer and memory match.

While few detectors apply to binary programs, there are exceptions. Early work on memory debugging unmapped memory upon free [38, 45] or used dynamic binary instrumentation [42]. While useful for debugging, such approaches incur a very high memory and performance overhead. Recently, Google Chrome has integrated *sampled* page-permission-based heap memory error detection for end users, resulting in over a hundred (and predominantly) use-after-free bugs being discovered post deployment in the field [51]. Some alias-based approaches also work on binary programs [17, 18]. Like DangZero, these solutions map allocations to new virtual pages (aliases). Unlike DangZero, existing work does not properly handle memory reclamation.

7.3 Virtualization for security

Finally, there is significant work on virtualization for security, especially on secure hypervisors [37]. Some projects use virtualization extensions for secure domain isolation, using unprivileged instructions to switch EPT-based domains [28, 35, 46]. Of particular interest to DangZero are prior efforts to make privileged instructions available to applications in guest mode (unikernel-style) to support mitigations, such as the sandboxing environment of Dune [6] and the efficient multi-variant execution support of MvArmor [27]. In DangZero, we use the ring 0 access to implement a highly efficient alias-based detector of UAF via direct page table access.

8 LIMITATIONS

Our current DangZero prototype has two main limitations that can be addressed with more engineering effort. First, since we build on KML, DangZero cannot currently run on Linux kernel versions newer than v4.0 (the latest release for KML). We believe porting KML to a newer kernel version is feasible, although mitigations such as kernel page table isolation (KPTI) require special handling. Second, the current implementation of DangZero is not yet thread-safe. Thread safety can be addressed with page table locking, as also suggested by prior systems that support direct page table access [6]. Additional limitations as a consequence of running programs in isolation, for example not sharing memory across processes, require hypervisor-based solutions such as cross-VM page sharing.

On a more fundamental level, our design has two other limitations. First, similar to all the state-of-the-art solutions [4, 17, 19], DangZero targets the default (`malloc`-family) allocator. However, applications may also rely on custom allocators (based on `malloc`, `mmap`, etc.) [8]. Simple custom allocators (e.g., plain `malloc` wrappers such as Nginx' `ngx_calloc`) are supported out of the box, but others may require custom instrumentation. Nonetheless, since our instrumentation strategy is based on an overlay allocator, one could, in principle, implement a custom overlay allocator for each custom allocator in the target application. Second, applications may rely on custom pointer encoding. Common encodings (e.g., slab pointers in Nginx, with custom metadata in the lowest 2 bits—see `NGX_SLAB_PAGE_MASK`) yield a memory representation preserving the pointer-object relationship and are supported out of the box. However, more sophisticated encodings may hinder our alias reclaimer's ability to locate some (dangling) pointers in memory/registers, a well-known limitation shared with all the solutions based on conservative garbage collection techniques [4, 10, 19, 34]. If necessary, an option is to explicitly implement support for custom encodings on a per-application basis, but the resulting implementation complexity may vary.

9 CONCLUSION

Using page protection facilities offered by the MMU to detect use-after-free bugs is an old idea originally devised for debugging. Over two decades later, despite modern alias-based optimizations, state-of-the-art defense solutions still suffer from nontrivial performance and scalability costs. With DangZero, we show these costs are not fundamental and a practical solution is possible. Using direct page table access, DangZero eliminates the operating system from the fast (allocation/deallocation) path and also unlocks efficient metadata management for GC-style alias reclaiming. Our evaluation shows our design significantly improves the performance of prior alias-based solutions—without incurring memory exhaustion or relaxing detection guarantees. On long-running benchmarks that commonly feature frequent, short-lived allocations, DangZero is also significantly more efficient than state-of-the-art GC-style protection systems—without resorting to spare CPU cores or memory overprovisioning. Finally, our design is amenable to arbitrary unikernel-style optimizations described in literature, which one could adopt to reduce DangZero's overheads even further.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback. We also thank Gabor Kozar and Kyrian Maat for their preliminary work on the Dune and KML implementations, respectively. This work was supported by the EU's Horizon 2020 programme under grant agreement No. 825377 (UNICORE), the Dutch Ministry of Economic Affairs and Climate through the AVR program (Memo project) and the Dutch Science Organization NWO through projects TROPICS, Theseus, and Intersect.

REFERENCES

[1] CVE-2015-2787. PHP 5.5.14 Use-After-Free Vulnerability. https://bugs.php.net/bug.php?id=68976
[2] CVE-2015-6835. PHP 5.4.44 Use-After-Free Vulnerability. https://www.exploit-db.com/exploits/38123
[3] CVE-2016-5773. PHP 7.0.7 Use-After-Free Vulnerability. https://bugs.php.net/bug.php?id=72434
[4] Sam Ainsworth and Timothy M. Jones. 2021. MarkUs: Drop-in use-after-free prevention for low-level languages. In *USENIX Security*.
[5] Periklis Akritidis. 2010. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security*.
[6] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazieres, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *OSDI*.
[7] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI*.
[8] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering custom memory allocation. In *OOPSLA*.

[9] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davi. 2022. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *IEEE EuroS&P*.

[10] Hans-J Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly parallel garbage collection. In *PLDI*.

[11] Jeff Bonwick et al. 1994. The slab allocator: An object-caching kernel memory allocator. In *USENIX ATC*.

[12] Jeremy Brown. CVE-2015-3205. Libmimedir VCF Memory Corruption Proof Of Concept. https://packetstormsecurity.com/files/132257/Libmimedir-VCF-Memory-Corruption-Proof-Of-Concept.html

[13] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2018. CUP: Comprehensive user-space protection for C/C++. In *AsiaCCS*.

[14] Zixian Cai, Stephen Blackburn, Michael Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors. In *ISPASS*.

[15] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: practical mitigation of temporal memory safety violations through object ID inspection. In *ASPLOS*.

[16] Jonathan Corbet. 2017. Five-level page tables. https://lwn.net/Articles/717293.

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

[18] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently detecting all dangling pointer uses in production servers. In *DSN*.

[19] Márton Erdős, Sam Ainsworth, and Timothy M. Jones. 2022. MineSweeper: A Clean Sweep for Drop-In Use-after-Free Prevention. In *ASPLOS*.

[20] Daniel Micay et al. 2019. Hardened malloc. https://github.com/GrapheneOS/hardened_malloc.

[21] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. *BSDCan* (2006).

[22] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security*.

[23] Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, et al. 2020. Cornucopia: Temporal safety for CHERI heaps. In *IEEE S&P*.

[24] Dinko Galetic and Denis Kasak. 2017. Use-After-Free Leading to An Invalid Pointer Dereference. https://hackerone.com/reports/213261

[25] Binfa Gui, Wei Song, and Jeff Huang. 2021. UAFSan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In *ISSTA*.

[26] Frederick Boland Jr. and Paul Black. 2012. The Juliet 1.1 C/C++ and Java Test Suite. *IEEE Computer* (2012).

[27] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization. In *DSN*.

[28] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No need to hide: Protecting safe regions on commodity hardware. In *EuroSys*.

[29] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *EuroSys*.

[30] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS*.

[31] Daan Leijen. 2020. Mimalloc. https://github.com/microsoft/mimalloc.

[32] John Leitch. Issue 24613. array.fromstring use after free. https://bugs.python.org/issue24613

[33] Beichen Liu, Pierre Olivier, and Binoy Ravindran. 2019. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *Middleware*.

[34] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *CCS*.

[35] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *CCS*.

[36] Toshiyuki Maeda and Akinori Yonezawa. 2003. Kernel Mode Linux: Toward an operating system protected by a type theory. In *ASIAN*.

[37] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P*.

[38] Microsoft. 2022. GFlags and PageHeap. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap

[39] Matt Miller. 2019. Trends and Challenges in the Vulnerability Mitigation Landscape. https://www.usenix.org/conference/woot19/presentation/miller

[40] MITRE. 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ISMM*.

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*.

[43] Nginx. 2022. Run nginx with "daemon off" or "master_process off" settings in a production environment. http://nginx.org/en/docs/faq/daemon_master_process_off.html

[44] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *CCS*.

[45] Bruce Perens. 1987. Electric Fence. https://elinux.org/Electric_Fence

[46] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective memory protection for kernel and user space. In *IEEE S&P*.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*.

[48] Kostya Serebryany and Dmitry Vyukov. -. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html.

[49] Zekun Shen and Brendan Dolan-Gavitt. 2020. HeapExpo: Pinpointing promoted pointers to prevent use-after-free vulnerabilities. In *ACSAC*.

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

[51] Vlad Tsyrklevich. 2019. GWP-ASan: Sampling heap memory error detection in-the-wild. https://sites.google.com/a/chromium.org/dev/Home/chromium-security/articles/gwp-asan

[52] Erik Van Der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and complete type-safe memory reuse. In *ACSAC*.

[53] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable use-after-free detection. In *EuroSys*.

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

[55] Yves Younan. 2015. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *NDSS*.

[56] Insu Yun, Woosun Song, Seunggi Min, and Taesoo Kim. 2021. HardsHeap: A Universal and Extensible Framework for Evaluating Secure Allocators. In *CCS*.

[57] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *ASPLOS*.

Benchmark	b-Host	b-VM	b-jemalloc	DangZero	Oscar	FFmalloc	MarkUs	MineSweeper
400.perlbench	183	187	176	344	785	187	226	209
401.bzip2	303	311	304	302	308	304	304	304
403.gcc	161	165	206	228	203	241	363	239
429.mcf	176	196	176	181	176	176	176	175
433.milc	300	313	305	324	318	323	316	307
444.namd	257	258	257	257	257	256	257	257
445.gobmk	318	321	319	324	320	319	322	321
447.dealII	196	193	194	271	580	197	211	209
450.soplex	144	147	146	150	150	155	157	149
453.povray	90.7	97.5	89.2	101	95,6	89,1	90,6	89,6
456.hmmer	259	256	206	260	261	224	262	208
458.sjeng	357	354	357	367	362	357	357	357
462.libquantum	200	208	199	204	205	198	199	199
464.h264ref	331	499	318	495	347	326	322	319
470.lbm	176	176	175	177	176	177	176	179
471.omnetpp	226	237	187	480	1026	201	326	265
473.astar	281	294	263	347	392	263	277	269
482.sphinx3	323	334	323	336	350	328	343	343
483.xalancbmk	131	139	97,8	264	533	130	321	258

Table 5: Absolute values of CPU SPEC 2006 runtimes. (b- indicates baseline)

Benchmark	b-Host	b-VM	b-jemalloc	DangZero	FFmalloc	MarkUs	MineSweeper
600.perlbench_s	271	282	272	362	279	309	300
602.gcc_s	386	397	380	519	-	625*	505*
605.mcf_s	567	619	564	710	556	557	559
619.lbm_s	946	956	957	965	944	945	949
620.omnetpp_s	364	402	329	836	357	702	410
623.xalancbmk_s	284	287	169	734	236	437	438
625.x264_s	375	379	375	379	375	375	375
631.deepsjeng_s	341	352	341	356	341	341	341
638.imagick_s	6341	6855	6312	6832	6320	6319	6319
641.leela_s	455	474	453	512	468	493	499
644.nab_s	2114	1978	2110	1980	2125	2133	2131*
657.xz_s	2241	2333	2238	2345	2240	2237	2240

Table 6: Absolute values of CPU SPECspeed 2017 runtimes. (b- indicates baseline, * indicates reported ratio)