Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing

Wookhyun Han
KAIST

Byunggill Joe
KAIST

Byoungyoung Lee
Purdue University

Chengyu Song
University of California, Riverside

Insik Shin
KAIST

***Abstract*—Memory errors are one of the most common vulnerabilities for the popularity of memory unsafe languages including C and C++. Once exploited, it can easily lead to system crash (i.e., denial-of-service attacks) or allow adversaries to fully compromise the victim system. This paper proposes MEDS, a practical memory error detector. MEDS significantly enhances its detection capability by approximating two ideal properties, called an infinite gap and an infinite heap. The approximated infinite gap of MEDS setups large inaccessible memory region between objects (i.e., 4 MB), and the approximated infinite heap allows MEDS to fully utilize virtual address space (i.e., 45-bits memory space). The key idea of MEDS in achieving these properties is a novel user-space memory allocation mechanism, MEDSALLOC. MEDSALLOC leverages a page aliasing mechanism, which allows MEDS to maximize the virtual memory space utilization but minimize the physical memory uses. To highlight the detection capability and practical impacts of MEDS, we evaluated and then compared to Google's state-of-the-art detection tool, AddressSanitizer. MEDS showed three times better detection rates on four real-world vulnerabilities in Chrome and Firefox. More importantly, when used for a fuzz testing, MEDS was able to identify 68.3% more memory errors than AddressSanitizer for the same amount of a testing time, highlighting its practical aspects in the software testing area. In terms of performance overhead, MEDS slowed down 108% and 86% compared to native execution and AddressSanitizer, respectively, on real-world applications including Chrome, Firefox, Apache, Nginx, and OpenSSL.**

I. INTRODUCTION

For the popularity of memory unsafe languages like C and C++, memory errors are one of the most common software bugs, especially in large-scale software such as browsers and OS kernels. Memory errors are also one of the most severe bugs from the security perspective—they can easily lead to system crash (*i.e.*, denial-of-service attacks) or even allow adversaries to take full control of the vulnerable system (*i.e.*, arbitrary code execution and privilege escalation). In the past few decades, numerous solutions have been proposed to prevent memory error related attacks [34]. These defense techniques can be put into two general directions: exploit mitigation techniques and memory error detectors.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23312
www.ndss-symposium.org

Exploit mitigation techniques focus on preventing attackers from utilizing memory errors to perform malicious activities. Because these techniques tend to have lower runtime performance overhead (< 10%), most widely deployed mechanisms belong to this category, such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), and Control-Flow integrity (CFI). However, their limitation is also obvious: they can be easily bypassed by new exploit techniques—from code injection to return-oriented program [33] to advanced ROP attacks [29] to data-only attacks [16], attackers have always been able to find new creative ways to exploit memory errors.

Memory error detectors [17, 21, 22, 24, 25, 31], on the other hand, aim to detect the root cause. Since these detection techniques can stop the attacks from happening in the first place, they have the capability to prevent all memory error related attacks. Unfortunately, achieving this is not without cost. First, these techniques tend to have relatively high performance overhead, ranging from 30% for hardware-based approach [17, 23] to over 100% for pure software-based approaches [21, 22, 24, 25, 31]. Second, some of them have difficulties in supporting the full language features of C and C++ (see §VIII for more details).

Despite their drawbacks, we believe memory error detectors are the more promising direction to fundamentally prevent memory error related attacks. More specifically, to defeat existing attacks, we have already accumulated a large set of exploit mitigation techniques. For example, the latest Windows system (Windows 10) has the following exploit mitigation techniques deployed: DEP, ASLR, stack guard, control-flow guard, return-flow guard, etc. However, as attackers are now shifting to data-only attacks [16] and information leak attacks [32], new mitigation techniques must be added. The problem is, even though the performance overhead of each individual mitigation technique may be low, the accumulated overhead could still be high, especially for defeating data-only attacks (*e.g.*, data-flow integrity [7]) and information leak (*e.g.*, dynamic information-flow tracking [37]). Yet, they still could not provide the strong security guarantees compared to memory error detectors.

Motivated by the above reasons, we present MEDS, a system that enhances the detectability of *redzone-based memory error detection*. In particular, existing memory error detectors can be categorized into two directions: redzone-based and pointer-based. Redzone-based detectors insert undefined memory between memory objects and prohibit access to the undefined area. Pointer-based detectors keep track of per-pointer capability and check the capability when accessing an object. Generally, redzone-based detectors have better compatibility with C/C++

features but their ability to detect memory errors is not as powerful as pointer-based one (refer to §II for more details).

The key idea behind MEDS is that full 64-bit virtual address space can be leveraged to approximate "infinite" gap between allocated memory regions (so as to detect spatial errors) and "infinite" heap (so as to avoid reusing freed memory and to detect temporal errors). More importantly, MEDS achieves this without blowing up physical memory use. MEDS realizes this idea through a new memory allocator, MEDSALLOC. MEDSALLOC uses user-space page aliasing mechanism (*i.e.*, aliasing between physical and virtual memory pages) to manage memory pools, thereby maximizing the virtual address utilization while minimizing the physical memory uses. The "infinite" gap allows MEDS to detect more spatial memory errors exhibiting a large out-of-bound offset than the state-of-art redzone-based memory error detector AddressSanitizer [31]. MEDS also detects temporal memory errors more robustly, as it fully utilizes available virtual address space for allocation and thus the virtual address is unlikely reused.

We have implemented MEDS based on the LLVM toolchain, and evaluated the prototype of MEDS on various real-world large applications, including Chrome, Firefox, Apache, Nginx, and OpenSSL. First, we evaluated MEDS on a set of unit tests and MEDS was able to correctly detect all the tested memory errors. Then we tested MEDS using four real-world memory corruption exploits in Chrome and Firefox, and MEDS showed three times better detection rates than AddressSanitizer (ASAN), a state-of-the-art memory error detection tool developed by Google. MEDS imposed a moderate runtime overhead—on average, MEDS slowed down 108%, which is comparable to ASAN; and it used 212% more memory.

Utilizing on MEDS's detection capability, it can be applied to detect lurking memory errors in production servers or fuzzing infrastructures, similarly ASAN has been popularly deployed and used in practice. To clearly demonstrate this aspect, we performed a fuzz testing with AFL [38], targeting 12 real-world applications. To summarize, MEDS significantly outperformed ASAN in assisting memory error detection capability of fuzzing for most of target applications — 68.3% improvements on average, ranging from 1% to 256%, depending on applications (shown in Table IV). Considering the huge popularity of AFL and ASAN in performing real-world fuzz testing, these results also signify the strong practical impacts of MEDS. Using with AFL, MEDS can augment the fuzz testing's detection capability, significantly better than the state-of-the-art memory error detection tool, ASAN. We note that ASAN is part of both GCC (since v4.8) and LLVM/Clang (since v3.1) mainlines, and many major vendors and open-source community heavily rely on ASAN for debugging and fuzz testing.

In summary, this paper makes the following contributions.

- **Design.** We designed MEDS, a new memory error detector with enhanced detection capability. The core of MEDS is MEDSALLOC, a new memory allocator that (1) fully utilizes the 64-bit virtual address space to provide "infinite" gaps between objects and to avoid reusing freed virtual addresses; and (2) leverages a novel memory aliasing scheme to minimize the physical memory overhead.
- **Implementation and Evaluation.** We implemented a prototype of MEDS based on the LLVM toolchain and have successfully applied it to a set of large real-world applications including Chrome, Firefox, Apache, Nginx, and OpenSSL. We evaluated several aspects of MEDS including (1) its compatibility, (2) its detection capability against artificial and real attacks, and (3) its runtime performance and memory overhead.
- **Practical Impacts.** According to our evaluation in the fuzz testing (using AFL), MEDS significantly outperformed ASAN in terms of detecting memory errors. We plan to opensource MEDS so that software vendors and open source communities can benefit from using MEDS. As demonstrated in our evaluation, MEDS is already mature enough to be released and used for real-world applications.

II. BACKGROUND AND CHALLENGES

A. Memory Errors

There are two general types of memory errors: *spatial errors* and *temporal errors*. Spatial memory errors refer to accessing memory that are outside the boundary of the allocated memory. Such errors can be caused by many types of software bugs, including missing boundary checks, incorrect boundary checks, insufficient memory allocation, type confusion, etc. Temporal memory errors can be further put into two sub-categories: reading *uninitialized* memory and accessing *freed* memory. Reading uninitialized memory can be problematic because its value is either unpredictable or can be controlled by attackers. Accessing freed memory is problematic because the freed memory can be *reallocated* to store another memory object, which may be controlled by attackers.

Hicks [15] formalize the definition of memory errors into two styles:

- **Access to undefined memory.** A memory region is undefined if it has not been allocated (out-of-bound), has not been initialized, or has been freed. While this definition is simple, it is not realistic. To support this definition, the gap between any two allocated regions must be infinite (i.e., infinite gap) and a freed memory region must never be reused (i.e., infinite heap).
- **Violation to the capability of a pointer.** The second definition associates a pointer with a capability to access memory between `base` and `end`. Capabilities can only be created through legal operations like allocation and the addresses taken thus are not forgeable; and are revoked (*i.e.*, has no capability) when the corresponding memory region is freed. A memory error can then be defined as accessing memory outside the capability of a pointer.

Following the definition above, existing approaches to detect memory errors can then be generally categorized into two directions: (1) *redzone-based detection*, which inserts undefined memory between objects and detects an access to the undefined area; and (2) *pointer-based detection*, which keeps track of per-pointer capability and checks capability when accessing an object. Both directions have their own advantages and disadvantages. Generally, redzone-based memory error detectors have better compatibility with C/C++ language features and thread model, so they can be applied to large-scale software like browsers. Pointer-based detectors usually have compatibility issues. For example, SoftBound is not compatible with some

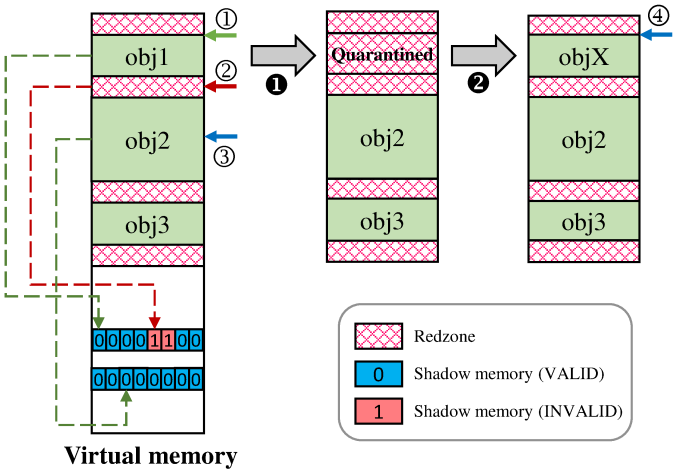

Fig. 1: Redzone-based detection in ASAN using redzone and shadow memory. In the beginning there are three allocated objects (leftmost), and then `obj1` is freed (middle). If the quarantine zone is exhausted due to repeated allocations, the freed space can be reused (rightmost).

SPEC CPU benchmarks [21] and GCC's support for Intel MPX (memory protection extension) is also reported to have compatibility issues [30]. On the other hand, pointer-based solutions usually have better detection capability as it is not realistic to implement infinite gap between objects and never reuse freed memory. Because we aim to build a practical tool that can support large-scale C/C++ programs utilizing various language features, we opt to follow the redzone-based direction and our description will focus on redzone-based detection in this section. We will describe more details on pointer-based detection in §VIII.

B. Redzone-based Memory Error Detection

Redzone-based detectors [5, 20, 25, 26, 31] insert undefined memory regions (*a.k.a.* redzones) between valid memory objects. These detectors then setup mechanisms to capture attempts accessing redzones (e.g., no virtual page permissions) such that access to such a region can be detected. In general, there are two key design factors in redzone-based approaches, namely (1) how to setup redzones to increase detection rates and (2) how to actually detect an access attempt to redzones. For instance, to detect temporal errors, DieHard [5] and its successor DieHarder [26] populate newly allocated memory and freed memory with *magic* values hoping that a later use of the magic value will cause catchable errors. They also add redzones around the allocated memory region to detect spatial errors. Out-of-bound read is captured in the same way as in detecting temporal errors. Out-of-bound write is captured by checking if magic value of the redzones has been modified when the memory is freed. Paged-heap [20] surrounds allocated region with two extra memory pages (one in each direction) of no access permission so that out-of-bound accesses will trigger page faults. Valgrind [25] uses valid value bit and valid address bit to capture reading undefined memory and out-of-bound access.

AddressSanitizer (ASAN [31]) is by far the most mature redzone-based memory error detector which is supported by both Clang and GCC. It demonstrated a good balance between detection precision and performance, and is able to handle large complex software like Google Chrome and Firefox. The key to ASAN's high efficiency is in how it represents redzones using *shadow memory* (illustrated in Figure 1). Shadow memory is a bit-vector, showing valid/invalid memory addresses. One bit in the shadow memory represents one byte in target application's virtual memory space, where the bit `0` in the shadow memory represents valid and the bit `1` represents invalid. ASAN enforces that all memory read and write operations must first refer to the shadow memory to check validity of the target address (*i.e.*, the corresponding bit in the shadow memory should be `0`). To detect out-of-bound access, ASAN surrounds all memory objects (include stack and global objects) with redzones. Moreover, to detect use-after-free, ASAN marks the whole freed region as redzone when an object is freed. Then ASAN maintains a fixed size of quarantine zone (i.e., 256 `MB` by default) to avoid reusing freed memory (i.e., ASAN does not really release the freed memory regions, but keeps holding those regions in quarantine zone until the quarantine zone becomes full). For example, when the object `obj3` is freed, the corresponding region is marked as redzone by updating the corresponding shadow memory bits as invalid (illustrated in Figure 1 - ❶). This freed region will be kept in quarantine zone to avoid reuse.

While this approach is similar to the valid address bit of Valgrind, ASAN's special shadow memory address scheme [31] makes the checks much faster. In particular, ASAN uses a direct mapping scheme to locate the shadow memory in that it simply performs bit-shift operations on the virtual address to obtain the corresponding shadow memory location. This in fact requires to reserve certain virtual address space for the shadow memory, but it is efficient as locating the corresponding shadow memory only involves a simple bit-shift instruction. ASAN has shown very good compatibility with existing code in practice. It has no issues in supporting large scale software like browsers, and it is the default memory error detector for Google's cloud-based fuzzing platform [27].

Limitations of Existing Redzone-based Detectors. As discussed previously, the detectability of a redzone-based memory error detector relies on (1) how large the redzones between objects are and (2) how long freed objects remain as redzones. Specifically, if an out-of-bound access falls into another allocated memory region or use-after-free access falls into a re-allocated memory region, then the error cannot be detected. Unfortunately, existing redzone-based memory error detectors all failed to implement or approximate the infinite gap and infinite heap requirements properly; so their detectability is limited. For example, by default ASAN sets up the redzone in the range of 16 `Byte` and 2,048 `Byte`, which can be easily bypassed by skipping over this redzone. Moreover, considering the infinite heap, the default size of its quarantine zone is only 256 `MB`; so its detection for temporal errors can also be bypassed if a program keeps (or an attacker induces a program to keep) allocating memory objects to force reuses.

To clarify these limitations, Figure 1 illustrates memory layouts as well as its redzone enforcement through the shadow memory. In the beginning there are three allocated objects, `obj1`, `obj2`, and `obj3` (leftmost). In this setting, suppose a program performs a pointer arithmetic operation, `p = p + idx`, where `p` is a pointer that is originally pointing the base address of `obj1` (i.e., ①) and `idx` is an integer variable. Then further suppose

that the program dereferences using `p`. Checking the shadow memory bits, ASAN can correctly determine that dereferencing is valid when `idx` is zero (*i.e.*, ①) and invalid when `idx` is the size of `obj1` (*i.e.*, ②), respectively. However, if `idx` is bigger than the size of `obj1`, it is possible that the dereference can be allowed according to the shadow memory bit although this should not be allowed (*i.e.*, ③). Moreover, although freed `obj1` region will be kept in quarantine zone (*i.e.*, after ❶), such a region is reused if the quarantine zone is exhausted due to repeated allocations (*i.e.*, reused for `objX` after ❷). Thus, if there is another memory dereference using `p` after `obj1` is freed, it can result in use-after-free (*i.e.*, ④).

To see the real-world implications of this limitation, we also tested four real-world exploits and found that ASAN was easily bypassed (see Table I). We note it is fundamentally challenging to enlarge these parameters in ASAN, because it would significantly increase the memory uses.

III. Problem Scope and Objectives

In this section we define the problem scope of this paper, our objectives, and key evaluation metrics we aim to achieve.

Problem Scope. This work focuses on the problem of *memory error detection* for user space C/C++ programs. We assume the operating system kernel, all firmware, and all hardware as our trusted computing base (TCB). We do not consider attacks targeting our TCB or launched from within our TCB. We also do not consider exploit against vulnerabilities other than memory errors or memory errors in other languages (*e.g.*, assembly and dynamically generated code). We do not restrict which language features the target program can use or where the memory errors can occur—the vulnerability can exist in the main executable or any linked libraries. We also do not restrict how attackers can exploit the vulnerability.

Objectives. As discussed in §II, different memory detectors have different capabilities in detecting memory errors—some of them can only detect spatial errors [17, 20, 21, 24, 36], some can detect use-after-free but not uninitialized memory [22, 31], and some can detect all types of errors [5, 25, 26]. Their detection rate also varies, some can only provide probabilistic detection [5, 26], some can provide deterministic detection but can be bypassed [20, 25, 31], and some can detect all occurrence of the error they can detect thus can provide the strong memory safe guarantee [17, 21, 22, 24, 36].

In this work, we aim to *enhance the detectability on memory errors for large-scale C/C++ programs.* There are two goals in this statement. First, we aim to handle large-scale programs such as popular server applications and browsers. We choose them as the target because of their importance and the belief that security solutions must be practical to make real impact. Second, we want to provide better detectability than existing solutions for large-scale programs. However, providing lower runtime performance overhead is not our primary goal—we will try our best to reduce the performance overhead, but when there is a trade-off between detectability and performance, we will opt for the detectability.

Evaluation Metrics. Given the current status quo (§II), to achieve our goals, we can either try to solve the compatibility issue of pointer-based solutions, or try to improve the detectability of redzone-based solutions. This work explores the second direction, and our evaluation metrics are: (1) MEDS must be able to run all the programs, the state-of-the-art redzone-based solution, ASAN can handle; (2) MEDS must be able to detect more memory errors than ASAN; and (3) the runtime performance and memory overhead must be comparable to ASAN.

IV. Design

This section presents the design of MEDS. §IV-A illustrates the design overview of MEDS. Then §IV-B introduces MEDSALLOC, a new memory allocator with page aliasing. Then §IV-C describes how MEDS manages and enforces inaccessible memory regions, redzone. §IV-D describes how all memory objects (including heap, stack, and global objects) are allocated through MEDSALLOC, such that MEDS comprehensively provides redzone for all kinds of memory objects. Lastly, §IV-E presents user-level copy-on-write schemes for MEDS.

A. Overview

MEDS takes a redzone-based approach to detect memory errors because it provides the best compatibility among the two different directions (§II). As suggested by its name, a redzone-based approach detect memory errors by inserting *redzones* (undefined memory regions) between memory objects and marking freed memory objects as *redzones*. Therefore, the detectability of a redzone-based memory error detector depends on how closely it can approximate the two ideal properties:

- **P1: Infinite gap.** To detect all spatial errors, the redzone between two memory objects must be *infinite* so that out-of-bound accesses will always fall into the redzones.
- **P2: Infinite heap.** To detect all temporal errors, a new memory object must always be allocated from a fresh virtual address so that the freed region (redzone) will never be re-used during the execution.

Unfortunately, due to limited hardware resources (both physical and virtual memory space) imposed by the current computing architecture, it is not feasible to fully satisfy these properties. Thus, state-of-the-art redzone-based detection tools make practical design trade-offs between security risks and resource consumption. For instance, by default ASAN [31] only inserts a 256 `Byte` redzone between memory objects to detect spatial errors and only maintains a 256 `MB` heap quarantine zone to detect temporal errors. Enlarging any of these two parameters imposes heavy physical memory usage unsuitable for large-scale programs. To clearly demonstrate this, we tried to experiment ASAN with these enlarged settings: for the redzone size, ASAN includes hard-coded assertions and design decisions limiting these parameters and thus we were not able to run; for quarantine zone, ASAN quickly used up all physical memory space if a large quarantine zone size is provided, and got killed due to out-of-memory. As a result, if a spatial memory error happens beyond the redzone size, such a memory access violation cannot be detected. Similarly, freed memory will be recycled when the quarantine zone is full, resulting in undetectable temporal errors. Our evaluation in §VI clearly demonstrates this limitation in that four real-world

vulnerabilities in Chrome and Firefox were easily bypassed by slightly modifying an input.

MEDS improves the approximation towards these two ideal properties through fully utilizing the 64-bit virtual address space. Specifically, the 64-bit virtual address space has provided us with a great opportunity to (1) increase the redzone size between objects and (2) reduce virtual address reuse. The challenge, however, is how to minimize the physical memory usage. MEDS overcomes this challenge through a novel combination of *page aliasing* and *redzones*. Page aliasing denotes the intentional aliasing between virtual and physical memory pages (*i.e.*, a set of different virtual pages are mapped to the same physical page), which is a common technique used to reduce use of physical pages, such as in copy-on-write (CoW) and same-page merge [4]. However, redzone enforcement with page aliasing normally comes with the potential for increasing the fragmentation significantly. This is because the granularity of memory object allocation differs from that of page access permission. That is, all the objects within the same virtual page have to share the same access permission. This makes it complicated to perform access checks when a single virtual page contains both a valid object and a redzone. One approach to overcome this is, as suggested in PageHeap [20], that maps all the redzone virtual pages (i.e., containing redzones only without any valid objects) to a single physical page while increasing the allocation granularity to page level (*i.e.*, allocating at most one object in a single virtual page) at the cost of generating the internal fragmentation. Thus, MEDS aims to over-provision virtual memory space, yet without wasting physical memory, to meet both **P1** and **P2**. Toward this end, we design new redzone schemes for MEDS. Since MEDS intensively makes use of huge virtual address space, simply adopting ASAN's shadow-memory based redzones would require impractical physical memory space to store shadow-memory itself. Thus, MEDS orchestrates page access permission settings as well as shadow-memory based redzones to efficiently manage and enforce redzones for all invalid memory space.

B. MEDSALLOC*: A Memory Allocator with Page Aliasing*

To implement the above idea, we design MEDSALLOC, a new user-space allocator which maintains the special mapping between virtual and physical page and redzone setup. With MEDSALLOC, we can provide each memory objects with virtual view as if they don't share their page with others. Thus, while objects are tightly packed in the physical memory space, those are sparsely located in the virtual memory space (Figure 2). This allows MEDS to meet **P1** with low memory overheads—the target program now sees large redzones between objects, but these only impose a small memory use as the redzones are not actually backed by physical memory dedicated for the redzones. It is worth noting that MEDSALLOC only uses shadow memory to mark redzone at sub-page level (red color boxes), and page level gaps (denoted as dots in Figure 2) are still marked by page table permissions. This further reduces the memory footprints of shadow memory.

In order to meet **P2**, MEDSALLOC maintains the allocation pools for virtual memory space and always try to map the newly allocated objects to a fresh virtual address so as to fully utilize whole virtual address space to avoid address reuse. Please note that MEDSALLOC *does not* need to be compatible with

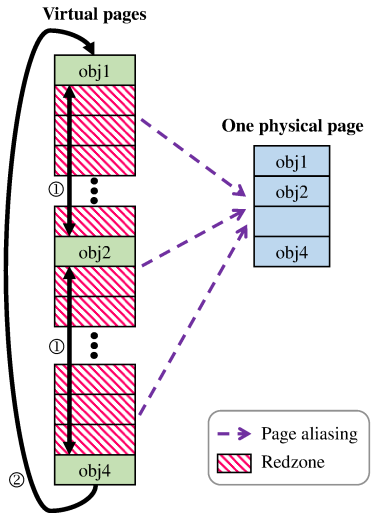

Fig. 2: Aliased memory pages with shadow-memory-based redzone enforcement in MEDS. While each object is sparsely allocated in virtual memory space (left side), its actual memory footprints are tightly packed in physical memory space (right side) using page aliasing. ① illustrates a large redzone between objects (**P1**), which does not actually impose physical memory uses. ② illustrates a size of heap in MEDS, which fully utilizes virtual memory space before begin reusing virtual address (**P2**).

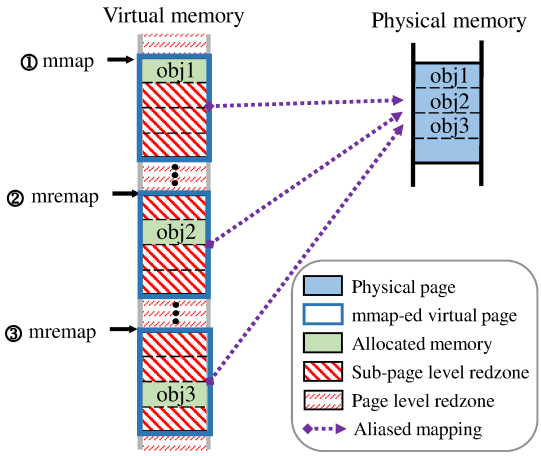

Fig. 3: An example of MEDS's page aliasing scheme with redzone

ASLR because MEDS ensures a stronger security guarantee than ASLR. While MEDS is a memory error detector, ASLR statistically helps *after* a memory error has been exploited. Therefore MEDSALLOC can utilize virtual addresses in a simple sequential manner.

The rest of this section first provides detailed information on page aliasing mechanisms as they are the key enabling features of MEDSALLOC. Then we present more design details on MEDSALLOC, which takes a two-layered scheme using global and local allocators (shown in Figure 4).

Page Aliasing. Page aliasing implicates the intentional aliasing between virtual and physical memory pages such that multiple virtual pages are mapped to the same physical page. Using page aliasing, there can be multiple memory views (through

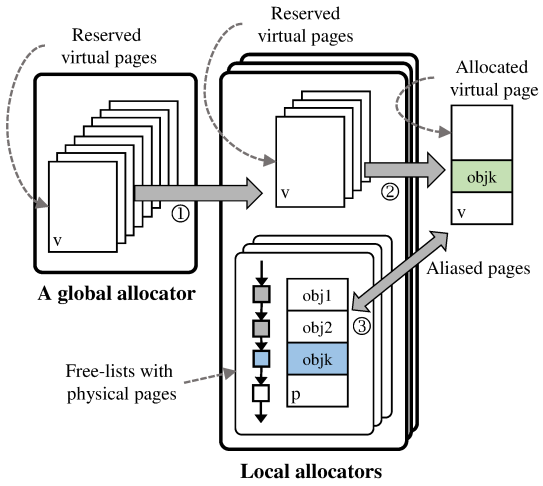

Fig. 4: A work-flow of MEDSALLOC.

multiple virtual pages) to the same memory content (backed by the same physical page). In practice, page aliasing is popularly used to reduce use of physical pages, such as copy-on-write (CoW) and same-page merge [4].

Since this page aliasing mechanism has to rely on virtual memory management, its implementation varies depending on underlying architecture and kernel. In the case of x86-64 architecture (as well as x86) running Linux, page aliasing can be realized through invoking `mmap()` and `mremap()` syscalls. In response to the `mmap()` request, the Linux kernel creates a new virtual memory page that is mapped to a new physical memory page. Here, if the `MAP_SHARED` flag has been specified, the kernel allows the mapping to be shared later (*i.e.*, the new physical memory page can be mapped by multiple virtual memory pages within the same process or with its child processes). We then use `mremap` to create additional aliased virtual pages within the same process' virtual address space. Suppose a mapping between a virtual page `V1` and a physical page `P1` is established by `mmap`. The kernel maps a new virtual page `V2` to the same physical page `P1` *without* removing the old mapping between `V1` and `P1` when `mremap()` is invoked with (1) `old_address` and `new_address` pointing to `V1` and `V2` respectively, (2) `old_size` equal to zero, and (3) `MAP_FIXED` flag being set. Please note this behavior is undocumented in the man page but is described in [35].

Figure 3 shows an example of this aliasing process. If a user process invokes `mmap()` with the `MAP_SHARED` flag set, the kernel creates new virtual pages for the process and returns the base address of such virtual pages (①). After that, when the user invokes `mremap()` where the `old_address` parameter points to the address returned by `mmap()`, the kernel creates an aliased page at `new_address` (②). This aliasing can be repeated as many times as a user process requests, and the kernel returns yet another new aliased virtual page (③).

Global Allocator. To improve the performance of allocators (by reducing use of locks), modern heap allocators feature a global allocator (*i.e.*, per-process allocator) which manages available virtual address space for a running process, and partitions and then distributes virtual address space to local, per-thread allocators (illustrated in Figure 4-①). Here, the key difference between MEDSALLOC and traditional heap allocators' design is that MEDSALLOC never requests actual physical memory from the kernel; instead, it only distributes virtual addresses to the local allocators and takes over the duty of managing available virtual addresses from the kernel. This design choice enables MEDS to meet **P2**. When MEDS looks for an unmapped virtual space, instead of trying to reuse a recently freed virtual space, the global allocator always starts from the last allocation address and follows a monotonic direction so it can fully cycle the whole virtual address space and delay address reuse as late as possible. Again, because as a memory error detector, MEDS provides stronger security guarantee than ASLR, so MEDSALLOC does not need to randomize the allocated virtual addresses.

Local Allocators. Local allocators (*i.e.*, per-thread allocator) maintain virtual memory pages assigned from the global allocator, and maps, or aliases a virtual page to an appropriate physical memory page. That is, each local allocator actually commits physical memory page allocation from kernel. Furthermore, in order to make efficient physical memory uses for small object allocations (*i.e.*, smaller than a page size), each physical memory page is managed with multiple free-lists, which partitions a page into multiple memory slots. We employ a size-class allocation scheme for this free-list, similar to `tcmalloc` [28]—the class is determined by the allocation size, and each class has its own free-list. The difference is that in `tcmalloc`, free-list is used to manage mapped virtual pages (*i.e.*, a virtual-physical page pair); but in MEDSALLOC, the free-list only manages physical pages. A local allocator (1) uses these free-lists to find a physical page with proper and empty memory slot for an allocation, (2) picks up a virtual memory page from a reserved pool, and (3) aliases the virtual page with the physical page.

For example, during the initialization of a local allocator (*i.e.*, right after loading the target application and before executing any target program's code), it reserves a chunk (*e.g.*, 256 MB) of virtual addresses with the help of the global allocator and creates a local virtual page pool (illustrated in Figure 4-①). Next, upon receiving an allocation request from a thread, the local allocator selects an available virtual memory page from the local virtual page pool (illustrated in Figure 4-②). Next, in order to find an available physical page, it scans through one of free-lists corresponds to the allocation size and selects an available physical page and maps it with the available virtual page above (illustrated in Figure 4-③, which allocates `objk`). If the free-list is used for the first time thus no physical page has been associated with it, the local allocator maps a new physical page to the virtual page using `mmap()` syscall with `(MAP_SHARED|MAP_FIXED)` flag. On the other hands, if the free-list has an associated physical memory, it simply reuses that physical page with page aliasing (*i.e.*, `mremap`). After the object allocation, additional virtual pages are allocated from the local virtual page pool to setup a redzone of pre-configured size (*e.g.*, 1 `MB`) ensuring **P1**.

Deallocation. When deallocating an object, MEDSALLOC returns the associated physical memory page back to the free-list. If the physical page is not associated with any active objects (*i.e.*, when all objects using the physical page are freed), the physical page is removed from the free-list. After that, MEDSALLOC simply unmaps the object region, and the

```
1  def enforce_redzone_on_alloc(addr, size):
2    # Compute the start and end address of object (page-aligned)
3    start = start_pageaddr(addr)
4    end = end_pageaddr(addr + size)
5
6    # Maps the corresponding shadow memory.
7    # Shadow memory bits are initialized as INVALID.
8    map_shadow_memory(start, end)
9
10   # Set shadow memory bits (VALID) for an object region.
11   # The rest shadow memory bits are left as INVALID.
12   set_shadow_memory(addr, size, VALID)
13   return
```

(a) Redzone management after allocation

```
1  def enforce_redzone_on_dealloc(addr, size):
2    # Compute the start and end address of object (page-aligned)
3    start = start_pageaddr(addr)
4    end = end_pageaddr(addr + size)
5
6    # Mark memory space to be deallocated inaccessible.
7    mprotect(start, end-start, PROT_NONE)
8
9    # Unmaps the corresponding shadow memory.
10   unmap_shadow_memory(start, end)
11   return
```

(b) Redzone management after deallocation

Fig. 5: Pseudo-code algorithms on redzone management (per-byte granularity) of MEDS.

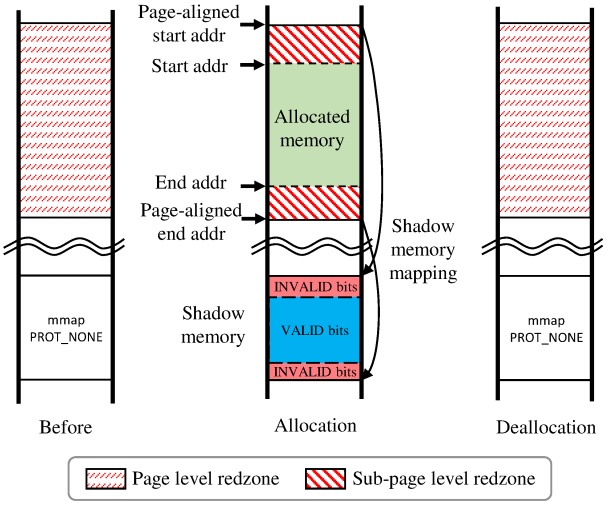

Fig. 6: Redzone management on memory (de)allocation

physical page will be automatically returned to kernel because there is no virtual pages associated with the physical page.

Optimization. MEDS employs an optimization scheme in allocating objects larger than a page size (*i.e.*, 4 KB, or 2 MB if a huge page is used). In particular, since there is almost no advantage of performing page aliasing—the physical page will be fully occupied for such objects thus no space is left for aliasing, we directly allocate these objects from physical pages without searching through a free-list.

C. Redzone Management and Enforcement

We need additional mechanisms, which we call redzone management and enforcement, to detect spatial memory error. The MEDSALLOC itself does not provide access control. For example, in Figure 4, using an address pointing to the `objk`, other objects including `obj1` and `obj2` can also be accessible. To capture such an offending access, one may simply adopt shadow-memory-based redzone enforcement exercised in ASAN. However, since MEDS utilizes much larger redzones than ASAN, the shadow memory usage for maintaining the redzones would lead to high memory consumption. Because of the problem, the simple redzone scheme is not practical for MEDS. Therefore, MEDS employs two different redzone management schemes, page level redzone and sub-page level redzone, where only the sub-page level redzone is actually represented with shadow-memory. In the following, we first describe how MEDS manages these two different redzones and then explain how MEDS enforces redzones (i.e., detecting memory accesses to redzones).

Managing Redzone. In order to manage redzones, MEDS basically intercepts all allocation and deallocation functions invoked by the target application in runtime and updates the shadow memory. A particular challenge here is the memory consumption, if such redzones are all represented using shadow memory. That is, MEDS by design produces very large redzones between memory objects. If MEDS commits dedicated shadow memory for entire redzones, then shadow memory itself will occupy a lot of physical memory.

In order to address this challenge, we first categorize redzones into two different types, a page level redzone (*i.e.*, a gap between virtual pages) and a sub-page level redzone (*i.e.*, gaps within a page). Then we leverage the observation that one shadow memory page governs exactly 32 KB memory (*i.e.*, 8 virtual pages), which means our page level redzones (4 MB) consume shadow memory in the granularity of 128 pages. Since every byte in page level redzones are inaccessible, the corresponding shadow memory page will be filled with `1`. However, we do not need to allocate individual shadow pages for page level redzones—we can simply leave those shadow pages as unmapped so checking against those shadow pages will always trigger a page fault that will be captured by our signal handler.

Based on the observation above, MEDS only maintains sub-page level redzones in shadow memory, and page level redzones do not impose any physical memory use. Figure 5 shows pseudo algorithms on how MEDS manages the shadow memory, where a snapshot of virtual memory changes are illustrated in Figure 6. At the time of the object allocation (Figure 5-(a)), because the allocated virtual page is used to be a page level redzone (*i.e.*, its corresponding shadow memory is unmapped), MEDS first `mmap` new physical page(s) from the kernel for its shadow memory page(s) and initializes the entire page(s) as `INVALID` (`line 8`). Then MEDS sets the corresponding shadow memory (*i.e.*, a range of object addresses to be allocated, from `addr` to `addr+size`) as `VALID` (`line 12`).

At the time of object deallocation (Figure 5-(b)), MEDS first marks the virtual page(s) which is being deallocated inaccessible using `mprotect` (`line 7`). This permission setting on page level redzones is always possible in MEDS, because all virtual pages are always exclusively associated with a single object in the virtual memory space. Moreover, instead of explicitly marking the corresponding shadow memory space as `INVALID`, MEDS simply unmaps the shadow memory space to mark them as a redzone (`line 10`). Again, any attempt to access this freed memory region will be anyway detected through page fault,

```
1 // Before instrumentation
2 value = *load_addr;
3
4 // After instrumentation
5 if (!check_shadow_memory(load_addr))
6     report_and_terminate();
7 value = *load_addr; // Safe to load.
```

(a) Load instrumentation

```
1 // Before instrumentation
2 *store_addr = value;
3
4 // After instrumentation
5 if (!check_shadow_memory(store_addr))
6     report_and_terminate();
7 *store_addr = value; // Safe to store.
```

(b) Store instrumentation

Fig. 7: Redzone enforcement (per-byte granularity) using memory access instrumentation on load and store instructions.

because the associated shadow memory is not accessible.

Enforcing Redzone. MEDS ensures that all memory accesses are valid by enforcing redzones. The security guarantee that all memory accesses are properly safeguarded is made by the fact that any access attempt touching redzones is detected because (1) MEDS explicitly checks shadow memory (for sub-page level redzones) or (2) MEDS implicitly captures page fault events (for page level redzones). More specifically, MEDS instruments all memory access instructions, including load and write, such that the access is only permitted after checking the validity through the shadow memory.

Figure 7 illustrates how MEDS instruments load and store instructions. For load instructions (Figure 7-(a)), MEDS first checks the shadow memory for a given address to be accessed (`line 5`). If the given address points to page level redzones, MEDS will capture page faults while loading the corresponding shadow memory bit, because such shadow memory space is not accessible. If the shadow memory bit is properly loaded but indicates `INVALID` (*i.e.*, sub-page level redzones), MEDS does not permit the original load instruction being executed (`line 6`). For both of these violation attempts, either through capturing page fault events or detecting the `INVALID` shadow memory bit, MEDS reports a detailed information on violation such that developers or users can easily understand the cause of access violation. If the bit indicates `VALID`, MEDS allows to perform the original load operation (`line 7`) so that the program execution semantics are kept intact for benign load operations. Store instructions are handled in a similar way as for the load.

Optimization. Similar to ASAN, for memory intrinsic functions like `memset()` and `memcpy()`, instead of checking the safety of all repeated load/store instructions within these memory intrinsic functions, MEDS checks its safety using its parameters. However, due to its small redzone size, after checking the start, end, and mid of the buffer, if all checks succeed, ASAN still needs to check the shadow values for all the bytes of the buffer. However, because MEDS uses a much larger gap between objects, we only need to check the start, end, and well-aligned bytes (*e.g.*, 4 `MB`-aligned, which is the current default redzone size of MEDS).

D. Memory Object Allocation

MEDS allocates all memory objects using MEDSALLOC (§IV-B), such that all memory objects are surrounded by approximated infinite gap and its allocation pool follows the concept of approximated infinite heap. Generally there can be three different types of memory objects depending on where an object is allocated—heap, stack, and global objects. As each object type goes through a different allocation mechanism, MEDS properly caters its allocation process per allocation type so that all memory objects are allocated using MEDSALLOC.

Heap Objects. Heap objects are allocated through a limited set of runtime functions (*e.g.*, `malloc`, `calloc`, etc.[1]). Similar to ASAN, we install interception hooks to these functions so that MEDS can take control over allocation processes. Then, upon receiving a heap allocation request from a user program, MEDS simply leverages MEDSALLOC to return an aliased memory object.

Stack Objects. Stack objects are allocated within a corresponding function's stack frame. Unlike heap objects, MEDS takes different approaches in handling stack objects depending on whether they are allocated implicitly or explicitly. For the implicitly allocated stack objects such as return addresses and spilled registers, since accesses to them are always safe, MEDS does not need to protect them (*a.k.a.*, safe stack [19]) with redzones. On the other hand, MEDS migrates the explicitly allocated stack objects (i.e., stack variables) into heap space using MEDSALLOC so as to easily leverage the features of MEDSALLOC for safeguarding them with redzones. For each stack object in a function, MEDS instruments a runtime function call `alloc_stack_obj(size, alignment)` at the corresponding function's prologue. This function performs dynamic heap allocation using MEDSALLOC for a given size while observing the alignment constraint for the allocated object. MEDS also instruments another runtime function call `free_stack_obj(ptr)` at the function's epilogue, which properly frees a stack object (located in heap space under MEDS) right before the function returns. MEDS also registers this free runtime function call to exception handling chains so that the stack objects can be freed when a stack unwinding happens due to an exception. In this way, MEDS places all the stack variables in heap, where its allocation is always performed by MEDSALLOC.

Global Objects. Unlike stack and heap objects, the addresses for global objects are located at the time of loading a program. More precisely, in the case of an ELF executable, the ELF loader maps virtual memory pages as specified in a program header section of the ELF format.

A straightforward design decision leveraging MEDSALLOC would be simply creating aliased memory pages for each global object, while considering mapped data pages by the loader as packed physical memory pages. However, we found this is not feasible without compromising compatibility. Because the built-in ELF loader implemented in the Linux kernel always assigns `MAP_PRIVATE` (instead of assigning `MAP_SHARED`) when mapping data memory pages, those memory pages cannot be aliased. We may workaround this issue through either 1) using a

[1]Handling `malloc()` typically covers cases using C++ allocation operators (*i.e.*, `new`), as generated code for a `new` operator eventually invokes `malloc` to allocate memory space.

user-level custom ELF loader instead of using the built-in ELF loader or 2) simply modifying the built-in ELF loader to specify `MAP_SHARED`. Both of these workaround approaches may have negative impacts on compatibility. It might be cumbersome to setup an execution environment using a custom loader for end-users, or it is generally discouraged to modify the underlying kernel.

Therefore, in order to preserve compatibility, MEDS implements user-level re-allocation schemes for global objects. After loading a target program but before executing the original entry point of the program, MEDS enumerates a list of global objects and re-allocates each of them using MEDSALLOC. If a global object requires initialization (i.e., data with non-zero bytes), MEDS accordingly copies those underlying data as well. Since now the locations of global objects have been migrated to heap space, MEDS relocates all the references (which pointed to original global objects) to reallocated ones in heap space by referring a relocation table in ELF. While this scheme for global objects may seem performance expensive, we highlight that this procedural only needs to be performed once and thus it only adds an one-time fixed cost to the program loading procedural.

E. User-level Copy-on-Write (CoW)

As noted in §IV-B, MEDS uses `mmap()` syscall with a `MAP_SHARED` flag to alias a physical page with multiple virtual pages, but this approach has compatibility issue with the CoW scheme of the Linux kernel. More specifically, when the `fork()` syscall is invoked, the child process shares the same physical pages with its parent process. Then, the kernel CoW to perform lazy copy and unshare the modified physical pages. However, pages mapped with the `MAP_SHARED` are not subjected for CoW, as the kernel interprets that such pages should be shared between a parent and child process. As a result, for process protected by MEDS, `fork()` will break the normal isolation guarantees between processes.

To address this issue, MEDS designs a user-level copy-on-write mechanism. First, MEDS intercepts all `fork`-like syscalls. Right before `fork()`, MEDS marks all virtual pages allocated by MEDS with `MAP_SHARED` as non-writable. After `fork()`, when a process attempts to write any of such pages, a pre-installed signal handler catches the attempt via page fault. Then MEDS allocates a new physical page, maps it to a temporary virtual address (with `MAP_SHARED`), hard-copies the content from the old physical page, unmaps the old physical page, remaps a new physical page to the old virtual address, unmaps the new physical page from the temporary address, and then passes control back to the process to continue the write operation. This mechanism can also be implemented at the kernel-level by adding a dedicated flag for page aliasing, but we chose to implement user-level solutions to avoid installing a kernel extension for better compatibility.

V. IMPLEMENTATION

We have implemented a prototype of MEDS based on the LLVM Compiler project (version 4.0). MEDS is implemented in total 10,812 lines of `C` and `C++` code. Overall, MEDS takes `C` or `C++` source code of a target application as input, and generates executables. The instrumentation module is implemented as an extra LLVM pass. The runtime library module is based on sanitizer routines in LLVM. All standard allocation and deallocation functions are hooked to be delegated by MEDSALLOC. Copy-on-write (COW) is implemented by hooking `fork()` and installing a custom signal handler to capture invalid memory access attempts.

VI. EVALUATION

In this section, we evaluate our prototype of MEDS against our objectives (§III) by answering the following questions:

- **Compatibility.** Does MEDS introduce compatibility issue to our target large-scale production programs? (§VI-A)
- **Detectability.** Does MEDS properly detect memory errors against attacking exploits (§VI-B) and in fuzz testing (§VI-C)?
- **Performance.** How much performance overhead does MEDS impose? (§VI-D)

Experimental Configuration. MEDS is configured to have 4 `MB` of the redzone and 80 `TB` of the quarantine zone. ASAN is configured to have default parameters—from 16 `Byte` to 2,048 `Byte` to the redzone[2], and 256 `MB` of the quarantine zone. As noted before, enlarging ASAN's parameters ends up having out-of-memory issues due to heavy physical memory uses.

Experimental Setup. All our evaluations were performed on Intel(R) Xeon(R) CPU E5-4655 v4 @ 2.50GHz (30MB cache) with 512GB RAM. We ran Ubuntu 16.04 with Linux 4.4.0 64-bits. We have used MEDS to build the following five applications for evaluation: Chrome browser (58.0.2992.0), Firefox browser (53.0a1), Apache web server (2.4.25), Nginx web server (1.11.8), and OpenSSL library (1.0.1f).

A. Compatibility

One of the key goals of MEDS is to maintain compatibility in running target applications, especially for the large-scale commodity programs. In order to check such compatibility, we ran the basic functionality unit tests provided by respected vendors: 2,242 test cases in Chrome, 781 test cases in Firefox, and 1,772 test cases in Nginx. MEDS passed all of these unittests, implying that MEDS truly meets compatibility requirements for complex programs.

B. Detectability against Attacking Exploits

Recall that MEDS detects memory errors by approximating the concept of the infinite gap and heap. In this subsection, we first test the detection capability of MEDS on a set of simple unit tests that cause memory corruptions. Then we use real-world vulnerabilities to see MEDS's detection capability in practical use-cases. Lastly, we show various measures showing the effectiveness of MEDS's approximation on the infinite gap and heap.

Memory Error Unit Tests. To see whether MEDS can detect all different kinds of memory errors, we ran a set of unit

[2] ASAN takes the minimum and maximum size of redzone. Then, depending on the allocation size, ASAN picks the redzone size within this minimum and maximum range.

tests available in the LLVM ASAN. It has 50 unit test cases, including stack overflows, heap overflows, use-after-free, etc. In addition to these cases, in order to better compare MEDS against ASAN and demonstrate MEDS's limitation as well, we also added the following four tests: two heap overflow cases accessing beyond the redzone of either ASAN or MEDS (4 `MB`), respectively; and two heap use-after-free cases which allocate either less or more than the quarantine zone size, respectively. In all of these tests, a simple vulnerable program is run with a specific input triggering a memory error, and the test passes if the program properly stops and reports an error. Overall, MEDS was able to pass most of these tests except one, showing that MEDS does handle all different memory error cases. This exception case, as expected, was in heap overflows accessing beyond MEDS's redzone size (4 `MB`). As for ASAN, it failed to detect three cases due to its small redzone and quarantine zone size. According to these unit test results, the detection capability of MEDS is arguably a super set of ASAN.

Juliet Test Suite. NIST provides the Juliet test suite [6], which is developed for testing the effectiveness of software assurance tools. Every test case has two versions, a call to a bad function with a vulnerability (so as to measure false negatives) and a call to a good function that have patched the vulnerability (so as to measure false positives). We particularly focused our testing on memory corruption related testcases in Juliet, a total of 11,414 testcases: 3,124 tests for Stack buffer overflow (`CWE` 121), 3,870 tests for Heap buffer overflow (`CWE` 122), 1,168 tests for buffer underwrite (`CWE` 124), 870 tests for buffer overread (`CWE` 126), 1,168 tests for buffer underread(`CWE` 127), 820 tests for double free (`CWE` 415), and 394 tests use-after-free (`CWE` 416). We compiled all of these testcases using MEDS as well as ASAN, and measured false positives and false negatives in terms of detectability. In most of time, both MEDS and ASAN showed 0 false positives and false negatives. However, sometimes both showed false negatives on the testcases, ranging from zero to 288. We analyzed the details of those false negative cases and found that these test cases involve random memory access (*i.e.*, an access address is computed through `rand` function seeded by time). In other words, these random accesses may skip over the redzone size enforced by both schemes, resulting in false negatives.

In order to better compare the detection capability and understand its practical implications with respect to this random access, we modified those 288 cases with the following constraints: (1) allocating a thousand more objects (currently Juliet tests only allocate one object for each test) and (2) limit the random access within the range of stack/heap segments. The first constraint takes account of practical running environments, where most real-world programs allocate a huge number of memory objects at runtime. The second constraint considers general programming practice — in practice a pointer value is mostly deduced from an address of existing objects. We applied these changes to 288 testcases, and ran a million times to measure the detection probability. Our result shows that MEDS detected 98% of those while ASAN detected 35% of those. Although this modification on Juliet test is arguably in favor of MEDS, we still believe this outstanding detection probability of MEDS demonstrates enough its significant improvement in detection capability over ASAN.

Detecting Real-world Memory Errors. To better understand whether MEDS can truly detect memory errors in realistic use-cases, we launched memory corruption attacks against a set of vulnerabilities in popular applications including Chrome and Firefox. For each vulnerability, we first rolled back the target application's source code to the vulnerable version, and then used both ASAN and MEDS to build the application to compare the detection capability.

App.	CVE	Types	Detection	
			ASAN	MEDS
Chrome	2016-1653	S, W	△	✓
Chrome	2016-5182	S, W	△	▲
Chrome	2016-5184	T, RW	△	▲
Firefox	2016-2798	S, R	△	▲

TABLE I: Detection capability of MEDS on real-world vulnerabilities: **S** - spatial memory errors; **T** - temporal memory errors; **W** - a write violation; **R** - a read violation; ✓ - detected; ▲ - partially detected (difficult to bypass); △ - partially detected (easy to bypass)

As shown in Table I, MEDS enhanced its detection capability in Chrome and Firefox over ASAN, both in spatial and temporal memory errors. In fact, the table demonstrates not only the effectiveness of MEDS's approximated infinite gap and heap but also its limitation. In the case of CVE-2016-1653, since the vulnerability offers a limited range of violation access less than 4 `MB` (*i.e.*, less than MEDS's redzone size), MEDS was able to fully detect it whereas ASAN was not. However, for the rest three cases, because these offer full control over the pointer (*i.e.*, complete arbitrary memory read or write vulnerability), MEDS was also bypassed as ASAN did. We still note that bypassing MEDS is more difficult than ASAN, thus those are marked as ▲ in MEDS and △ in ASAN, respectively.

Effectiveness of Approximation. MEDS elevates the level of detection capability by approximating the infinite gap and heap, but apparently there should be a certain upper bound due to the limited memory resources. Thus, we study practical impacts of those limits in terms of detection capability. In particular, an offset size of memory accesses directly impacts the effectiveness of redzone based detection. In fact, this offset size is strongly related to the allocated object size, because intermediate pointer arithmetic only involves in shifting a pointer within the same object. Therefore, the possible difference caused by pointer arithmetic is mostly smaller than the associated object size. Thus, we measured the size of each allocation across all the applications under our evaluation, and found that all objects were smaller than 4 `MB`, which implicates 4 `MB` redzone would provide reasonably good detection capability. This measurement also shows the limitation of ASAN, as 11% of objects were bigger than 256 `Bytes` (*i.e.*, the default redzone size of ASAN) and accessing routines onto those 11% of objects may be abused to bypass a redzone size. As noted before in §IV, enlarging this parameter in ASAN is not suitable for large-scale applications due to out-of-memory issues. It is worth noting that MEDS can be further augmented through strictly restricting the pointer arithmetic up to the maximum object size (*i.e.*, 4 `MB` in these applications). This will enable MEDS to truly achieve the infinite gap.

MEDS also approximates the infinite heap by cycling

App.	Recycle frequency (min)		
	H	*HS*	*HSG*
Chrome	50.3	15.2	15.1
Firefox	160.7	32.1	32.0
Apache	141.2	95.7	95.7
Nginx	4.5	0.5	0.5

TABLE II: The frequency of virtual address recycling on MEDS: *H* - aliasing heap objects; *HS* - aliasing heap and stack objects; *HSG* - aliasing all objects including heap, stack, and global. Note that first virtual address reuse on ASAN is done very quickly at initialization in case of Chrome and Firefox.

	Buffer overflow			Use-after-free		
	ASAN	MEDS	Improv	ASAN	MEDS	Improv
First crash time (s)	631.85	51.64	12.23x	985.10	86.02	11.45x
Crashes per an hour	8.76	27.74	3.16x	7.48	20.85	2.78x

TABLE III: Detection performance of MEDS and ASAN with micro-benchmarks.

through 64-bits of virtual memory space. More precisely, MEDS alone cannot fully use such 64-bits space, but it currently utilizes 80 TB virtual memory space— given the total of 47-bits user-land virtual address space in x86 (total 128 TB), it reserves 16 TB for shadow memory, another 16 TB is reserved for internal memory allocations for MEDS, and yet another 16 TB is reserved for Linux stack. Therefore, since MEDS starts reusing virtual memory space after allocating objects over 80 TB, we try to project a time taken to trigger this action from the end-user perspective. Specifically, we ran MEDS applied versions of Chrome and Firefox, which visited websites every 5 minutes using the same tab; and ran those of Apache and Nginx which serves 25,000 requests (with concurrency level 50) per a second. According to our running results (Table II), Chrome, Firefox, and Apache started to reuse the address space after 49 minutes, 160 minutes, and 141 minutes, respectively. We believe this is a reasonably long enough time, not interfering end-user experiences, especially when considering that most users would frequently close and create new tabs. Nginx quickly drained the virtual address space though—it only took 4 minutes until the recycle. We suspect this is because Nginx is designed to repeat heavy memory reallocations. Although this would not be an ideal, in this case the Nginx process can be re-spawned frequently before reaching this virtual address recycling time.

C. Detectability in Fuzz Testing

In order to demonstrate MEDS's effectiveness in detecting memory errors while performing fuzz testing, we run a fuzz testing using both micro-benchmarks and real-world applications. We used American Fuzzy Lop (AFL) as a fuzzing framework [38], which is one of the most popular fuzzers in practice. Target programs were first instrumented using AFL to enable its feedback based fuzzing functionality, and further instrumented with either MEDS and ASAN to compare the detection capability.

Fuzzing Micro-benchmark Programs. In this evaluation, we developed and tested two simple yet realistic vulnerable programs, exhibiting buffer overflow or use-after-free vulnerabilities, respectively. These testing programs were written to highlight the effectiveness of MEDS, especially in terms of non-linear memory violation cases (i.e., beyond the size of a redzone) and temporal violation cases with heavy memory allocations (i.e., beyond the size of a quarantine zone). Thus, these may not represent general detection capability of all memory error cases. However, since these vulnerable codes were taken and simplified from real-world vulnerabilities, we believe this testing still has practical implications in terms of memory error detection, which we will further showcase using real-world applications.

The first case on buffer overflow vulnerability is caused by an integer overflow on allocation size. It takes width and height of the canvas and allocates the canvas. After that, the program takes offset, size, and data to write on the canvas. There is an integer overflow when computing the canvas size. The second case has a use-after-free vulnerability. Initially, it has a set of pointers, each of which points to a heap object. Then the program takes an integer value k, which frees k number of objects. After freeing the objects, it allocates new objects more than freed objects, and attempts to access one of the pointers that were pointing to heap.

We have performed 10 times for the micro-benchmarks and Table III shows the average times taken to encounter the first crash, and average crashes per an hour. MEDS encounters the first crash 12 times earlier than ASAN in our micro-benchmark. When running the input for the first crash of MEDS with ASAN, ASAN usually cannot detect the vulnerability. Also, MEDS has 3 times higher average crashes than ASAN during fuzzing. The result shows MEDS can find a target vulnerability faster than ASAN. In other words, MEDS is effective in terms of detecting performance.

Fuzzing Real-world Programs. To clearly demonstrate practical aspects of MEDS in augmenting fuzz testing capability, we also run AFL using real-world programs. Table IV shows the results while fuzzing each program for six hours. We collected a set of target applications from GitHub and the Debian repository, where its popularity is implicated by either the popularity pair (the number of forks and the number of stars in GitHub) and the installation ranking (among 26,762 applications in Debian repositories), respectively. The applications are all recent versions so that bugs found from this test are all new bugs, and we are already contacting the corresponding development community to report these issues. The complexity of applications are represented in terms of the lines of code (LoC). The total number of executions denotes the number of executed instances during six hours of the fuzz testing. Since the same memory error can be triggered through many different inputs, AFL only keeps the crash exhibiting unique execution paths, which is called a unique crash.

Overall MEDS outperformed ASAN in augmenting memory error detection capability of fuzzing for all target applications we run, in terms of the total number of unique crashes—68.3% improvements on average, ranging from 1% to 256%. In fact, these results are particularly interesting because MEDS is no better than ASAN in terms of execution speeds (although sometimes MEDS is faster than ASAN), as it is highly depending on application's runtime characteristics (i.e., memory allocation

App.	Description	Popularity		Complexity	Total execs (K)		Total unique crashes			Unique crahses per 1M execs		
		GitHub$^{\alpha}$	Debian$^{\beta}$	(LoC)	ASAN	MEDS	ASAN	MEDS	Improv	ASAN	MEDS	Improv
PH7	PHP interpreter	(35, 321)	-	43K	387	360	8	29	256%	20.67	79.17	283%
lci	LCODE interpreter	(61, 355)	-	50K	413	820	21	54	157%	50.85	65.85	30%
picoc	C interpreter	(161, 1240)	-	68K	4,535	6,020	108	231	114%	23.81	38.37	61%
ImageMagick	Image tool	(212, 933)	-	622K	110	58	9	14	56%	81.82	241.38	195%
wren	Sciprt language	(190, 1991)	-	13K	340	222	92	110	20%	270.59	495.50	83%
espruino	JS interpreter	(359, 1157)	-	18K	167	143	260	295	13%	1,556.89	2,062.94	33%
tinyvm	Tiny virtual macine	(123, 1154)	-	30K	182	170	73	80	10%	399.17	468.69	17%
raptor	RDF format parser	-	699	162K	1,320	1,250	2	5	150%	1.52	4.00	164%
swftools	Tools for SWF files	-	6,476	158K	44	50	97	123	27%	2,204.55	2,460.00	12%
exifprobe	Probe EXIF files	-	6,512	45K	573	666	135	150	11%	0.24	0.23	-4%
metacam	Probe EXIF files	-	8,355	4K	504	457	58	61	5%	115.03	133.48	16%
jhead	Image tool	-	4,010	10K	1,490	2,470	85	86	1%	57.04	34.82	-39%

TABLE IV: Fuzzing real-world applications to compare memory error detection capability of ASAN and MEDS. α denotes (the number of forks, the number of stars) in GitHub, and β denotes the installation ranking from the Debian popularity contest. Each application was fuzzed for 6 hours using the AFL fuzzer [38].

behavior). This execution speed can be deduced from the total number of executions. For example, in the case of PH7, MEDS was slightly slower than ASAN (i.e., 7 % slower). Seven applications were slower when running with MEDS, however, more unique crashes occur during the fuzz testing. Five applications (i.e., lci, picoc, swftools, exifprobe, and jhead) were faster when running with MEDS. Among these, in terms of unique crashes per executions MEDS was slower in two applications (i.e., exifprobe and jhead). We suspect this is because MEDS reached to the point earlier than ASAN, where AFL gets saturated in exploring more execution paths in these two applications. After being saturated, MEDS spent the rest fuzzing cycles, more cycles than ASAN as MEDS has faster execution speeds, without finding new unique crashes. In other words, MEDS found most of unique crashes faster than ASAN, but spent the rest fuzzing time without finding more as AFL gets saturated. For the rest of three applications (i.e., lci, picoc, and swftools), they have higher unique crashes per executions when running with MEDS.

Even for the seven applications that showed slower execution speeds in MEDS (i.e., PH7, ImageMagick, wren, espruino, tinyvm, raptor, and metacam), MEDS still was able to find more unique crashes than ASAN. This implicates that, the advantages in providing enhanced detection capability outweighs the disadvantages in slowing down the execution speed, resulting in overall fuzzing performance improvements made by MEDS (in terms of finding more unique crashes).

We believe this clearly demonstrates the improved memory error detection capability of MEDS over ASAN. Considering the huge popularity of AFL and ASAN in performing real-world fuzz testing, these results also signify the strong practical impacts of MEDS— when used together with AFL, the prototype of MEDS can help the fuzz testing processes, significantly better than the state-of-the-art memory error detection tool, ASAN.

D. Performance Overheads

The security service of MEDS obviously comes with cost, which mainly impacts two performance factors: runtime speed and physical memory usage.

Runtime Speed. The major factors imposing runtime speed overheads for MEDS are (1) it executes extra instructions to check all memory load and store instructions; (2) since MEDS utilizes more virtual address space, there will be more TLB misses; and (3) each object allocation needs to invoke `mremap` syscalls for page aliasing.

To better understand these aspects, we ran benchmarks for applications — Table V shows the running results of Chrome, Firefox, Apache, and Nginx, and Table VI shows that of OpenSSL. For Chrome and Firefox, we used Octane benchmarks [14]; for Apache and Nginx, we used Apache benchmark [13] which serves 25,000 requests per second; and for OpenSSL, we used OpenSSL's speed command to encrypt memory blocks using SHA1 [12]. For each run, we applied three different settings of MEDS to better understand performance impacts from object coverage. In other words, the MEDS column with *H* denotes that MEDS safeguards heap objects (*i.e.*, all heap objects have been allocated using MEDSALLOC). Similarly, *HS* denotes for both heap and stack objects, and *HSG* denotes for all object types including heap, stack, and global.

On average, MEDS slowed down the execution about 27% on MEDS-H, 94% on MEDS-HS, and 108% on MEDS-HSG compared to the baseline. First, as MEDS increases the object coverage (from heap object types to all object types), the execution gradually slowed down because MEDS will miss more TLB and invoke more system calls. This performance change is especially noticeable between MEDS-H and MEDS-HS for Nginx (i.e., 24% to 250%). This is because Nginx allocates a huge number of stack objects at runtime, which in turn incurs a huge number of allocations (when a function is invoked) and deallocation (when a function returns) for MEDS. The stack object allocation is not a performance bottleneck for the baseline, however, as it only requires to shift the stack pointer to reserve and release stack memory space for objects.

Compared to ASAN, MEDS slowed down the execution about 11%, 73%, and 86%. As MEDS does not impose significant overheads in terms of instrumented instructions compared to ASAN (*i.e.*, both check the shadow memory bit), we inspected other performance factors—TLB misses (Table VII) and the number of invoked system calls (Table VIII). Overall MEDS indeed incurs much more TLB misses (*i.e.*, on average 499% more than ASAN) and invokes much more system calls (*i.e.*, on average 32 times more than ASAN). However, we believe

App.	Benchmark (Metric)	Baseline Performance	Performance (Slowdown)			
			ASAN	MEDS		
				H	*HS*	*HSG*
Chrome	Octane	28,177	24,117 (17%)	21,553 (31%)	20,713 (36%)	19,525 (44%)
Firefox	(Score, high)	26,970	23,076 (22%)	20,043 (35%)	N/A	N/A
Apache	ApachBench	5,671	5,087 (11%)	4,826 (18%)	4,540 (25%)	4,327 (31%)
Nginx	(# of requests, high)	8,132	7,370 (10%)	6,538 (24%)	2,528 (222%)	2,364 (250%)

TABLE V: Runtime performance (a score for Octane, and # of requests per second for ApacheBench; the higher the better) overheads of MEDS, along with ASAN and the baseline for comparison. Overall, on average MEDS slows down an execution 108% compared to the baseline, and 86% to ASAN.

Block Size (Bytes)	Baseline Performance	Slowdown			
		ASAN	MEDS		
			H	*HS*	*HSG*
16	72K	248%	624%	672%	759%
64	201K	276%	357%	424%	483%
256	434K	150%	169%	209%	238%
1024	635K	60%	95%	102%	118%
8192	746K	12%	19%	22%	25%

TABLE VI: OpenSSL performance (# of KB processed per a second) of MEDS, along with the baseline and ASAN. Larger block size decreases the overhead of both ASAN and MEDS. Especially, the block size is critical to the performance of MEDS.

Application	# of TLB misses			Overhead	
	Baseline	ASAN	MEDS	ASAN	MEDS
Chrome	27,821k	40,185k	119,855k	44%	331%
Firefox	42,548k	44,711k	121,062k	5%	185%
Apache	3,391k	3,640k	3,741k	10%	10%
Nginx	452k	542k	1,150k	20%	154%
OpenSSL	945k	1,126k	4,468k	19%	372%

TABLE VII: TLB utilization while running benchmarks

it is a reasonable cost of MEDS's enhanced security services, in terms of detectability.

Physical Memory. MEDS imposes more physical memory uses because it keeps extra metadata for shadow memory as well as page-alias mapping information. As shown in Table IX, MEDS-H, HS, and HSG imposed 133%, 200%, and 212% more physical memory uses on average than the baseline, respectively. In particular, while MEDS imposed 432% in OpenSSL, and 301% in Apache, it imposed 109% on average for the rest of four applications. This is because all memory allocations in OpenSSL were small sized allocations (*i.e.*, from 8 to 32 `Byte`) and MEDS appends 8 `Byte` of per-object metadata to keep aliasing information. Also, OpenSSL does not deallocate memory objects during the evaluation. Thus, corresponding shadow memory pages are mapped to physical memory pages. This runtime characteristic was also captured in TLB misses in Table VII—OpenSSL incurred the highest TLB misses as it intensely stretches virtual address space. On the contrary, ASAN imposed 95% more on average than the base line, because ASAN actually commits physical memory space for the redzone as well as the quarantine zone. We believe this demonstrates the effectiveness of our page aliasing mechanism in that MEDS does not impose impractical physical memory uses while utilizing

App.	# of system calls			Overhead	
	Baseline	ASAN	MEDS	ASAN	MEDS
Chrome	82,313	100,893	1,788,698	1.23x	21.73x
Firefox	213,388	227,352	1,805,388	1.06x	8.46x
Apache	548,684	949,493	1,213,057	1.73x	2.21x
Nginx	275,311	280,279	535,144	1.02x	1.94x
OpenSSL	127	7,184	975,416	57.57x	7,680.44x

TABLE VIII: The number of system calls invoked while running benchmarks

App.	Baseline Memory Usage	Overhead			
		ASAN	MEDS		
			H	*HS*	*HSG*
Chrome	733MB	111%	83%	110%	116%
Firefox	725MB	128%	95%	130%	136%
Apache	217KB	82%	44%	58%	74%
Nginx	195KB	98%	78%	292%	301%
OpenSSL	339KB	57%	367%	409%	432%

TABLE IX: Physical memory uses of MEDS, along with the baseline and ASAN for comparison. Overall, on average MEDS uses physical memory 218% more than the baseline, and it uses 68% more than ASAN.

immense virtual address space.

VII. DISCUSSION

Potential Use-cases. Throughout this paper, we tried to neutralize use-cases of MEDS as we believe MEDS's contribution is general in enhancing memory error detection capability. One specific use-cases would be in deploying MEDS for mitigating memory corruption attacks for large scale applications. Since MEDS indeed meets compatibility requirements (as it can run large scale programs including Chrome, Firefox, Nginx, and Apache) and enhances detection capability compared to other detection tools, it would be a good fit for these cases especially focusing on detection itself. However, the performance overheads that MEDS introduces can be an issue, so it may not be suitable for performance critical applications.

The other use-cases of MEDS would be augmenting the fuzz testing. As we have shown in §VI-C, MEDS significantly outperforms the state-of-the-art memory error detection tool, ASAN. Recognizing the importance of fuzz testing, the vast majority of vendors today employ fuzz testing in their regular software development cycles with huge computing resources. For example, Google reported that they are dedicating a cluster

of hundreds of virtual machines for fuzzing, which runs around 6,000 Chrome instances simultaneously. Because MEDS is capable of finding more memory errors than ASAN given the same computing time, we believe MEDS will be useful not only saving computing resources for fuzzing, but also notifying the memory error bugs earlier in their development cycles.

Kernel-Level Support for Performance Improvements. This paper focused on keeping the compatibility of MEDS, particularly without introducing new features in an underlying operating system, Linux. As shortly mentioned before, the performance of MEDS can be further improved if it can leverage a few kernel changes in the future. For example, when implementing a user-level copy-on-write (COW) (§IV-E), the kernel can be modified to maintain a special flag for page aliasing. This would require to add a few additional flags in `mremap()` system calls. By doing this, MEDS does not need to implement relatively expensive user-level COW mechanisms, reducing runtime overheads of MEDS. As another example, MEDS has to re-allocate all the list of global objects at loading time §IV-D. This is because the kernel always assigns `MAP_PRIVATE` in memory pages used for global objects, prohibiting to be used for a page aliasing mechanism. This redundant allocation phase can be avoided if we can provide yet another ELF loader in the Linux kernel, which specifies `MAP_SHARED` for those memory pages.

VIII. ADDITIONAL RELATED WORK

Pointer-based Memory Error Detection. Pointer-based detection techniques keep track of pointer capabilities and check the validity of memory access based on the capabilities. Depending on where the capabilities of pointers are stored, pointer-based detectors can be further classified into fat-pointer-based and disjoint-metadata-based. CCured [24] is a representative work for fat-pointer-based approaches where unsafe (*WILD*) pointers are extended with its capabilities stored together with the pointer itself. One drawback of software fat-pointer-based approaches is that they break the memory layout compatibility with unprotected code, which requires special hardware support (*e.g.*, CHERI [36]) to eliminate. SoftBound [21] is a representative work in disjoint-metadata-based approach where the capabilities are stored in a dedicated table. While this approach does not break the memory layout of objects, accessing metadata is usually more expensive. Intel MPX [17] is a new hardware-based security feature introduced in the latest Intel processors, which is essentially a hardware implementation of SoftBound. There is a couple of works [10] and [11] which utilizes the pointer-based approach with low overhead. SGXBound [18] uses 32-bit of pointer values to store the upper bound of the object, and the upper bound address stores the lower bound of the object. However, it only utilizes 32-bit of virtual address, since they assume that the applications run on Intel SGX which has limited memory.

A critical limitation of pointer-based approaches is in its limited compatibility with C/C++ language features. In order to function correctly, these approaches must propagate capabilities correctly between pointers, which is not easy for certain language features. As a result, they all suffer from backward-compatibility issues especially for C++ programs. For instance, CCured only supports limited features of C and the prototype of SoftBound cannot compile all C benchmarks in the SPEC CPU benchmark suite, so there is still a long way to go before they can support large-scale complex software like Chrome and Firefox. Even commodity features like Intel MPX has false positives in running Chrome browsers [30]. As this issue is deeply connected to the difficulty of covering all different C/C++ syntax use-cases as well as applying optimization for Intel MPX, it is unclear whether it can be resolved in the near future.

Memory Error Exploits and Mitigation. Memory errors (ultimately) will allow attackers to perform *arbitrary memory read and write*. Attackers can then leveraging such capabilities to launch different attacks. For example, a simple stack buffer overflow bug may allow attackers to overwrite (1) stack content with malicious shellcode and (2) the return address, leading to arbitrary code execution when the function returns. Based on how these capabilities are abused, Szekeres et. al [34] classify existing attacks into four categories: code corruption attacks, control-flow hijacking attacks, data-only attacks, and information leak. For each specific exploit strategy, a set of corresponding mitigation mechanisms are then developed. For instance, code integrity measurement (*e.g.*, code signing) [3] and data execution prevention (DEP) [2] is developed to defeat code corruption attacks. And a large number of techniques have been proposed to prevent control-flow hijacking attacks, including Stack cookie [8], shadow stack [9], control-flow integrity (CFI) [1], vtable pointer integrity [39], and code pointer integrity [19]. The problem with mitigation techniques is that arbitrary read and write capability is too powerful that it usually allows attackers to find a new way to launch the attacks.

IX. CONCLUSION

This paper presents MEDS to enhance the detectability of memory errors. MEDS achieves this via utilizing the 64-bit virtual address space to approximate the infinite gap and infinite heap. A novel allocator MEDSALLOC uses page-aliasing scheme to approximate above properties while minimizing physical memory overhead. Our evaluation on MEDS using large-scale real-world programs showed that MEDS provides good compatibility and detectability with moderate runtime overhead.

ACKNOWLEDGMENT

This research was supported, in part, by NSF award CNS-1718997, ONR under grant N00014-17-1-2893, Korea NRF/MSIT (2017M3C4A7065925), and Samsung Research Funding & Incubation Center (SRFC-IT1701-05).

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] S. Andersen and V. Abella. Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[3] Apple. ios security. https://www.apple.com/la/iphone/business/docs/iOS_Security_May12.pdf, 2012.

[4] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Ottawa Linux Symposium (OLS)*, 2009.

[5] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[6] T. Boland and P. E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 45(10):0088–90, 2012.

[7] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium (Security)*, 1998.

[9] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[10] G. J. Duck and R. H. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142. ACM, 2016.

[11] G. J. Duck, R. H. Yap, and L. Cavallaro. Stack bounds protection with low fat pointers. In *Symposium on Network and Distributed System Security*, 2017.

[12] O. S. Foundation. OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/, 2017.

[13] T. A. S. Foundation. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2017.

[14] Google. Octane - The JavaScript Benchmark Suite for the modern web. https://developers.google.com/octane/, 2017.

[15] M. Hicks. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/, 2014.

[16] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[17] Intel Corporate. Intel architecture instruction set extensions programming reference. https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference, 2013.

[18] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221. ACM, 2017.

[19] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[20] Microsoft Support. How to use pageheap.exe in windows xp, windows 2000, and windows server 2003. https://support.microsoft.com/en-us/kb/286470, 2009.

[21] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[22] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management*, 2010.

[23] S. Nagarakatte, M. M. Martin, and S. Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 175. ACM, 2014.

[24] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[25] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[26] G. Novark and E. D. Berger. DieHarder: Securing the Heap. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[27] T. C. Projects. Using ClusterFuzz. http://dev.chromium.org/Home/chromium-security/bugs/using-clusterfuzz, 2017.

[28] P. M. Sanjay Ghemawat. Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2014.

[29] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[30] K. Serebryany. Address sanitizer intel memory protection extensions. https://github.com/google/sanitizers/wiki/AddressSanitizerIntelMemoryProtectionExtensions, 2016.

[31] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-Sanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[32] F. J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[33] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[34] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[35] L. Tovalds. mremap feature discussion. https://lkml.org/lkml/2004/1/12/265, 2004.

[36] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[37] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[38] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017.

[39] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining trust on virtual calls. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.