SlimGuard: A Secure and Memory-Efficient Heap Allocator

Beichen Liu
Virginia Tech
beichen.liu@vt.edu

Pierre Olivier
Virginia Tech
polivier@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

Attacks on the heap are an increasingly severe threat. State-of-the-art secure dynamic memory allocators can offer protection, however their memory footprint is high, making them suboptimal in many situations. We introduce SlimGuard, a secure allocator whose design is driven by memory efficiency. Among other features, SlimGuard uses an efficient fine-grain size classes indexing mechanism and implements a novel dynamic canary scheme. It offers a low memory overhead due its size classes optimized for canary usage, its on-demand metadata allocation, and the combination of randomized allocations and over-provisioning into a single memory efficient security feature. SlimGuard protects against widespread heap-related attacks such as overflows, over-reads, double/invalid free, and use-after-free. Evaluation over a wide range of applications shows that it offers a significant reduction in memory consumption compared to the state-of-the-art secure allocator (up to 2x in macro-benchmarks), while offering similar or better security guarantees and good performance.

CCS Concepts • **Security and privacy** → **Systems security**; • **Software and its engineering** → **Allocation / deallocation strategies**.

ACM Reference Format:
Beichen Liu, Pierre Olivier, and Binoy Ravindran. 2019. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361532

1 Introduction

Attacks targeting the heap have become an increasingly severe threat in the recent years. Heap-based vulnerabilities including buffer over-reads and overwrites, invalid and double free, or use-after-free, are regularly detected in applications written in memory-unsafe languages such as C and C++, widely utilized in today's computer systems. Attacks exploiting these vulnerabilities can lead to sensitive data leak [7] or corruption [17], control flow hijacking/arbitrary code execution [9, 26, 33, 34], as well as denial of service.

An obvious line of defense to protect against heap vulnerabilities exploitation is to use a secure dynamic memory allocator. Such allocators have been proposed in both the industry and academic domains [1, 10, 25, 28, 31, 32]. Early work suffered from performance [25, 28] and compatibility [10] issues, as well as low security guarantees [1, 25]. These drawbacks are all addressed in state-of-the-art secure allocators [31, 32]. However, these allocators still have limitations in terms of memory overhead: they assume that memory is available in large quantities. We measured that in some situations, the memory usage of such state-of-the-art allocators is quite high. For example, Guarder [32] has a 71.3% memory overhead compared to the standard Glibc malloc implementation for the PARSEC [5] canneal benchmark. In the age of cloud computing with tenants renting and paying computing resources (including memory) on-demand, this is quite problematic. Moreover, nowadays security is crucial in memory-constrained environments such as Edge and IoT.

State-of-the-art secure allocators' memory footprint is due to several factors, including reasons related to the various security guarantees offered by the allocator. For example, the addition of a one byte canary after a heap object to protect against overflows may lead to the selection of a larger class size and thus to memory waste. In this paper, we address the following challenge: *can we design a memory allocator that is both secure and memory efficient, while still offering good performance and compatibility?*

In that context, we propose SlimGuard: a secure memory allocator focusing on having a low memory overhead. It aims to offer similar security guarantees compared to state-of-the-art secure allocators, while having a memory overhead similar to standard non-secure memory allocators (e.g. Glibc's). To that aim, we evaluate the impact on memory consumption for various security features of state-of-the-art secure allocators. We integrate these features in SlimGuard, redesigning them to have a low memory overhead, and for some of them to increase the security guarantees offered. SlimGuard differs from existing secure allocators on several points. It uses fine-grain size classes indexed by an efficient mechanism, significantly reducing memory waste compared

Middleware '19, December 8–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7009-7/19/12...$15.00
https://doi.org/10.1145/3361525.3361532

to the classic power-of-two size classes. It combines two important security features, randomized mappings and over-provisioning, into an a single memory efficient mechanism, *entropy-based over-provisioning*. Heap objects metadata are segregated from the corresponding data to protect against metadata corruption attacks. Metadata is indexed by a combination of bitmaps and free-lists for fast `malloc` and `free` operations while keeping memory overhead low. An original type of heap canary is provided in the form of *dynamic canaries*, where canary values differ among objects, at no additional memory overhead cost. Further memory savings are obtained with on-demand metadata allocation. Additionally, SlimGuard offers randomized memory allocations, double-free checking mechanisms, and guard pages.

While keeping a controllable memory footprint and offering security guarantees, SlimGuard also offers performance similar to Glibc's malloc, and presents a high degree of compatibility as it does not require application recompilation.

We evaluate SlimGuard's security mechanisms and show that it protects against a wide range of existing exploits. We also evaluate its memory overhead and performance over micro-/macro-benchmarks including the PARSEC [5] and MiBench [16] suites. SlimGuard outperforms the state-of-the-art secure memory allocator by 2x in terms of memory consumption for a number of macro-benchmarks, while offering similar or better levels of security and performance.

The contributions presented in this paper are as follows: (1) The design of SlimGuard, a secure memory allocator whose security features integration is driven by memory efficiency; (2) The implementation of SlimGuard; and (3) Its evaluation, demonstrating its security guarantees, low memory overhead, and good performance compared to state-of-the-art memory allocators, both secure and non-secure.

The rest of the paper is organized as follows: in Section 2 we motivate securing the heap and discuss existing secure allocators as well as related work. We describe SlimGuard's threat model in Section 3 and present its design and implementation in Section 4. We analyze and evaluate the security features provided by SlimGuard in Section 5, then evaluate its performance in Section 6. We conclude in Section 7.

2 Motivation and Related Works

In this section, we detail the most common heap-related vulnerabilities, and show that nowadays they are an increasingly severe threat. Next, we present existing secure memory allocators and point out their limitations motivating the design of SlimGuard. Finally, we discuss related work.

2.1 Heap-Related Vulnerabilities

The occurrence of heap-related vulnerabilities discoveries and related attacks is dramatically increasing in recent years [31, 32]. We used the National Vulnerability Database (NVD) [27] to search for reported vulnerabilities on the heap since 2010, and counted the number of heap-related Common Vulnerabilities and Exposures (CVE) entries per year. These numbers

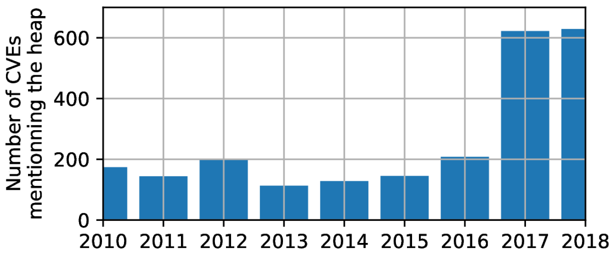

Figure 1. Number of heap-related CVE entries since 2010.

are presented on Figure 1. As one can observe, the number of CVE entries mentioning the heap has tripled after 2017, to reach more than 600 occurrences in both 2017 and 2018. Although there is a plethora of work targeting the 'static' part of the address space such as the code segment or static data, a large part of this work is not applicable to the heap [35].

Vulnerabilities on the heap can fall into diverse categories. A heap *overflow* happens when the program performs an out-of-bound write operation past a heap object due to a bug or a lack of proper bound checking. Less common, heap *underflows* concern situations where a buggy access to a heap buffer leads to memory being written before the buffer. Similarly, heap objects *over-read* and *under-read* concern read rather than write accesses. All these operations allow an attacker to potentially write and read part of the address space which can lead to information leaks (such as HeartBleed [7]) or data/metadata corruption [17], leading to control flow hijacking or denial-of-service.

Other vulnerability classes include *use-after-free* (or dangling pointers), in which memory is erroneously accessed after having been freed. Consequences depend on the use of the accessed memory. They include information leak [36] but also to control flow hijacking [30], when the freed memory is later reallocated for an object of type B and then accessed as the initial object of type A [36]. An *invalid free* happens when the application tries to free a value that is not pointing to an object created by the memory allocator. A *double free* happens on a pointer that was already freed in the past. Invalid and double-frees can be exploited for arbitrary code execution, data corruption, and denial-of-service [31].

A past study [32] presents a breakdown by type of NVD's heap-related CVEs and note that the most common are, by far, overflows, followed by use-after-free and over-reads. Invalid and double-frees are less common.

2.2 Existing Secure Memory Allocators

An obvious level of protection against the exploitation of these vulnerabilities is the dynamic memory allocator, i.e. the implementation of `malloc`, managing heap objects. Here, we present the existing memory allocators providing security features [25, 28, 31, 32].

OpenBSD. The allocator of OpenBSD 6.0 [25] (referred to as OpenBSD in the rest of this paper) is an evolution of PHK-malloc [18] originally written for FreeBSD. OpenBSD adds security features including (1) the segregation of data and

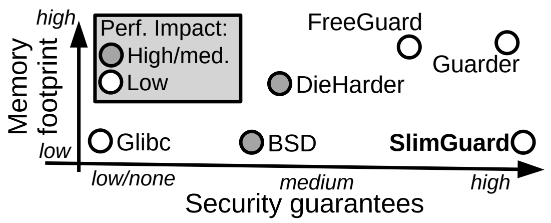

Figure 2. SlimGuard in the secure allocators design space.

metadata in order to protect against metadata exploits based on data buffer overflowed onto inline metadata and (2) randomized allocations making it harder for an attacker to determine the layout of the heap. Due to its reliance on a bitmap rather than a free-list to maintain the status (free/used) of heap objects, OpenBSD has a low memory overhead. However it suffers from performance problems (up to a 4x slowdown [31]) due in particular to frequent invocations of the `mmap` system call. Limitations in terms of security guarantees have also been identified, such as an unstable level of entropy for the allocation of small objects (with size <2KB), or an outright low entropy for large objects [32].

DieHarder. DieHarder [28] implements security features such as randomized allocations. It also offers over-provisioning, a technique in which some object slots are never allocated, giving a probabilistic chance that an overflow hitting an unallocated slot will have no impact (at the cost of memory overhead). DieHarder has been identified to have a non-negligible memory overhead, an unstable randomization entropy, and a significant (up to 9x) performance impact [32].

FreeGuard and Guarder. FreeGuard [31] improved over OpenBSD and DieHarder by combining all security features from previous work with a negligible performance overhead. In SlimGuard we provide similar security features, although designed and implemented differently, so we will depict them in details in the next section. Guarder [32] is an evolution of FreeGuard, and can be considered as the state-of-the-art secure memory allocator. Guarder improves upon FreeGuard by providing a deterministic level of entropy that is tunable and can be set higher than FreeGuard's low entropy (2 bits). Guarder also presents a negligible performance overhead. Both FreeGuard and Guarder do not focus on memory overhead and as a result these systems have a very large memory footprint: more than 2x in multiple scenarios [31, 32].

Given that existing secure memory allocators have issues either in terms of performance (OpenBSD/DieHarder), security (OpenBSD/DieHarder/FreeGuard), or memory overhead (DieHarder/FreeGuard/Guarder), we propose SlimGuard– a memory allocator that offers a low memory footprint, good performance, and good security guarantees. SlimGuard is thus a novel point in the design space, as illustrated by Figure 2.

2.3 Other Related Works

Other secure allocators have been proposed in the past. In general, each focuses on a particular security feature, such as providing a non-deterministic layout/location of the heap [4, 19, 29], or segregating data and metadata in the address space [19, 37, 38]. These features are supported in SlimGuard. Cling [1] protects against use-after-free vulnerabilities by forcing address space to be reused only by objects of the same type. SlimGuard also protects against such vulnerabilities by a combination of segregation of metadata, randomized allocations, and guard pages.

Other protection techniques focus on a particular type of vulnerability on the heap. Multiple work have targeted use-after-free and/or double-free vulnerabilities, either through the use of a customized compiler for FreeSentry [36] and DangNULL [21], or at runtime for Undangle [6]. Other work targets buffer overflows either through the introduction of additional metadata checked at runtime [2, 3] or with the use of a customized compiler [10]. SlimGuard provides protection against many vulnerabilities. Moreover, we do not require recompilation nor access to the sources, which is not acceptable in some cases such as the use of proprietary code.

3 Threat Model and Assumptions

We assume a threat model similar to the one of state-of-the-art secure allocators [31, 32]. It is well-known that security by obscurity is not a good practice, so we assume that it is possible for the attacker to access the sources of SlimGuard.

We trust the host operating system (OS), in particular the fact that the `mmap()` system call can provide sufficiently randomized virtual memory areas for SlimGuard to keep secret the start addresses of data and metadata areas, as well as the loading location of SlimGuard's shared library code and static data. We also assume that the system is correctly configured so that the location of such areas is not accessible through channels such as the `/proc` virtual filesystem [11]. We assume a 64 bit machine as the host, and we trust the random number generator to be efficient and not to be tampered with or subverted by the attacker.

4 SlimGuard: Design and Implementation

The security principles implemented within SlimGuard are the following: *Randomized memory allocations* with a significant entropy remove the capacity by the attacker to create a deterministic layout of objects on the heap [33]. *Over-provisioning* protects in a probabilistic way against buffer overflows. *Segregating metadata from data* allows to protect against metadata corruption-based attacks [17] that are straightforward in systems storing metadata inline as headers with dynamically allocated objects. These metadata include in particular the state of each slot (free or used), checked upon `free` to protect against double-free-based attacks. Heap over- and under-flows are protected against with the use of *heap canaries*. Unmapped *guard pages* prevent heap buffer overflows and over-reads. Use-after-free attacks are made harder by using *delayed randomized memory reuse* and optionally *destroying data on free*.

In the rest of this section, we first give an overview of SlimGuard's working principles. Next, we describe each security feature we provide and put the emphasis on how we optimize its integration for low memory consumption. Next, we discuss SlimGuard' compatibility. It is important to note that, although SlimGuard integrates similar features as existing secure allocators, due to our focus on memory consumption the design and implementation of such features differs significantly from existing allocators.

4.1 Overview

Small vs. Large Objects. A central feature of a dynamic memory allocator concerns how to manage objects of different sizes. Generally, a distinction is made between small and large objects [15, 20, 31, 32]. Small objects are managed internally by the allocator, while for large objects the allocator generally relies on the mmap and munmap system calls. Large object management is thus relatively straightforward. It is also secure because of the high level of entropy of Linux's anonymous mappings (at least 28 bits and up to 33 for hardened kernel with PaX/grSecurity [24]) and the fact that such mappings are generally surrounded by unmapped pages makes them robust against overflows. Moreover, because such objects are freed with munmap, a use-after-free would invariably trigger a page fault and crash the program. The relatively high cost of invoking the mmap system call has to be put into perspective with the relatively low frequency of the allocation of such large objects. SlimGuard adopts a similar management for large objects, the configurable threshold being set by default at 128 KB. In the rest of this section we focus on depicting small objects management.

Size-Classes Management and Indexing. Existing state-of-the-art secure memory allocators [31, 32] use power-of-two size classes to manage small objects. Indeed it is quite likely for the programmer to request memory with a size equal to a power-of-two. Although this is intuitive in non-secure allocators, in state-of-the-art secure allocators this ends up being the source of a very large memory overhead. In effect such allocators place a *heap canary* right after allocated objects in the address space. The canary is a small one-byte value used to check for buffer overflows. It is placed with the object within the allocation slot. It makes that with any allocation of a power-of-two size 2^n, that object will need to be allocated in the next size-class, i.e. 2^{n+1}, to be able to store a small single-byte canary alongside the data. This effectively wastes $2^n - 1$ bytes of memory. Because of the high likelihood of power-of-two allocations, this leads to a potentially huge memory overhead for state-of-the-art secure allocators – for example in PARSEC [5] canneal, 88.1% of the 20 million calls to malloc fall into this category. As a result Guarder has a 70% memory overhead for this benchmark (see Section 6).

Supporting canaries in SlimGuard is crucial as buffer overflows are the most common vulnerability on the heap [32]. We decide not to rely on power-of-two size classes and rather define finer-grain size classes. We have a total of 176 size classes, divided into 11 *subdivisions* of 16 size classes each. It is illustrated on Figure 3 Ⓐ and Ⓑ, where each slot of the array Ⓑ corresponds to a size class. Slots representations on the Figure contain the lowest and highest sizes managed by that particular class. The managed sizes of classes within a subdivision increase linearly by a factor determined by the subdivision index. We choose to have 11 subdivision to obtain a gradual size class increase up to 128 KB, the small/large object boundary.

To index size classes we use a one-dimensional array Ⓑ containing one element per size class, each being a pointer to the beginning of an area containing the data Ⓒ. We name it the *data area*. We align all heap objects within that area to an 8-byte granularity for easy management purposes. When it is needed to satisfy an allocation request of size *size* we use the following formula to find the corresponding slot: $index = 16*(I_{MSB}-6)+bits[I_{MSB}-1 : I_{MSB}-log_2(16)-1]$. In this equation, 16 is the number of size classes per subdivision, I_{MSB} is the index of *size*'s MSB, and 6 is a constant derived from the number of subdivisions and a minimum alignment of 8 bytes for size classes boundaries.

Because of this fine-grain size-classes division, memory wastage is significantly reduced even in the presence of canaries and power-of-two-sized allocations. Let us take the example of a program calling malloc(32). After the extra byte for canary, the allocation needs to be rounded up to 40 for SlimGuard instead of 64 for state-of-the-art secure allocators such as FreeGuard or Guarder. In that case we are effectively saving 40% memory. Our fine-grain size class management scheme takes inspiration from Two-Levels Segregated Fit [23]. In essence, we improve that scheme by merging its two levels of indexation into a unidimensional array, optimizing both metadata memory footprint and indexing computations over the original algorithm.

Managing the Data Area. The data area (Ⓒ on Figure 3) is a large area (multiple GBs) of contiguous virtual memory allocated through mmap the first time a size-class is used. It is composed of a used (mapped) section containing fixed-size free and used slots Ⓓ, and an unused (unmapped) section Ⓔ. These sections are separated by the data area *limit pointer* Ⓕ, which is dynamically incremented when the number of free slots is low. The mapping of the data area to actual physical memory happens implicitly as Linux performs on-demand mapping for anonymous mappings requested without the MAP_POPULATE flag such as our data area.

Indexing free slots can be made using a bitmap [25, 28] which is memory-efficient but very slow to scan upon malloc calls to find a free slot to serve an upcoming allocation request. The other solution is to use a free-list Ⓖ which is efficient in terms of performance, but consumes more memory because (1) pointers need to be stored and (2) we may index in the free-list slots that will never be used which is pure memory waste. In SlimGuard we use a free-list to index free slots. Thus, allocation is made with an O(1) complexity.

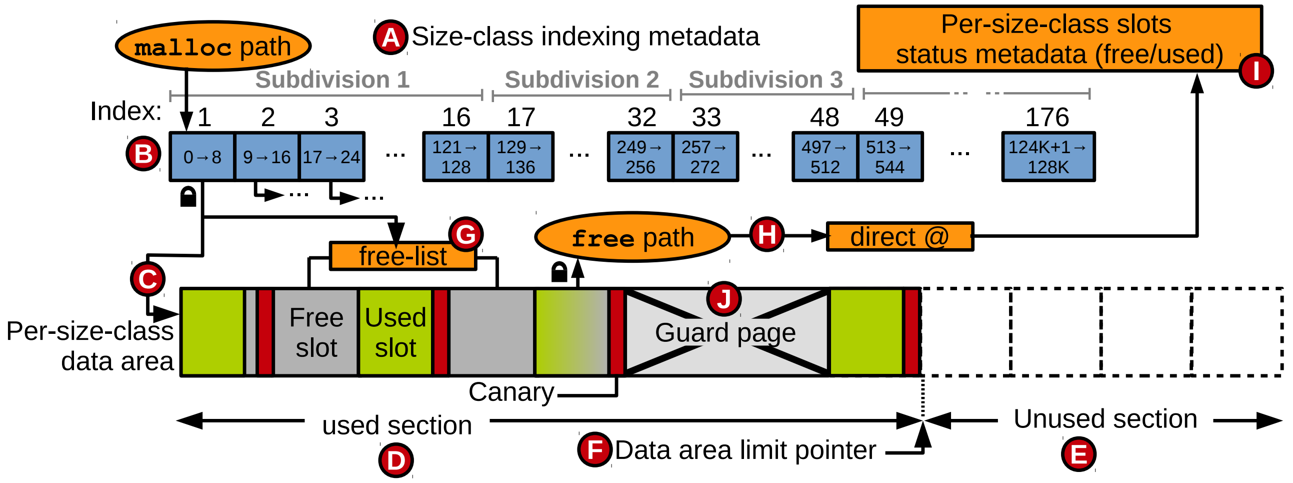

Figure 3. Overview of SlimGuard's design, including size-class indexing metadata (Ⓐ and Ⓑ, each slot containing the range of sizes it manages), free/used slots status metadata (Ⓘ) and one of the per-size-class data area (bottom).

We minimize the memory impact of this list by having it index only the free slots of the used (mapped) section Ⓓ. Thus, in program making only a few memory allocations the free-list memory usage is relatively low, whereas in application making more allocations it is larger in absolute but more acceptable in regard to the application consuming more memory itself.

For each size class, the corresponding data area as well as its management data structures (such as the free-list) are created and allocated on-demand the first time an allocation is made for this particular class size. It is crucial to do so because the amount of size classes in SlimGuard is higher than in other allocators using power-of-two size classes: in order to manage small objects up to 128 KB, we use 176 size classes whereas power-of-two allocators would only have 15 size classes. Such on-demand allocation helps to limit the per-size-class overhead, in particular in programs doing just a few memory allocations.

Processing `malloc` and `free`. In the general case, both `malloc` and `free` are executed in constant time in SlimGuard.

When `malloc` is called, the relevant data area is identified using the indexing array as described above (Ⓐ, Ⓑ and Ⓒ on Figure 3), according to the requested size. Next, a free slot is picked from the free-list Ⓖ. These operations are thus efficiently realized in constant time (the free-list is an array that can be directly addressed). When the size of the free-list falls under a certain threshold, we increment the data area limit pointer by the size of a slot and add the new free slot to the free-list.

When `free` is called Ⓗ, a bitmap maintaining the free/used status of slots (Ⓘ on Figure 3) is first checked to protect against double frees. As a bitmap it takes very little space in memory. The slot is then marked as free in these metadata Ⓘ and added to the free-list Ⓖ.

We have a per-size-class bitmap and when `free` is called, in order to access this bitmap we need to know the size-class of the freed buffer. For security reasons the data areas are located at random locations one from another, so it is not possible to infer the size class of a pointer from its location in constant time: we have to iterate and check for each allocated data area if the pointer falls within its boundaries. In order to speed-up that process, we always start to iterate with the size-class of the latest previously freed pointer. This is quite efficient as we noticed that a lot of programs free buffers of the same size one after the other (a common example is a multi-dimensional array allocated and freed within loops). In the best case, which happens very often, `free` is achieved in constant time.

SlimGuard maintains per-memory-page used object counters updated with allocations and deallocations. When a counter reaches 0, the corresponding memory page is released to the OS through a `madvise` call with the parameter `MADV_DONTNEED`. The counters themselves are allocated on-demand and do not generate memory consumption for data areas' pages past the limit pointer.

In the next subsections we describe SlimGuard's security features. For each feature we detail the protection offered, an analysis of its memory impact, and a description of its implementation within SlimGuard.

4.2 Randomized Allocations and Over-Provisioning

Protection Offered. Non-secure dynamic heap allocators do not randomize the addresses returned by `malloc`, leading to situations in which the attacker can deterministically determine and control the heap layout. A concrete example of attack is *Heap Feng Shui* [33], in which a carefully crafted sequence of memory allocation requests is used to generate a deterministic heap layout and ensure the success of a subsequent heap spraying attack. The level of security for randomized allocations is measured in number of bits of entropy, indicating how many different locations can be returned by a call to malloc (*n* bits of entropy correspond to 2^n possibilities). That number should be high enough, for example FreeGuard [31] only offers 2 bits of entropy which corresponds only to 4 possible locations. That number should also be stable [32], i.e. the system should give a guaranteed minimal number of bits of entropy for each allocation. In OpenBSD [25] and DieHarder [28], that entropy is unstable and depends of the state of the allocator. Entropy can fall as low as 3 bits for OpenBSD and 5 bits for DieHarder [32].

Over-provisioning [28, 31, 32] is a technique with which a certain number of heap slots are never used by the allocator. These slots are located randomly in between used slots. The rationale is that a buffer overflow ending in such an unused location will be tolerated and will have no impact.

Memory Overhead Considerations. The memory overhead generated by randomized allocations is fully dependent on the method for providing such allocations. However, to provide a sufficient level of entropy, this overhead can be quite large. Indeed, to avoid calling mmap to obtain memory pages too often (something that OpenBSD does at the cost of a non-negligible performance overhead), secure allocators such as Guarder [32] maintain multiple per-size class pools with multiple allocated pages from which requests for allocations are served. In such cases the more entropy bits are required the higher the memory overhead is.

Over-provisioning is a direct loss of memory, although it can help to tolerate against overflows in a probabilistic way. It is a good example of a direct trade-off between memory consumption and probabilities of attack protection.

Integration in SlimGuard. In SlimGuard we combine randomized allocations and a certain form of over-provisioning (OP) within a single feature called *entropy-based OP*. In our design, it corresponds to *ensuring that the free-list size will never fall under a certain threshold*. This is simply made by creating a new free slot through the increment of the data area limit pointer (Ⓕ on Figure 3). In effect, we increment that pointer by the size of a slot when the allocation of an object makes the free-list size fall below the threshold. Remember that each free slot of the mapped section of a data area is indexed within a free-list. Upon malloc, a free slot in that list is randomly chosen to serve that request. Thus, by ensuring that the list size never falls under 2^n, we can provide a stable *n* bits of entropy for randomized allocations. By doing so, we also ensure in effect that 2^n randomly distributed free slots will never be allocated, providing a form of OP at no additional memory cost.

Because such OP is a side effect of randomized allocations, it is not as efficient as directly-managed OP techniques [31, 32] that give the user a direct control about the percentage of OP space. Moreover, with such techniques free slots are spread evenly across the data area. In order to thwart overflows an efficient OP scheme maximizes the number of free slots that are contiguous to a used slot. In SlimGuard, any type of control over the free-slots providing entropy would go against the fact that their location should be random. Thus, we also provide an option to place additional free slots evenly within the data area for situations where memory consumption is less of a concern but the amount of OP must be controllable.

4.3 Data and Metadata Segregation

Protection Offered. Multiple legacy heap allocators store metadata (data free/used status, free-list pointers, etc.) inline with the data. Because of the close and deterministic location of such metadata related to the corresponding data, it is relatively straightforward for an attacker to overwrite the metadata, for example through a heap object over- or underflow. This is at the core of the unlink [17] attack where Glibc's DLmalloc implementation is tricked into overwriting a function pointer target with the address of shellcode when processing carefully corrupted heap metadata.

In secure allocators, metadata maintains the status of each object (free or not). This status is checked when the object is freed to protect against double or invalid free.

Memory Consumption Considerations. Moving the metadata out-of-band indirectly increases memory consumption: indeed, inline metadata can be quickly looked up given an allocated object address, for example in the case of a call to free. When metadata is segregated from allocated objects, such a lookup requires an indexing mechanism that may itself consume memory. These indexing data structures may also be used to find free memory blocks for upcoming allocation requests. Allocators offering slots of fixed size [1, 25, 28] can have their metadata indexed with simple bitmaps that are quite memory efficient, but slow to scan when searching for free blocks [31, 32]. Similar to allocators with inline metadata, secure allocators using slots of non-fixed sizes [31, 32] link these metadata with *free-lists* which management and usage is less memory-efficient than bitmaps but offers increased performance.

Integration in SlimGuard. In SlimGuard, data and metadata are segregated: the free/used status of each slot is in a bitmap which is located at a random place in the virtual address space (Ⓘ on Figure 3), i.e. at a random offset from the corresponding data slot. In SlimGuard, it is possible for us to use a bitmap for such metadata because we have fixed-size slots: upon free, we can then quickly lookup the right bit in the bitmap in constant time based on the slot position within the data area. However, as mentioned above, bitmaps are slow on the malloc path as they need to be scanned. Thus, we also use a free-list for which we try to keep the size as small as possible, as explained above in the description of the malloc operation: they only index free slots from the mapped section of the data area.

4.4 Dynamic Canary

Protection Offered. Canaries are guard values with a size of generally 1 byte, placed before or after allocated objects on the heap to protect against overflows. An overflow on a given object would overwrite and modify the value of the corresponding canary, and a security flag is raised when detecting a change in the canary value. Canaries are important as they protect against overflow, by far the most widespread vulnerability on the heap [31, 32]. However, their efficiency is limited by the fact that canaries values can only be checked at `free`. Existing allocators [31, 32] generally check a small set of canaries during each `malloc` and `free` operation. Moreover, it is possible to leak the value of a canary through a buffer over-read, which is in the top-3 most common vulnerabilities on the heap [32]. Unfortunately state-of-the-art secure allocators use the same value for the canary among all allocated objects. Thus, a leak breaks the entire canary system, i.e. it becomes possible to successfully overflow buffers by setting the canary to the particular leaked value.

Memory Consumption Considerations. The overhead due to canaries is twofold. It is first comprised of the space needed to store the canary itself, which is equal to the size of the canary multiplied by the number of allocated objects. Secondly, because of the size-class system used in state-of-the-art secure allocators, the additional space required for the canary may lead to the selection of an allocation slot of next size class (by definition the canary needs to be stored inline with the data). As previously described, because existing secure allocators implementing canaries use power-of-two size classes, this can have a significant impact on the memory overhead [31, 32]. As a matter of fact canaries are generally disabled in the evaluation of such systems because of this large memory overhead [31, 32].

Integration in SlimGuard. Because of the fine-grain size classes we use in SlimGuard, as previously mentioned the memory impact of canaries triggering a jump to a superior size class is significantly reduced.

In SlimGuard we also acknowledge the possibility of a canary value leak. Such an event would break any secure allocator having a static canary (a single canary value), which is the case for all state-of-the-art allocators s[31, 32]. We propose a new method in which canaries values are different among separate objects on the heap without impacting memory overhead: *dynamic canaries*. When an object is allocated, we hash its address and use the hash value as a 1-byte canary, placed at the end of the allocated slot as depicted on Figure 3. The dynamic canary technique is more secure than the state-of-the-art allocators, while staying memory efficient. Indeed, because we use a hash value of the returned address, SlimGuard does not have to store this value for comparison later when the time has come to check its consistency.

The hash function used to compute canary values is created at load time, and will be different over multiple runs of a program. It is composed of a random combination of fast operations including multiple basic bits manipulations such as shifts of different values, XORs, etc. and arithmetic computations with random numbers such as additions and multiplications. As a result, it is very hard for an attacker to find the hash function even in the presence of multiple canary leaks.

At `free` time, SlimGuard rehashes the deallocated pointer and compares this value with the canary at the end of the corresponding slot. If the values are different, the behavior of SlimGuard is configurable: a security flag can be raised, or the program can be killed. In both cases information about the location of the potential buffer overflow can be printed. When `free` is called, in order to locate the canary we need to know the size class the object belongs to. We also need to perform that same operation to access the bitmap containing the free/used status for checking against double-free. Thus, we only have to go through that process once for both canary and double-free checking.

The canary is placed at the end of each slot rather than right after the buffer. In that way, we can avoid storing the size of the buffer in order to retrieve the canary value when it needs to be checked. Note that any overflow ending up between the buffer and its canary will have no effect. Currently the canary is only checked at `free` time but it is also possible to check sets of canaries during `malloc` and `free` as made by other secure allocators [31, 32].

4.5 Guard Pages

Protection Offered. Guard pages are unmapped virtual memory pages placed close to allocated heap objects. In terms of overflow detection, a guard page acts as a perfect canary because any overflow hitting a guard page will instantly trigger a page fault. Moreover, guard pages provide a more comprehensive protection: on the contrary to canaries they also protect against buffer over-reads. Such pages also do not consume memory because they are not mapped. However their granularity of placement is obviously limited at the page level so it is not possible to place a guard page after each allocated object on the heap.

One issue with guard pages is the performance cost of placing them. OpenBSD [25] implicitly places a guard page between each data page by requesting each of these data pages on-demand through `mmap`. Because it involves a system call it is costly in terms of performance. In FreeGuard [31], at load time a large virtual memory area is allocated for data and guard pages are placed randomly within. By default, 10% of this area becomes guard pages. It is done with `mprotect`, considerably faster than `mmap`. However this involves a lot of operations during the initialization process which may slow down fast-executing programs making just a few memory allocations. Guarder [32] has a similar initialization phase and introduces additional guard pages at runtime.

Integration in SlimGuard. In SlimGuard, we propose *fully-on-demand guard pages*. Each time the data area limit pointer (Ⓕ) in Figure 3) is incremented by one element to obtain a new free slot, we check if we are at the frontier of a memory page, i.e. if the newly created slot would span over the next memory page. If it is the case, we have the opportunity to place a guard page there and create the new free slot on the subsequent page. This opportunity occurs every other page for class sizes of less than 4096 bytes (the size of one page). Although it happens less frequently considering larger class sizes, because of their size it actually leads to one opportunity to place a guard page between every object.

In SlimGuard, we place on-demand guard pages by calling `mprotect`. Although it is about 20x faster than `mmap`, it is still a system call and therefore has a non-negligible performance overhead. SlimGuard can be configured to place a guard page once every n opportunities (per size class).

In the case of fast-executing programs making just a few memory allocations, the fact that SlimGuard places guard pages in an on-demand fashion makes that its performance overhead is negligible compared to the tens or hundreds of thousands of `mprotect` calls made by state-of-the-art allocators such as Guarder or FreeGuard during initialization.

4.6 Other Security Features

Delayed Memory Reuse. Delayed memory reuse [25, 31] consists in delaying the reuse of freed memory objects in order to make use-after-free attacks more difficult: if the buffer is not reused, such attacks have good chances to fail. This is generally implemented through the use of a delay buffer that holds freed memory slots for a given number of allocations after which they can be reused.

In SlimGuard, the free-list for each data area acts as a random delay buffer. The next allocated element from a particular size class will be picked randomly within the free-list which is guaranteed to be at least of size 2^n, n being the level of entropy selected. If this entropy is large enough, when an element is sent back to the free-list it has very low chances to be picked up as one of the next allocations.

Destroy-on-Free. Destroy-on-free [28] is a technique which consists in zeroing out every freed object or filling it with random data. This helps in preventing uninitialized reads and use-after-free, however the performance impact can be significant due to the memory write overhead. In SlimGuard it is left to the user discretion to activate destroy-on-free, which is implemented as a simple call to `memset`. Similarly to other allocators [32], destroy-on-free is disabled by default due to its impact on performance.

4.7 Multithreading and Compatibility

Highly scalable allocators use distributed, per-thread data structures [12] to concurrently serve allocation requests and offer high performance with large number of threads. This has the drawback of presenting a memory overhead that increases with the number of threads. It is the case for state-of-the-art secure allocators. Because we focus on memory efficiency, in SlimGuard we choose to have a memory overhead independent of the number of threads, without compromising too much on performance scalability. SlimGuard supports multithreading through the use of fine-grained locking. In effect we serialize each access to a size-class data structures with pthread mutexes, as represented by the locks illustrated on Figure 3. Thus, two threads can concurrently allocate or free memory of different size classes.

Table 1. Security features comparison.

Feature	GLib-C	Open-BSD	Die-Harder	Free-Guard	Guar-der	SlimGuard
Canary	✓ Static	- Static (weak)	✘	- Static	- Static	✓ *Dynamic*
Rand. alloc. entropy	✘	- Unstable	- Unstable	- Low (2 bits)	✓*Stable high*	✓*Stable high*
Over-prov.	✘	✘	✓	✘	✓	✓
Guard pages	✘	- On-Demand (slow)	- Weak	- Static	✓ Static+ OD	✓ *On-Demand (fast)*
Segregated metadata	✘	✓	✓	✓	✓	✓
Double & invalid free detection	✓	- (Weak)	✘	✓	✓	✓
Delayed mem. reuse	✘	✓	✓	✓	✓	✓

This allows SlimGuard's performance to scale to a medium number of threads, however it is unlikely to scale to large number of threads (e.g. hundreds). We believe it is acceptable for two reasons. First, applications with very high number of threads are not the norm: a recent study over the entire Microsoft Azure's VM workload [8] shows that more than 95% of the VMs have less than 10 VCPUs. Second, in applications with high numbers of threads it is unlikely that memory footprint represents a critical concern. Thus one would choose over SlimGuard a secure allocator such as Guarder, focusing on performance rather than memory usage.

SlimGuard is implemented in C for efficiency. As most other custom dynamic allocators, SlimGuard is a binary-compatible shared library and the user can use `LD_PRELOAD` to substitute SlimGuard to the default allocator without recompilation for dynamically compiled binaries.

5 Security Analysis and Evaluation

In this section we compare SlimGuard's security features with existing secure allocators and describe how it protects against common heap vulnerabilities. Next, we present SlimGuard's efficiency on some real-world bugs.

5.1 Security Features Comparison

We compare in Table 1 the security features implemented in SlimGuard and the existing secure allocators OpenBSD [25], DieHarder [28], FreeGuard [31] and the state-of-the-art Guarder [32]. We also include Glibc's allocator [15, 20] for reference. Even if it includes a few features such as optional canaries and double/invalid free detection, it is not designed to be secure so most of the features are absent.

Only SlimGuard proposes dynamic canaries, an additional defense in the case of a canary leak. Similarly to Guarder, SlimGuard guarantees a fixed level of entropy which can be set very high, on the contrary to allocators offering unstable (OpenBSD, DieHarder) or low (FreeGuard) entropy settings. SlimGuard supports over-provisioning and separates data and metadata. It provides on-demand guard pages efficiently, i.e. through calls to `mprotect` rather than obtaining implicit guard pages through frequent slow calls to `mmap` as done by OpenBSD. Its design can detect double and invalid frees, and it also provides randomized delayed memory reuse. In conclusion, SlimGuard provides security guarantees that are as good and sometimes better than state-of-the-art secure allocators.

5.2 Security Analysis

Multiple features allow SlimGuard to protect against buffer overflows/over-reads. Overflows targeting inline metadata will fail because of the segregation of data and metadata. Randomized memory allocations remove from the attacker the knowledge of the heap layout and make it very difficult for him to set a particular target for an overflows/over-read. Even if the attacker has access to the application/SlimGuard source code, or even if he can launch dry runs and explore the heap layout with GDB, it is still very hard to determine that layout because it is different for each run. Canaries will detect overflows overwriting them, and an overflow within over-provisioned space will have no impact. Finally, guard pages will protect against both overflows and over-reads.

Concerning use-after-free, SlimGuard offers significant protections in cases where an object A is freed, the memory containing it is reallocated to an object B, and then the program accesses such memory as object A [36]. Because in SlimGuard the memory will be (1) reused with a probabilistic delay and (2) reallocated in a random fashion, it is very hard for the attacker to obtain a meaningful sequence of steps. Moreover, the presence of guard pages can protect against brute-force attempts, but it depends on the sizes of objects A and B. Finally, when activated the destroy-on-write feature can prevent any use-after-free.

SlimGuard protects against any double free as it uses a separate metadata area containing the used/free status of each slot. SlimGuard can also thwart invalid frees, because it checks that a freed pointer falls within a data area (needed for accessing the used/free status metadata).

5.3 Security Evaluation

In this subsections we present SlimGuard's efficiency at protecting against real-world bugs. Note that we mostly use the same examples as Guarder [32] and FreeGuard [31].

Buffer Overflows: gzip-1.2.4 and ncompress-4.2.4. Gzip and ncompress are compression programs which, in these particular versions (obtained through BugBench [22]), contain buffer overflows due to a call to `strcpy` without proper bounds checking. Same as Guarder [32], we moved the target buffers from the stack to the heap for testing purposes. Both bugs are detected by SlimGuard at free time through the check of the overwritten canary, which allows to notify the user that an overflow happened, print the related location, and halt execution. Finally, in the case that buffer is protected by a guard page, the programs would be halted immediately.

Buffer Over-reads: HeartBleed. The well-known HeartBleed [7] bug within the cryptographic library OpenSSL 1.0.1 allows an attacker to supply a malicious payload smaller than its advertised size, which results in a buffer over-read through a `memcpy` operation. With guard pages SlimGuard can protect against HeartBleed in a probabilistic way, depending on the amount of guard pages the system is configured to place which is a parameter defined by the user.

Invalid and Double `free`: ed-1.14.1 and ImageMagick 7.0.4-1. `ed` is a text editor from GNU. This version contains a programming mistake leading to `free` being called on a pointer that was never allocated with `malloc`. SlimGuard detects that the pointer does not fall within one of the data areas and is not a large object either, and halts execution after having printed information about the bug. ImageMagick is a command line image manipulation tool and the concerned version contains a double free vulnerability. Because SlimGuard maintains the used/free status of slots, it is able to detect the bug.

6 Performance Evaluation

After showing in the previous section that SlimGuard provides good or sometimes better security guarantees compared to other secure allocators, with this performance evaluation we show that compared to a state-of-the-art secure allocator (Guarder [32]) SlimGuard offers a significantly lower memory overhead. We also show that SlimGuard performs similarly or better. We define Guarder as the state-of-the-art because (1) FreeGuard [31] and Guarder [32] represent the most recent literature for secure heap allocators and (2) Guarder is an evolution of FreeGuard.

We investigate the following questions: (1) What are the memory overhead and performance of SlimGuard, and how are they influenced by its security features? (2) How does the memory and performance overhead of SlimGuard compare to the state-of-the-art secure allocator Guarder?

We compare SlimGuard to Guarder in micro-/macro-benchmarks, and to OpenBSD (macro-benchmarks) that has security/performance issues, but is also memory-efficient. We also include Glibc in macro-benchmarks because it is the default allocator of many popular distributions. We use a 4 cores (8 hyper-threads) Xeon E5-2637 clocked at 3GHz, with 64GB of RAM. It runs an Ubuntu 16.04 distribution with Linux v4.4. We use the latest version of Guarder [14] and a port of OpenBSD's allocator to Linux [13]. For Glibc we use the distribution's version, 2.24-11. Glibc's code being compiled with -O2 level of optimizations, we use that same level to compile the other allocators and the benchmarks. These allocators are compiled as shared libraries and hooked using `LD_PRELOAD`. We use GCC v5.4.0.

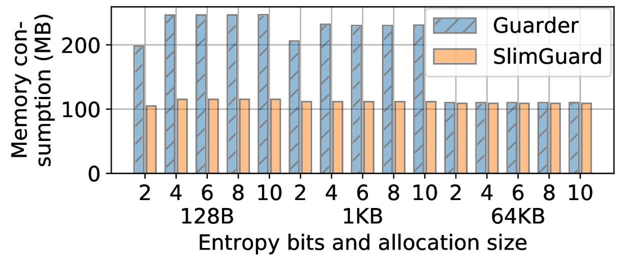

Figure 4. Entropy and allocation size impact on memory usage.

We use a set of micro- and macro-benchmarks. The micro-benchmarks include an analysis of the impact on performance and memory overhead for allocation size, entropy for randomized mappings, and amount of guard pages, as well as a study of the initialization time and over-provisioning efficiency. Concerning the macro-benchmarks, we use the PARSEC [5] suite, representative of data-intensive applications (a large fraction of the software running in today's datacenter) and making an intensive use of the heap. We use the native dataset size and run both serial and multi-threaded versions. As the focus of SlimGuard is security with a low memory overhead, we also run benchmarks from the embedded suite MiBench [16], still widely used today. Embedded systems are memory-constrained, and integrated in security-critical domains such as edge/IoT.

If not otherwise stated, the entropy level is 8 bits, canaries are enabled, destroy-on-free is disabled, and guard pages are set to a value of 10% (1 guard page every 10 data pages).

6.1 Micro-benchmarks

Memory Usage, Randomization Entropy Level. To evaluate the memory savings brought by SlimGuard over Guarder and measure the impact of the entropy level (customizable in both allocators), we ran a micro-benchmark we developed. In this benchmark n buffers of size s are allocated with `malloc` in a loop (n iterations). Next the buffers are filled with `memset` in another loop, then freed in a third loop. We measured the peak memory consumption (resident set size) as well as the execution time of the `malloc` and `free` loops. To investigate allocations of different sizes, we varied s to be 128 B, 1 KB, and 64 KB. We fixed n for each value of s so that $n * s$ is always equal to 100 MB, which gives us a sufficient number of iteration for all cases, and allows us to understand if the performance/memory consumption depends on the allocation size. We also varied the entropy (2 to 10 bits).

Results are presented on Figure 4. With allocation of 128 B and 1 KB, SlimGuard consumes about 2x less memory than Guarder. These memory allocations having a size equals to a power-of-two leads to a jump to the next size class in both systems. Because Guarder uses power-of-two size classes this doubles the memory consumption. In SlimGuard, because of the fine-grained size classes, the memory waste is far inferior. Indeed, the total memory consumption is very close to 100MB so in that case SlimGuard's memory overhead is small compared to the user memory (100MB). Concerning the 64 KB size, one can see that it is close to 100 MB for both SlimGuard and Guarder. The explanation is as follows: 64 KB is a power-of-two so Guarder will also allocate a larger slot for each object (128 KB). However, only the object itself is accessed by our benchmark, which corresponds to the first 64 KB. The last 64 KB starting on a page frontier are not accessed and will not cause any on-demand paging by Linux: this will not generate memory waste. This remark can be generalized to any allocation size above the page size, i.e. 4KB. The number of entropy bits does not seem to generally impact the memory consumption, although one can observe a 10-15% increase with Guarder when going from 2 to 4 bits (only in the 128 B and 1 KB cases).

The execution time of the `malloc` and `free` loops for these benchmarks are presented on Figure 5. SlimGuard is generally better than Guarder, due to its optimized allocation and deallocation paths. `malloc` latency is about 10% faster in SlimGuard for a size of 128 B, and close to 2x faster for 8 KB. `free` latency is also faster for SlimGuard in the 8 KB case. However, it is important to note that because in these tests we always free elements of the same size, `free` in SlimGuard can happen in constant time while it would not be the case with variable sizes (see Section 4.1). Remember that we perform less iterations when the allocation size increases so it is normal that the execution time of both loops decreases with larger allocation sizes. We also observe that the latency of `malloc` is generally longer than the latency of `free`, as malloc includes more operations such as index computation and random number generation. We also determined that the cost of computing the canary represents 5 to 10% of `malloc`'s latency in SlimGuard.

Guard Pages Amount. Guard pages are provided by both SlimGuard and Guarder, and are an efficient way to protect against multiple types of attacks. However, their introduction has a cost because it involves a system call: `mprotect`. We used the same benchmark from the previous experiment and fixed the allocation size to 4095 and the number of iterations to 10 000. We varied the amount of guard pages from 0 (disabled) to 50% (one guard page between every data page) and measured the execution time of our micro-benchmark.

Results are presented on Figure 6. The performance decrease with the amount of guard pages, which is to be expected as both SlimGuard and Guarder offer on-demand guard pages. However SlimGuard manages these pages in a better way as the performance impact is lower, for example it is about 15% faster than Guarder for a ratio of 50%.

Initialization Time. The initialization time is a very important metric in fast-running and latency sensitive programs. In some scenarios (e.g. FaaS), user computations are fast and systems software initialization becomes a bottleneck. Because Guarder allocates a large amount of guard pages at initialization time, we varied the guard page ratio in the same way as the previous experiment to observe its impact. We evaluated

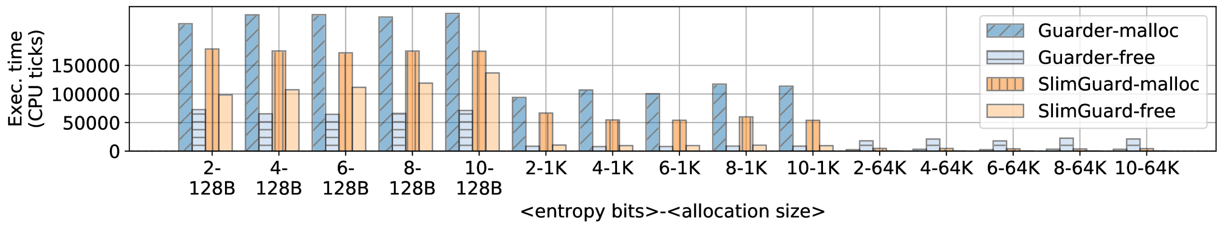

Figure 5. Performance of `malloc` and `free` with different allocation sizes and entropy.

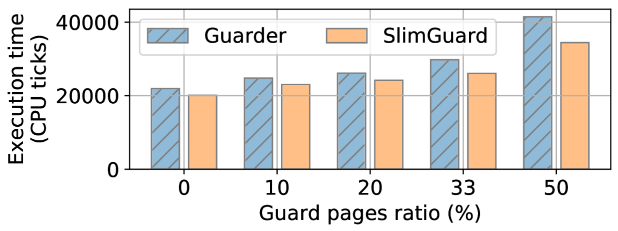

Figure 6. Performance impact of the amount of guard pages.

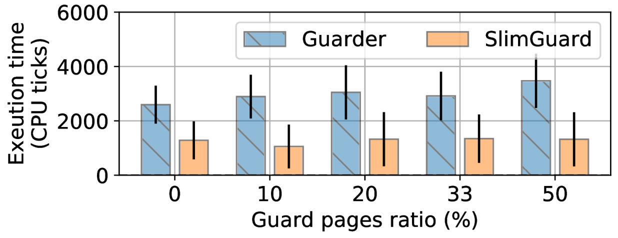

Figure 7. Initialization time for SlimGuard and Guarder.

the initialization time of SlimGuard and Guarder by running a simple program which main function just allocates one byte through `malloc`, frees it and exits. We measured the total execution time through a wrapper using `gettimeofday`. While this time encompasses more than the allocator initialization time, it allows us to capture any overhead happening on-demand. For each run we fix all parameters apart from the allocators so results are comparable.

Results are presented on Figure 7. We noticed a high variation in the results for each run because the programs execute very quickly. Thus each bar is the average value of 50 runs with the standard deviation as error bar. One can observe that SlimGuard's initialization time is about 2x faster than Guarder's. This is due to the fact that most of SlimGuard's data structures are allocated on-demand. Moreover, contrary to Guarder in SlimGuard this latency does not increase with the amount of guard page because they are set on-demand.

Entropy-Based Over-Provisioning As mentioned earlier, such OP is an interesting side effect of our entropy management strategy but it is not as efficient as directly-controlled OP schemes. In order to assess the percentage of entropy-based OP in various situations, we fixed the allocation size to 1KB and measured the percentage of OP obtained while varying the parameters impacting this percentage in SlimGuard: the number of allocations is varied from 1000 to 100000 and

Table 2. OP percentage according to the number of allocation and entropy.

Entropy bits	**8**			**9**			**10**		
Allocations	**1K**	**10K**	**100K**	**1K**	**10K**	**100K**	**1K**	**10K**	**100K**
OP %	12.2	1.34	0.13	25.3	2.67	0.25	51.8	4.95	0.51

the entropy from 8 to 10 bits. To measure this percentage, we observe the data area after the allocations and count the percentage of free slots that are contiguous to a used slot, i.e. the ones that would protect against over/underflows.

Results are presented in Table 2. As one can observe, when the number of allocations is small (1K) , the OP obtained is relatively high: from 12% with 8 bits of entropy to 51.8% with 10 bits. However when the number of allocations is low the OP percentage falls (less than 1% for 100K allocations). In these situations it is possible either to increase the level of entropy (up to 15 bits, giving about 30% of OP for this experiment) or to request additional free slots to complement the ones obtained through entropy-based OP.

6.2 Macro-benchmarks

We measured memory overhead and execution times for the serial versions PARSEC and MiBench suites. For MiBench we only include numbers for the programs that use the heap, namely `jpeg-6a/6b`, `mad`, `typeset`, `dijkstra` and `patricia`. MiBench runs quickly and its execution times are slightly unstable, so we present the average values of 50 runs.

Memory consumption numbers are presented on the top of Figure 8. SlimGuard's memory footprint is always better than Guarder, whose high memory consumption is partially due to the large number of power-of-two-sized allocations made by some benchmarks such as `canneal` (1.7x memory overhead) and several of the MiBench programs. With its fine-grained size classes, SlimGuard has less or no overhead in these cases. The overhead due to power-of-two size classes in allocators such as Guarder can only lead to a maximum of 2x memory consumption (see Section 4.1), but in some benchmarks that overhead is superior: 3x for `dijkstra` and 12x for `swaptions`. These programs have a very dynamic memory usage, i.e. calls to `malloc` and `free` are interlaced as opposed to other benchmarks using only `malloc` at the beginning and `free` at the end of the program. We found out that contrary to SlimGuard, Guarder does not reuse memory thus in such cases (dynamic memory usage), its memory consumption is very high.

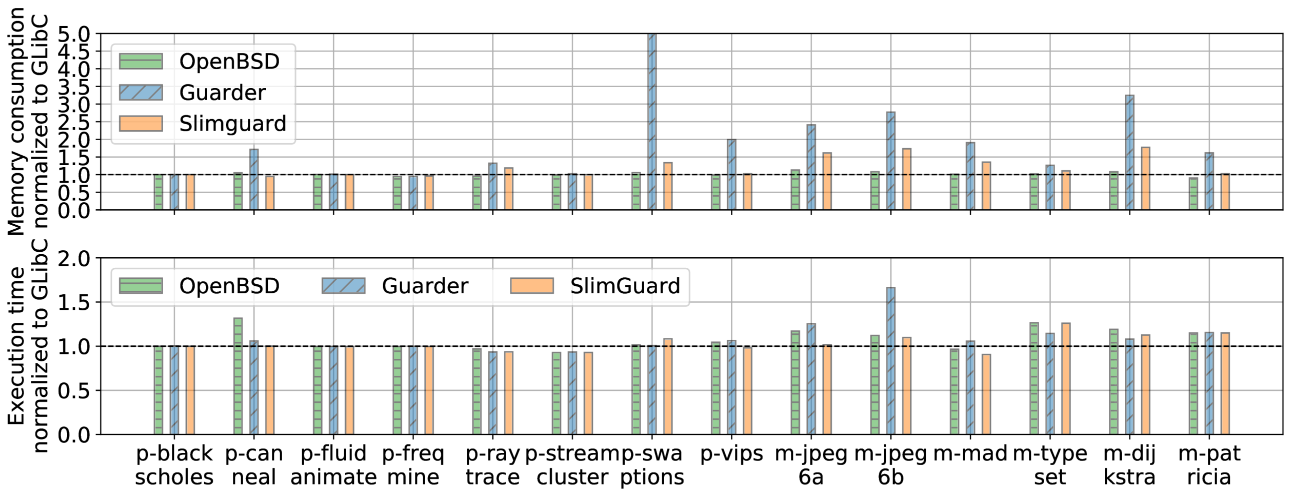

Figure 8. Memory footprint (top) and execution time (bottom) of SlimGuard, OpenBSD and Guarder for PARSEC (`p-*`) and MiBench (`m-*`) macro-benchmarks. All values are normalized to Glibc's memory footprint and execution time.

SlimGuard's overhead is slightly higher than Glibc's for some of the MiBench benchmarks. This is due to a slightly higher per object cost and base memory footprint, combined with the very small memory usage of these applications (1.5 MB for `jpeg-6a`). However that overhead is still low compared to Guarder's. OpenBSD has low memory overhead, similar to Glibc's numbers in most cases. Although SlimGuard's memory overhead is similar to Glibc's in most cases, it is not as memory efficient as OpenBSD. However, OpenBSD's memory efficiency comes at the price of less security guarantees, as well as some performance overhead (for example in `canneal`). Performance numbers are presented on Figure 8. SlimGuard's performance is mostly similar to Glibc, while we can observe some punctual performance drops for Guarder (`jpeg-6b`) and OpenBSD (`canneal`).

We relaunched these macro-benchmarks disabling the memory release feature of SlimGuard and witnessed in most cases a slight performance and memory consumption improvement (less than 5%) due to less `madvise` calls and to the lack of per-page object counters. We also measured that the performance impact of destroy-on-free in these macro-benchmarks for SlimGuard is below 10%. This is because most of the execution is spent doing computations rather than executing `free`.

We present on Figure 9 results for PARSEC running with multiple threads (1 to 8). For space reasons we only present a subset of the benchmarks representative of the general trends. We also do not include results for OpenBSD whose implementation proved to be unstable on these multithreaded programs. Concerning memory consumption, the trends observed on the single threaded version of PARSEC are confirmed. Moreover, for some benchmarks such as `vips` we note that the number of allocations increases with the number of threads. In that case a large amount of these allocations' sizes are powers of two so the overhead of Guarder jumps

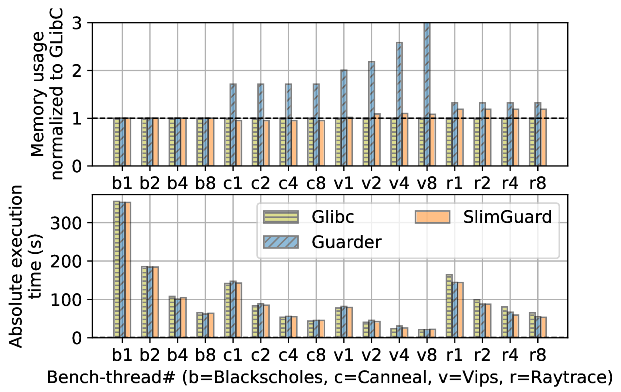

Figure 9. Multithreading memory overhead (top) and performance (bottom) results.

from 2x to more than 3x when going from 1 to 8 threads. SlimGuard's fine-grained size classes management reduces the memory wastage and its overhead compared to Glibc's is constant and negligible. Similar to Guarder SlimGuard performs as well as Glibc. The programs scale and the execution time reduces as more threads are used.

7 Conclusion

Existing secure dynamic memory allocators suffer from either a significant memory overhead, or performance/security concerns. We present SlimGuard, a secure allocator focusing on a low memory footprint while providing a high degree of security and performance. Evaluation shows that its memory overhead is more than 2x smaller than that of the state-of-the-art allocator in a number of macro-benchmarks.

SlimGuard is available at https://ssrg-vt.github.io/SlimGuard/.

Acknowledgments

This work is supported in part by ONR under grants N00014-16-1-2104 and N00014-18-1-2022.

References

[1] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers.. In *USENIX Security Symposium.* 177–192.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (sp 2008).* IEEE, 263–277.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium.* 51–66.

[4] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.. In *USENIX Security Symposium*, Vol. 12. 291–301.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* ACM, 72–81.

[6] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis.* ACM, 133–143.

[7] Marco Carvalho, Jared DeMott, Richard Ford, and David A Wheeler. 2014. Heartbleed 101. *IEEE security & privacy* 12, 4 (2014), 63–67.

[8] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles.* ACM, 153–167.

[9] Yu Ding, Tao Wei, Tielei Wang, Zhenkai Liang, and Wei Zou. 2010. Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks. In *ACSAC.*

[10] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016).* ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[11] Jake Edge. 2009. Linux ASLR Vulnerabilities. https://lwn.net/Articles/330866/.

[12] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada.*

[13] GitHub. 2012. Malloc Implementations. https://github.com/emeryberger/Malloc-Implementations.

[14] GitHub. 2018. Guarder Sources Repository. https://github.com/UTSASRG/Guarder.

[15] Wolfram Gloger. 2006. Ptmalloc. http://www.malloc.de/en/.

[16] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538).* IEEE, 3–14.

[17] Michel Kaempf. 2001. Vudo-an object superstitiously believed to embody magical powers.

[18] Poul-Henning Kamp. 1998. Malloc (3) revisited.. In *USENIX Annual Technical Conference.* 45.

[19] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. 2006. Comprehensively and efficiently protecting the heap. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 207–218.

[20] Doug Lea. 1996. A Memory Allocator. (1996). http://g.oswego.edu/dl/html/malloc.html.

[21] 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.*

[22] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

[23] M. Masmano, I. Ripoll, A. Crespo, and J. Real. 2004. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS '04).* IEEE Computer Society, Washington, DC, USA, 79–86. https://doi.org/10.1109/ECRTS.2004.35

[24] Daniel Micay. 2018. Linux ASLR comparison. https://gist.github.com/thestinger/b43b460cfccfade51b5a2220a0550c35.

[25] Otto Moerbeek. 2009. A new malloc (3) for OpenBSD. In *Proceedings of the 2009 European BSD Conference, EuroBSDCon*, Vol. 9.

[26] National Vulnerability Database. 2017. CVE-2017-0144. https://nvd.nist.gov/vuln/detail/CVE-2017-0144.

[27] NIST. 2019. National Vulnerability Database. https://nvd.nist.gov/.

[28] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10).* ACM, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[29] PaX Team. 2003. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.

[30] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: practical VTable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications.* ACM, 448–459.

[31] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2389–2403.

[32] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A Tunable Secure Allocator. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association, Baltimore, MD, 117–133. https://www.usenix.org/conference/usenixsecurity18/presentation/silvestro

[33] Alexander Sotirov. 2007. Heap feng shui in javascript. *Black Hat Europe* 2007 (2007).

[34] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 48–62.

[35] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 1675–1689.

[36] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers.. In *NDSS.*

[37] Yves Younan, Wouter Joosen, and Frank Piessens. 2006. Efficient protection against heap-based buffer overflows without resorting to magic. In *International Conference on Information and Communications Security.* Springer, 379–398.

[38] Yves Younan, Wouter Joosen, Frank Piessens, and HV den Eynden. 2005. *Security of memory allocators for C and C++.* Technical Report. Technical Report CW 419, Department of Computer Science, Katholieke Universiteit Leuven.