GUARDER: A Tunable Secure Allocator

Sam Silvestro[*] *Hongyu Liu*[*] *Tianyi Liu*[*] *Zhiqiang Lin*[†] *Tongping Liu*[*]
[*]*University of Texas at San Antonio*
[†]*The Ohio State University*

Abstract

Due to the on-going threats posed by heap vulnerabilities, we design a novel secure allocator — GUARDER— to defeat these vulnerabilities. GUARDER is different from existing secure allocators in the following aspects. Existing allocators either have low/zero randomization entropy, or cannot provide stable security guarantees, where their entropies vary by object size classes, execution phases, inputs, or applications. GUARDER ensures the desired randomization entropy, and provides an unprecedented level of security guarantee by combining all security features of existing allocators, with overhead that is comparable to performance-oriented allocators. Compared to the default Linux allocator, GUARDER's performance overhead is less than 3% on average. This overhead is similar to the previous state-of-the-art, FreeGuard, but comes with a much stronger security guarantee. GUARDER also provides an additional feature that allows users to customize security based on their performance budget, without changing code or even recompiling. The combination of high security and low overhead makes GUARDER a practical solution for the deployed environment.

1 Introduction

Vulnerability	Occurrences (#)
Heap Overflow	673
Heap Over-read	125
Invalid-free	35
Double-free	33
Use-after-free	264

Table 1: Heap vulnerabilities reported in 2017.

A range of heap vulnerabilities, such as heap over-reads, heap over-writes, use-after-frees, invalid-frees, and double-frees, still plague applications written in C/C++ languages. They not only cause unexpected program behavior, but also lead to security breaches, including information leakage and control flow hijacking [34]. For instance, the Heartbleed bug, a buffer over-read problem in the OpenSSL cryptography library, results in the leakage of sensitive private data [1]. Another example of a recent buffer overflow is the WannaCry ransomware attack, which takes advantage of a vulnerability inside Server Message Block [17], affecting a series of Windows versions [12]. Heap vulnerabilities still widely exist in different types of in-production software, where Table 1 shows those reported in the past year at NVD [29].

Secure memory allocators typically serve as the first line of defense against heap vulnerabilities. However, existing secure allocators, including the OpenBSD allocator [28] (which we will simply refer to as "OpenBSD"), DieHarder [30], Cling [2], and FreeGuard [33], possess their own strong deficiencies.

First, these allocators provide either low randomization entropy, or cannot support a stable randomization guarantee, which indicates they may not effectively defend against heap overflows and use-after-free attacks. Cling does not provide any randomization, while FreeGuard only provides two bits of entropy. Although OpenBSD and DieHarder supply higher entropy levels, their entropies are not stable, and vary across different size classes, execution phases, inputs, and applications. Typically, their entropies are inversely proportional to an object's size class. For instance, OpenBSD has the highest entropy for 16 byte objects, with as many as 10 bits, while the entropy for objects with 2048 bytes is at most 3 bits. Therefore, attackers may exploit this fact to breach security at the weakest point.

Second, existing allocators cannot easily change their security guarantees, which prevents users from choosing protection based on their budget for performance or memory consumption. For instance, their randomization entropy is primarily limited by bag size (e.g. DieHarder and OpenBSD), or the number of free lists (e.g. FreeGuard). For instance, simply incrementing FreeGuard's entropy by a single bit may significantly increase memory consumption, due to doubling its number of free lists.

Third, existing secure allocators have other problems that may affect their adoption. Both OpenBSD and DieHarder impose large performance overhead, with 31% and 74% on average. Also, they may slow down some applications by 4× and 9× respectively, as shown in Figure 3. This prohibitively high overhead may prevent their adoption in performance-sensitive scenarios. On the other hand, although FreeGuard is very efficient, its low entropy and deterministic memory layout make it an easier target to attack.

This paper presents GUARDER, a novel allocator that provides an unprecedented security guarantee, but without compromising its performance. GUARDER supports all necessary security features of existing secure allocators, and offers the highest level of randomization entropy stably. In addition, GUARDER is also the first secure allocator to allow users to specify their desired security guarantee, which is inspired by tiered Internet services [8].

Existing allocators provide unstable randomization entropies because they randomly select an object from those that remain available within a bag (e.g. OpenBSD), or among multiple bags belonging to the same size class (e.g. DieHarder). However, the number of available objects is reduced with every allocation, unless immediately offset by a deallocation, thus decreasing entropy. Also, their entropies greatly depend on the bag size, which limits the total number of available objects inside. GUARDER *proposes an allocation buffer to track available objects for each size class, then randomly chooses one object from the buffer upon each allocation*. The allocation buffer will be dynamically filled using both new and recently-freed objects on-demand, avoiding this decrease of entropy. The allocation buffer will simultaneously satisfy the following properties: (1) The buffer size can be easily adjusted, where a larger size will provide a higher randomization entropy; (2) The buffer size is defined independently from any size class in order to provide stable entropy for objects of different size classes; (3) It is very efficient to locate an item inside the buffer, even when given an index randomly; (4) It is more efficient to search for an available object by separating available objects from the large amount of in-use ones.

However, although it is possible to place deallocated objects into the allocation buffer directly, it can be very expensive to search for an empty slot in which to do so. In addition, it is difficult to handle a freed object when the allocation buffer is full. Instead, GUARDER *proposes a separate deallocation buffer to track freed objects*: freed objects will be recorded into the deallocation buffer sequentially, which will be more efficient due to avoiding the need for searching; these freed objects will be moved to the allocation buffer upon each allocation, and in a batched mode when the allocation buffer is reduced to half-full. More implementation details are described in Section 4.

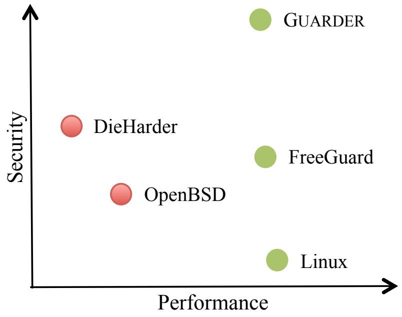

Figure 1: Comparing to performance vs. security of existing work

The combination of allocation and deallocation buffers also seamlessly integrates with other customization mechanisms, such as guard pages and over-provisioning. When filling the allocation buffer with new heap objects, GUARDER maintains a bump pointer that always refers to the next new object at the top of the heap. It will skip all objects tied to randomly-selected guard pages (and set them as non-accessible), and randomly skip objects in proportion to the user-defined over-provisioning factor. This mechanism ensures these skipped objects will never participate in future allocations and deallocations. In contrast, DieHarder is unable to place guard pages within the interior of a bag, since every object has a chance of being allocated in the future. For this same reason, DieHarder may incur a larger memory footprint or additional cache misses.

GUARDER designs multiple mechanisms to further improve its performance. First, it designs a novel heap layout to quickly locate the metadata of each freed object in order to detect double and invalid frees. Second, it minimizes lock acquisitions to further improve scalability and performance. Third, it manages pointers to available objects directly within the allocation buffer, removing a level of indirection compared to existing bitmap-based (e.g. DieHarder or OpenBSD) or free-list-based (e.g. FreeGuard) approaches. GUARDER also overcomes the obvious shortcoming of FreeGuard's deterministic layout by constructing per-thread heaps randomly. Compared to existing work, as shown in Figure 1, GUARDER achieves the highest security, while also imposing small performance overhead.

Overall, GUARDER makes the following contributions.

Supporting a stable and tunable security guarantee. It is the first allocator to support customizable security guarantees on randomization entropy, guard pages, and over-provisioning, which allows users to choose the appropriate security level based on their performance or memory budget. GUARDER implements a combination of allocation and deallocation buffers to support its customizable security.

Supporting the highest degree of security, but with reasonable overhead. GUARDER implements all necessary security features of existing secure allocators, and provides around 9.89 bits of entropy, while only imposing less than 3% performance overhead and 27% memory overhead when compared to the default Linux allocator. GUARDER achieves similar performance overhead to the state-of-the-art (FreeGuard), with less memory overhead, and while substantially improving randomization by providing over 200 times more objects (per each thread and size class) to randomly choose between. For example, where FreeGuard selects one out of ~ 4 objects, GUARDER chooses from over 948 objects.

Substantial evaluation of GUARDER and other secure allocators. The paper performs substantial evaluation of the performance and effectiveness of GUARDER and other existing allocators. Investigations were conducted through direct examination of source code and by performing extensive experiments. GUARDER is the first work to experimentally evaluate the randomization entropy and search trials of existing allocators.

2 Background

2.1 Heap Vulnerabilities

Heap vulnerabilities that can be defended or reduced by GUARDER include buffer overflows, use-after-frees, and double/invalid frees. These memory vulnerabilities can result in information leakage, denial-of-service, illegitimate privilege elevation, or execution of arbitrary code.

A buffer overflow occurs when a program reads or writes outside the boundaries of an allocated object, which further includes buffer underflows. Use-after-free occurs when an application accesses memory that has previously been deallocated, and has possibly been reutilized for other live objects [37, 10, 6]. A double-free problem takes place when an object is freed more than once. Finally, an invalid-free occurs when an invalid pointer is passed to heap deallocation functions.

2.2 Existing Secure Heap Allocators

There are multiple existing secure allocators, including OpenBSD [28], Cling [2], DieHarder [30], and FreeGuard [33]. Among these, Cling is an exception that does not support randomization, the most important feature of secure allocators. Cling only mitigates use-after-free vulnerabilities through constraining memory reuses to objects of the same type.

Based on our understanding, OpenBSD, DieHarder, and FreeGuard share many common design elements. (1) All employ the BIBOP style — “Big Bag of Pages” [14]. For BIBOP-style allocators, one or multiple continuous pages are treated as a “bag” that holds objects of the same size class. The metadata of each heap object, such as its size and availability information, is typically stored in a separate area. Thus, BIBOP-style allocators improve security by avoiding many metadata-based attacks. (2) They all distinguish between the management of “small” and “large” objects, but with different size thresholds. (3) These secure allocators manage small objects using power-of-two size classes. Further, they do not perform object splitting or coalescing, which is different from general purpose allocators, such as the default Linux allocator.

These allocators also have their own unique designs, which are discussed briefly as follows.

OpenBSD. OpenBSD utilizes a bitmap to maintain the status of heap objects, with each bag having a size of 4 kilobytes that is directly allocated from the kernel via an `mmap` system call. For small objects, one out of four lists will be chosen randomly upon each allocation. If no available objects exist in the first bag of the selected list, a new bag is then allocated and added to the current list. Otherwise, an index will be computed randomly, which will serve as the starting point to search for an available object. It will first check the remaining bits of the current bitmap word. If no available objects exist, it will move forward until finding one with available objects. Then, it performs a bit-by-bit search to identify the location of the first available object. For large objects, defined as those larger than 2 kilobytes, OpenBSD maintains a cache of at most 64 pages in order to reduce `mmap` system calls.

DieHarder. In DieHarder, the bag size is initially set to 64 kilobytes, and will be doubled each time a new bag is required. Similarly, a bitmap is used to manage the status of each small object, defined as less than 64 kilobytes, and the same bags may be used to satisfy requests from multiple different threads. DieHarder allocates objects randomly from among the available objects of all bags serving a given size class. If the chosen object is unavailable, it will then compute another random placement. To our understanding, this design may hurt performance (compared to OpenBSD), as it may unnecessarily load bitmap words from different cache lines.

DieHarder utilizes the over-provisional mechanism to help tolerate buffer overflows. A portion of objects will never be allocated; therefore, a bug overflowing into a non-used object will not harm the application.

Large objects will be allocated directly via mmap, with entropy supplied by the underlying OS's ASLR mechanism. Upon deallocation, any accesses to these objects can thus cause a segmentation fault. That is, DieHarder can strongly defend against use-after-free vulnerabilities in large objects.

FreeGuard. FreeGuard is the previous state-of-the-art secure allocator, but contains some compromise to its security guarantee.

It adopts a deterministic layout and utilizes shadow memory to directly map objects to their metadata. While this design avoids search-related overhead on deallocations, it will also sacrifice security, as the mapping between objects and metadata is computable.

FreeGuard implements multiple security mechanisms, such as guard pages and canaries. However, it provides only 2.01 bits of entropy by randomly choosing one-out-of-four free lists (and also rarely from new objects) on allocations.

2.2.1 Problems of Existing Secure Allocators

The problems of these secure allocators are summarized as follows.

Security Guarantee. The following problems exist in these secure allocators. (1) These allocators either have very limited randomization entropy (such as 2.01 bits for FreeGuard), or have unstable entropies that can vary greatly across different size classes, execution phases, executions, and applications. For OpenBSD and DieHarder, their entropies are inversely proportional to size class, and may change during execution or when executed using different inputs. For example, DieHarder's entropy for 1 kilobyte objects falls between 4.8 bits (e.g. bodytrack) and 13.3 bits (e.g. fluidanimate). (2) Their security guarantee is determined by their design, which is difficult to change for different requirements. OpenBSD and DieHarder's entropies are determined by their bag size, while FreeGuard's entropy is determined by its four free lists. (3) FreeGuard's metadata is unprotected, and the relationship between heap objects and metadata is deterministic. Thus, if an attacker were able to modify them, he may take control of the allocator and issue successful attacks afterwards. (4) OpenBSD has very limited countermeasures for protecting large objects (those with sizes larger than 2 kilobytes). Since its cache only maintains a maximum of 64 pages, its entropy should be less than 6 bits if an object can be allocated from the cache.

Performance and Scalability Issues. OpenBSD and DieHarder also have significant performance and scalability issues: (1) Their runtime overhead is too heavy for performance-sensitive applications, with 31% for OpenBSD and 74% for DieHarder (see Section 5.1). Based on our evaluation (as shown in Figure 3), OpenBSD can slow down a program up to 4× (e.g., swaptions), and DieHarder may reduce performance by more than 9× (e.g., freqmine). (2) They have a significant scalability problem, due to utilizing the same heap to satisfy requests from multiple threads [5].

3 Overview

This section discusses the threat model and basic idea of GUARDER.

3.1 Threat Model

Our threat model is similar to many existing works [9, 24]. First, we assume the underlying OS (e.g., Linux) is trusted. However, the ASLR mechanism is not necessarily required to be valid, since GUARDER manages memory allocations using a separate randomization mechanism, making its layout difficult to predict even if ASLR in the underlying OS is broken. Second, we also assume that the platform will use a 64-bit virtual address space, in order to support the specific layout of this allocator.

For the target program, GUARDER assumes the attacker may obtain its source code, such that they may know of possible vulnerabilities within. GUARDER further assumes the attackers have no knowledge related to the status of the heap, and cannot take control of the allocator. They cannot utilize a data leakage channel, such as /proc/*pid*/maps, to discover the location of metadata (in fact, such a leakage channel can be easily disabled). GUARDER also assumes the attackers cannot interfere with the memory management of the allocator, such as by hacking the random generator. Otherwise, they are able to change the order of memory allocations to increase their predictability.

3.2 Basic Idea of Guarder

GUARDER will defend against a wide range of heap vulnerabilities, such as heap overflows, use-after-frees, double and invalid frees, as well as reduce heap spraying attacks.

GUARDER implements almost all security features of existing secure allocators, as listed in Table 2. The only feature disabled by default is destroy-on-free. We argue that this feature is not necessary, since the strong randomization of GUARDER will decrease the predictability of every allocation, which will significantly decrease the exploitability of dangling pointers and makes meaningful information leakage much more difficult [30]. Compared to the state-of-the-art, GUARDER significantly increases randomization (entropy is increased by 7.8 bits, over 200

Security Features	Security Benefit	DieHarder	OpenBSD	FreeGuard	GUARDER
BIBOP style	Defends against metadata-based attacks	✓	✓	✓	✓
Fully-segregated metadata	Defends against metadata-based attacks	✓	✓	✓	✓
Destroy-on-free	Exposes un-initialized reads or use-after-frees	✓	◇	◇	◇
Guard pages	Defends against buffer over-reads and over-writes Defends against heap spraying	⊖	✓	✓	✓
Randomized allocation	Increases attack complexity of overflows and UAFs	✓	✓	✓	✓
Over-provisional allocation	Mitigates harmful effects of overflows	✓			✓
Check canaries on free	Early detection of overflows		⊖	✓	✓
Randomization entropy*	Increases attack complexity	$O(\log N)$	2–10	2.01	E

Table 2: Detailed comparison of security features of existing secure allocators.

✓: allocator has feature ◇: optional feature, disabled by default
⊖: weak implementation ∗: actual results of entropies can be seen in Figure 4

times), adopts the over-provisional mechanism (first proposed by DieHarder), and discards its deterministic layout. Additionally, GUARDER supports customizable security guarantees, without changing code or recompiling, which allows users to specify their desired level of security by setting the corresponding environment variables.

GUARDER, as a shared library, can be preloaded to replace the default allocator, and intercepts all memory management functions of applications automatically. It does not target support for applications with their own custom allocators, although these applications can be changed to use standard memory functions in order to benefit from GUARDER.

GUARDER employs different mechanisms for managing small and large objects, the same as existing secure allocators (described in Section 2.2). GUARDER borrows the same mechanism as DieHarder and FreeGuard for handling large objects, but defines large objects as those larger than 512 kilobytes. The major contribution of GUARDER lies in its management of small objects; in fact, most objects belong to this class, and have a dominant impact on application performance.

The basic idea of the allocator is shown in Figure 2. In order to reduce the performance overhead caused by a high number of `mmap` system calls, GUARDER requests a large block of memory once from the underlying OS to serve as the heap. Then, it divides the heap into multiple per-thread sub-heaps, where each sub-heap will be further divided into a set of bags. GUARDER also organizes objects into power-of-two size classes, starting from 16 bytes and ending with 512KB, and places metadata in a separate location. Each bag will have the same size (e.g., 4GB). Due to the vast address space of 64-bit machines [26, 2], the address space should accommodate all types of applications.

Per-thread design: GUARDER employs a per-thread heap design such that each thread has its own heap segment, and always returns freed objects to the heap belonging to the current thread. There is no need for GUARDER to acquire locks upon allocations and deallocations, which avoids lock acquisition overhead and prevents potential lock contention. FreeGuard, although also using a per-thread heap design, returns freed objects to the original owner thread, thus requiring a lock. This explains why GUARDER has overhead similar to FreeGuard, even with a much stronger security guarantee. However, this design could introduce memory blowup, where memory consumption is unnecessarily increased because freed memory cannot be used to satisfy future memory requests [5]. GUARDER further designs mechanisms to alleviate this problem, as described in Section 4.6.3.

Obfuscating bag order: GUARDER randomizes the order of bags within each per-thread sub-heap. In contrast, FreeGuard places bags in ascending order by their size class, which is very easy to predict. To shuffle the ordering of size classes, GUARDER employs a hash map to manage the relationship between each bag and its metadata. Further, metadata are randomly allocated using `mmap` system calls, rather than using a pre-allocated block, as in FreeGuard.

More importantly, GUARDER introduces separate allocation and deallocation buffers for each size class of each thread, which is a key difference between GUARDER and other secure allocators. This design allows GUARDER to support multiple customizable security features, including the over-provisioning mechanism that neither OpenBSD nor FreeGuard support. This design is further described as follows.

Allocation buffer. Each bag is paired with an allocation buffer that holds the addresses of available objects in the bag. This allocation buffer supports the user-defined entropy: if E is the desired entropy, then allocating an object randomly from 2^E objects will guarantee E bits of entropy. The idea of the allocation buffer is inspired by Stabilizer [11], but with a different design to reduce unnecessary allocations and deallocations, and support customizable securities.

GUARDER designs the allocation buffer as follows: its capacity will be set to 2^{E+1} (not 2^E), and ensures it will never fall below half-full. This design guarantees one

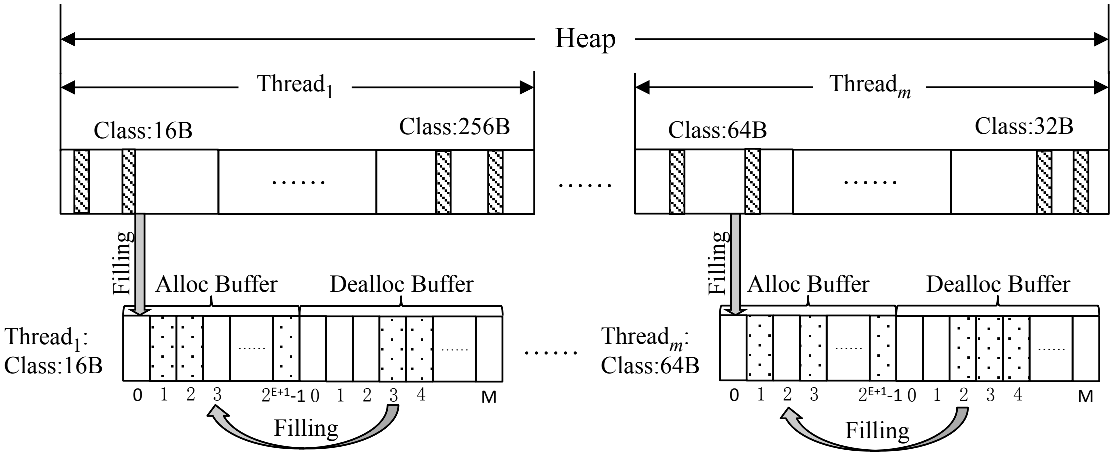

Figure 2: The basic idea of the allocator.

out of at least 2^E objects will be chosen randomly upon each allocation request, and reduces the number of filling operations by using double this size. The allocation buffer will be filled by objects from a separate deallocation buffer, described below, or from new heap objects.

Circular deallocation buffer. GUARDER designs a separate deallocation buffer to track freed objects for a given thread and size class. This design, separating the activities of allocations and deallocations into two different buffers, benefits performance, since freed objects can be recorded sequentially in the deallocation buffer. Because there is no need to search for an available slot, the deallocation step will be completed in constant time.

The allocation buffer will be filled after each allocation if at least one free object exists in the corresponding deallocation buffer. The empty slot created by the allocation will be filled immediately, which helps reduce the number of trials needed to find an available object during allocations. The allocation buffer will also be filled when the number of available objects falls below 2^E, in order to ensure the randomization guarantee. In this case, freed objects from the deallocation buffer will be utilized first, followed by those from a global object buffer. If this is still insufficient, new objects from the associated per-thread heap will be imported. This design helps minimize the number of searches upon each allocation, since the allocation buffer will never be less than half-full. In contrast, OpenBSD and DieHarder may require a large number of searches to identify an available object, ranging between one and several dozen. Table 3 describes the evaluation results for these allocators.

3.2.1 Defending Against Different Attacks

GUARDER defends against heap vulnerabilities by employing the combination of multiple mechanisms.

Defending exploits of buffer overflows. GUARDER can defend against the exploitation of buffer overflows in several ways. First, its strong randomization makes attacks much more difficult, since attackers must know the target chunk addresses at which to issue attacks. When objects are highly randomized, it is extremely difficult to know where an allocation will be satisfied, even if source code is available. Second, over-provisioning may tolerate overflows landing on unused objects, thus nullifying them. Third, guard pages can thwart overflow attempts. Finally, if some attacks modify the canaries placed at the end of each object, GUARDER can detect such attacks.

Defending exploits of use-after-frees. Similarly, GUARDER defends against such exploits in multiple ways. First, GUARDER separates the metadata from the actual heap, making it impossible to issue use-after-free attacks on freelist pointers. Second, its strong randomization makes meaningful attacks extremely difficult, with only a 0.11% success rate per try due to its 9.8 bits of entropy, as evaluated in Section 5.4. Since each subsequent free is a Bernoulli trial following a geometric distribution, it is expected to achieve the first successful attack after 891 tries. Finally, unsuccessful attacks may crash programs incidentally, due to guard pages placed inside, therefore the brute-force approach may not easily succeed.

Defending exploits of double and invalid frees. As discussed above, GUARDER can detect against every double and invalid free, due to its custom allocator. Therefore, GUARDER can choose to stop the program immediately, or skip these invalid operations. Therefore, GUARDER can always defend against such vulnerabilities.

4 Implementation Details

This section describes how GUARDER supports different security mechanisms based on its unique design of allocation and deallocation buffers. Additionally, this section also discusses certain optimizations to further reduce performance overhead and memory blowup.

4.1 Customizable Randomization Entropy

GUARDER supports customizable randomization to meet the various performance and security requirements of different users. As described in Section 3.2, this mechanism is achieved by altering the number of entries in each allocation buffer. Currently, 9 bits of entropy are chosen by default, and GUARDER guarantees that the number of available objects will never be less than 512 (2^9), where each buffer has 1024 entries.

Object selection is performed as follows: upon every allocation, a random index into the allocation buffer is generated. It will then acquire the object address stored at this index, if the object is available. If the index refers to an empty slot (i.e., contains a null value), the allocator will initiate a forward search starting from the selected index. The required number of searches is expected to be around two on average, given the fact that the allocation buffer is never less than half-full. However, this is actually not true due to certain worst cases. Therefore, we divide the allocation buffer into eight separate regions, and record the number of available objects within each. Thus, we can easily skip an entire region if no objects are present.

4.2 Customizable Over-Provisioning

Over-provisioning is a technique in which a certain number of heap objects are designated as never-to-be-used. Therefore, an overflow that occurs in a place containing no useful data can easily be tolerated [30].

GUARDER implements its over-provisioning by controlling the filling step of allocation buffers. For instance, the over-provisioning factor is set to $1/8$ by default, resulting in $1/8$ of objects from each bag being skipped. This also indicates that a given object will be pulled into the corresponding allocation buffer with a likelihood of 87.5%. However, the naive method of computing and comparing probabilities for each object is too expensive. Instead, GUARDER utilizes an associated over-provisional buffer, with a capacity equal to half the allocation buffer, in which new objects from a given bag are first placed. Then, the specified proportion (e.g., $1/8$) of these objects will be deleted from this buffer randomly, and will never participate in future allocations or deallocations. This method reduces the amount of computing and comparing by $7/8$ compared to the naive method.

In contrast to DieHarder, GUARDER's over-provisional mechanism significantly reduces memory footprint and cache loadings, since "skipped" objects will never be accessed in the future. In DieHarder, every object always has a probability of being allocated at some point during the execution. However, accessing these objects may increase the number of physical pages in memory, and involve unnecessary cache loading operations.

4.3 Customizable Guard Pages

GUARDER places guard pages within each bag to thwart overflow or heap spraying attacks. In contrast, DieHarder cannot place guard pages internally, since every heap object has some probability of being utilized. For this reason, DieHarder has a "weak implementation" listed under "Guard Pages" in Table 2, as it cannot stop heap spraying or buffer overflow attacks that only occur within each bag. OpenBSD designs each bag to occupy a single page, which practically places guard pages between bags.

Different from FreeGuard, GUARDER supports a flexible ratio of guard pages, obtained from an environment variable. When pulling from new heap objects during the filling procedure, GUARDER will randomly choose which pages to protect, in proportion to this value. For size classes less than one page, all objects within the page will be protected. If a size class exceeds one page, then multiple pages (equaling the size class) will be protected in order to not change the mapping between objects and their metadata.

4.4 Detecting Double and Invalid Frees

GUARDER can detect double and invalid frees by employing an additional status byte associated with each object. This object status metadata for each bag are located in a separate area. For each allocation, GUARDER marks its status as in-use. Upon deallocation, GUARDER will first compute the index of its status byte, then confirm whether it is an invalid or double-free. If so, it will stop the program immediately; otherwise, it will update the status accordingly. GUARDER can detect all double and invalid frees. Due to complexities brought by `memalign`, GUARDER treats any address within a valid object as a valid free, and consequently frees the object, which is similar to DieHarder.

Note that GUARDER may miss a special kind of double free, similar to existing work [23, 32], when a deallocated object has been subsequently reutilized for other purposes. For example, if a program invokes malloc(V1) → free(V1) → malloc(V2) → free(V1), then the second free(V1) will be considered a valid free operation.

4.5 Checking Canaries on Free

GUARDER also utilizes canaries to help thwart buffer overflow attacks. A single byte placed at the end of every object is reserved for use as a canary. This byte is located beyond the boundary of the size requested by the application. Upon deallocation, this byte's value is inspected; if modified, this serves as evidence of a buffer overflow. Then, GUARDER immediately halts the execution and reports to the user. GUARDER will additionally check the canary values of an object's four adjacent neighbors at the same time, which provides additional protection for long-lived objects that may never be freed by the application.

4.6 Optimizations

GUARDER has made multiple optimizations to further reduce its performance and memory overhead. To this end, GUARDER also employs the Intel SSE2-optimized fast random number generator (RNG) [31, 33].

4.6.1 Accessing Per-Thread Data

GUARDER must access its per-thread heap upon every allocation and deallocation. Therefore, it is critical for GUARDER to quickly access per-thread data. However, the implementation of Thread Local Storage (TLS) (declared using the "`__thread`" storage class keyword) is not efficient [13], and introduces at least an external library call, a system call to obtain the thread ID, and a table lookup. Instead, GUARDER employs the stack address to determine the index of each thread and fetch per-thread data quickly, as existing work [42]. GUARDER allocates a large block of memory that it will utilize for threads' stack areas. Upon thread creation, GUARDER assigns a specific stack area to each thread (e.g., its thread index multiplied by 8MB). Then, GUARDER can obtain the thread index quickly by dividing any stack offset by 8MB.

4.6.2 Reducing Startup Overhead

In order to support a specified randomization entropy, GUARDER needs to initialize each allocation buffer with 2^{E+1} objects, then place the specified ratio of guard pages within. However, some applications may only utilize a subset of size classes, which indicates that the time spent placing guard pages in unused bags is wasted. Therefore, GUARDER employs on-demand initialization: it only initializes the allocation buffer and installs guard pages upon the first allocation request for the bag.

4.6.3 Reducing Memory Consumption

To reduce memory consumption, GUARDER returns memory to the underlying OS when the size of a freed object is larger than 64 kilobytes, by invoking `madvise` with the `MADV_DONTNEED` flag.

GUARDER designs a global deallocation buffer to reduce the memory blowup caused by returning freed objects to the current thread's sub-heap. This problem is extremely serious for producer-consumer applications, since new heap objects would continually be allocated by the producer. If a thread's deallocation buffer reaches capacity, the thread will attempt to donate a portion of its free objects to a global deallocation buffer. Conversely, when a thread has no freed objects in its deallocation buffer, GUARDER will first pull objects from the global deallocation buffer before attempting to utilize new heap objects.

5 Experimental Evaluation

Experiments were performed on a 16-core machine, installed with Intel® Xeon® CPU E5-2640 processors. This machine has 256GB of main memory and 20MB of shared L3 cache, while each core has a 256KB L1 and 2MB L2 cache. The underlying OS is Linux-4.4.25. All applications were compiled using GCC-4.9.1, with `-O2` and `-g` flags.

We utilized the default settings for each allocator, except where explicitly described. By default, GUARDER uses 9 bits of randomization entropy, a 10% proportion of random guard pages, and a $1/8$ over-provisioning factor. OpenBSD's object junking feature was disabled in order to provide a fair comparison.

In order to evaluate the performance and memory overhead of these allocators, we performed experiments on a total of 21 applications, including 13 PARSEC applications, as well as `Apache httpd-2.4.25`, `Firefox-52.0`, `MySQL-5.6.10`, `Memcached-1.4.25`, `SQLite-3.12.0`, `Aget`, `Pfscan`, and `Pbzip2`. Note that Firefox uses an allocator based on `jemalloc` by default, although all figures and tables label it as "Linux" in this section. We did not evaluate single-threaded applications, such as SPEC CPU2006, due to the following reasons. First, multithreaded applications have become the norm, resulting from ubiquitous multicore hardware. Second, DieHarder and OpenBSD have a severe scalability issue, which cannot be observed using single-threaded applications.

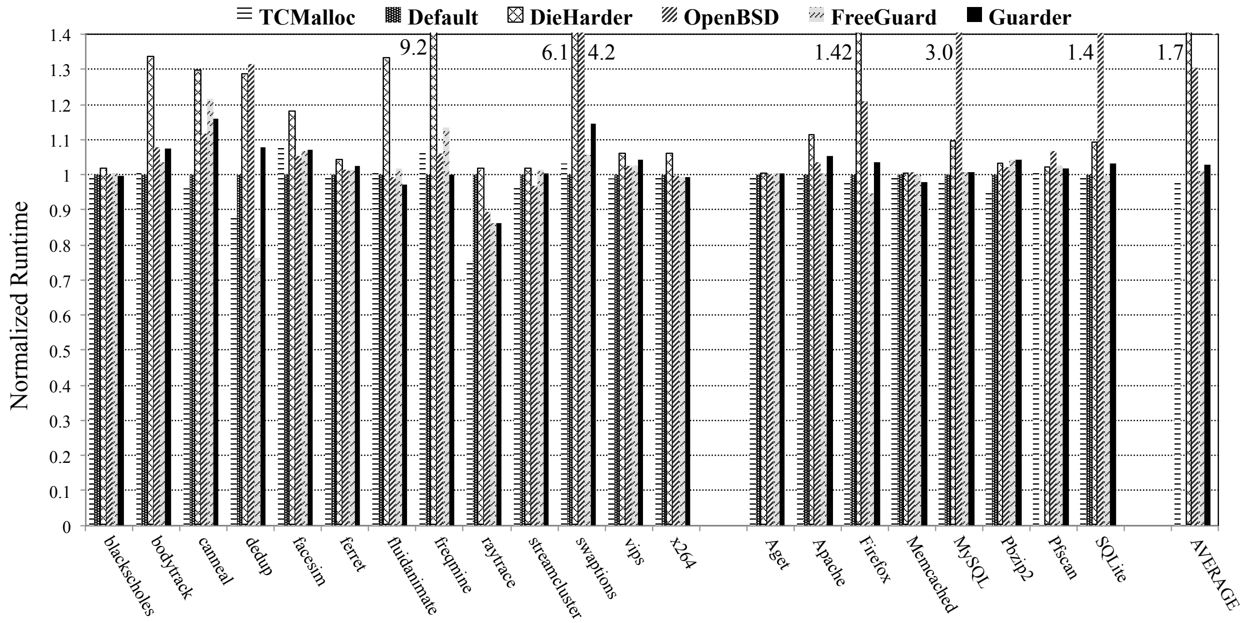

Figure 3: Performance overhead of secure allocators (and TCMalloc), where all values are normalized to the default Linux allocator.

5.1 Performance Overhead

To evaluate performance, we utilized the average results of 10 executions, as shown in Figure 3. DieHarder's destroy-on-free feature was disabled to allow for comparison with GUARDER. A value larger than 1.0 represents a runtime slower than the Linux allocator, while those below 1.0 are faster. On average, the performance overhead of these secure allocators are: DieHarder–74%, OpenBSD–31%, FreeGuard–1%, and GUARDER–3%, by comparing to the Linux allocator, while a known performance oriented allocator—TCMalloc–is slightly faster than it, with 1.6% performance improvement. That is, GUARDER imposes negligible performance overhead, while providing an unprecedented security guarantee. It has performance overhead similar to FreeGuard, but with much higher randomization entropy and support for heap over-provisioning, as evaluated in Section 5.3 and described in Section 6.2.

We further investigated why GUARDER runs faster than DieHarder and OpenBSD, and why it is comparable to FreeGuard. Based on our understanding, two factors can significantly affect the performance of allocators.

System call overhead. The first factor is the overhead of system calls related to memory management. These include `mmap`, `mprotect`, `madvise`, and `munmap`, however, this data was omitted due to space limitations. Based on our evaluation, GUARDER and FreeGuard impose much less overhead from `mmap` system calls, since they obtain a large block of memory initially in order to reduce the number of `mmap` calls. Although they impose more `mprotect` calls, our evaluation indicates that `mprotect` requires only about $1/20$ the time needed to perform an `mmap` system call.

Heap allocation overhead. We also evaluated the overhead associated with heap allocations by focusing on the number of searches/trials performed during allocations and deallocations, as well as the number of synchronizations. An allocator will impose more overhead when the number of searches/trials is larger. Similarly, if the number of synchronizations (mostly lock acquisitions) is larger, the allocator will also impose more overhead.

The average number of trials for each secure allocator is shown in Table 3, where the Linux allocator and TCMalloc typically only require a single trial upon each allocation and deallocation. These values were computed by dividing the total number of trials by the number of allocations or deallocations. For both allocations and deallocations, FreeGuard only requires a single trial due to its free-list-based design. In comparison, GUARDER makes random selections from allocation buffers that are consistently maintained to remain at least half-full. As a consequence, GUARDER's average number of allocation "tries" is about 1.77. Both OpenBSD and DieHarder exceed this value, with 3.79 and 1.99 times respectively. For each deallocation, DieHarder performs 12.4 trials, while OpenBSD, FreeGuard, and GUARDER only require a single trial. Based on our understanding, the large number of trials is a major reason why DieHarder per-

Trials		DieHarder	OpenBSD	FreeGuard	GUARDER
Allocation	Average	1.99	3.79	1	1.77
	Maximum	93	45	1	131
Deallocation	Average	12.40	1	1	1
	Maximum	141	1	1	1

Table 3: Number of trials for allocations and deallocations in different allocators.

Entropy (bits)				*GPR=10%, OPF=1/8*
8	9	10	11	12
1.003	1.000	1.016	1.031	1.047
Guard Page Ratio				*EB=9, OPF=1/8*
2%	5%	10%	20%	50%
0.987	0.990	1.000	1.016	1.046
Over-provisioning Factor				*EB=9, GPR=10%*
1/32	1/16	1/8	1/4	1/2
0.998	0.995	1.000	1.001	1.011

Table 4: Performance sensitivity to each parameter, normalized to the default settings of GUARDER. *EB = Entropy Bits, GPR = Guard Page Ratio, OPF = Over-Provisioning Factor*

forms much worse than other secure allocators. During each deallocation, DieHarder will compare against all existing minibags one-by-one to locate the specific minibag (and mark its bit as free inside), loading multiple cache lines unnecessarily. GUARDER utilizes a special design (see Figure 2) to avoid this overhead. Although DieHarder has less allocation trials than OpenBSD, its worse case is significantly worse than that of OpenBSD.

Synchronization overhead can be somehow indicated by the number of allocations, as shown in Table 5. For all other secure allocators, such as DieHarder, OpenBSD, and FreeGuard, each allocation and deallocation should acquire a lock, although FreeGuard will have less contention. In comparison, GUARDER avoids most lock acquisitions by always returning freed objects to the current thread's deallocation buffer. GUARDER only involves lock acquisitions when using the global deallocation buffer, employed to reduce memory blowup (described in Section 4.6.3). This indicates that GUARDER actually imposes less synchronization overhead than FreeGuard, which is part of reason why GUARDER has a similar overhead to FreeGuard, while providing a much higher security guarantee.

5.2 Performance Sensitivity Studies

We further evaluated how sensitive GUARDER's performance is to different customizable allocation parameter, such as the randomization entropy, the proportion of each bag dedicated to random guard pages, and the level of heap over-provisioning. The average results of all applications were shown in Table 4, where the data is normalized to that of the default setting: 9 bits of randomization entropy, 10% guard pages, and 1/8 of over-provisioning factor.

Randomization Entropy. Different randomization entropies were evaluated, ranging from 8 to 12 bits. As shown in Table 4, a higher entropy, indicating it is harder to be predicted and more secure, typically implies a higher performance overhead. For instance, 12 entropy bits may impose 4.7% performance overhead when comparing to the default setting. With a higher entropy, deallocated objects have a lower chance to be re-utilized immediately, which may access more physical memory unnecessarily, causing more page faults and less cache hits.

Guard Page Ratio. A higher ratio of guard pages will have a higher chance to stop any brute-force attacks. The performance effects of different ratios of random guard pages, including 2%, 5%, 10%, 20%, and 50%, were similarly evaluated. For the 50% ratio, almost every page (or object with size greater than 4 kilobytes), will be separated by a guard page. Similarly, a larger ratio of installed guard pages typically implies a larger performance overhead, due to invoking more `mprotect` system calls.

Over-provisioning factor. Different heap over-provisioning factors, including 1/32, 1/16, 1/8, 1/4, and 1/2, were evaluated. In the extreme case of 1/2, half of the heap will not be utilized. This evaluation shows two results: (1) A larger over-provisioning factor will typically imply larger overhead. (2) The performance impact of over-provisioning is not as large as expected, as over-provisioning will not affect cache utilization when skipped objects are completely removed from future allocations and deallocations. However, it may cause a much larger performance impact on DieHarder, due to its special design.

5.3 Memory Overhead

We collected the maximum memory consumption for all five allocators. For server applications, such as `MySQL` and `Memcached`, memory consumption was collected via the `VmHWM` field of `/proc/pid/status` file. For other applications, memory consumption was collected using the `maxresident` output of the `time` utility [22].

To ensure a fair comparison, we disabled the canary checking functionality for both FreeGuard and GUARDER (and is disabled by default in OpenBSD), since adding even a single-byte canary may cause an object to be allocated from the next largest size class.

In total, the memory overhead (shown in Table 5) of FreeGuard is around 37%, while DieHarder and OpenBSD feature slightly less memory consumption than the Linux allocator, with -3% and -6%, respectively. GUARDER imposes 27% memory overhead on evaluated applications, when using the default 9 bits of entropy. It especially imposes more than 4× memory overhead for `Swaptions`, `MySQL`, and `SQLite`.

GUARDER's memory overhead on certain applications can be attributed to multiple reasons, mostly relating to its management of small objects. First, GUARDER may increase its memory consumption due to its randomized allocation. For any given size class, GUARDER will place more than 2^n objects into its allocation buffer, then randomly allocate an object from among them. Therefore, GUARDER may access other pages (due to its randomized allocation policy) when there are still available/free objects in existing pages. Second, GUARDER's over-provisional mechanism will introduce more memory consumption, since some objects will be randomly skipped and thus never utilized. Note that GUARDER also achieves comparable average memory overhead to FreeGuard, due to its global free cache mechanism, which better balances memory usage among threads (particularly for producer-consumer patterns).

We also observe that GUARDER's memory overhead is near 0% when 7 bits of entropy are utilized. This further indicates the necessity to provide customizable security, as users may choose a lower entropy to reduce performance and memory consumption as needed.

5.4 Randomization Entropy

We further evaluated the randomization entropies of these secure allocators, with results shown in Figure 4. We are the first work that experimentally evaluates the entropies of each size class, by explicitly modifying these allocators. The basic idea is to update a per-size-class global variable upon each allocation, then compute the average entropy of each size class for different applications. We computed the entropy based on the maximum number of available choices upon each allocation using a $\log_2(N)$ formula. Note that we utilized the maximum number of entries in four bags to compute the entropy for OpenBSD upon each allocation. Because the bag size for OpenBSD is just one page, we do not show its entropies for objects larger than 4 kilobytes.

Both DieHarder and OpenBSD were seen to exhibit unstable entropy, and FreeGuard shows a constant low entropy (approximately 2 bits). By contrast, GUARDER's measured entropy is 9.89 bits for every size class, when the specified entropy is set to 9 bits. Taking the size class of 64 kilobytes for example, GUARDER will randomly allocate one object from over 831 objects, while DieHarder and FreeGuard will allocate from just 32 and 4 objects, respectively. This clearly indicates that GUARDER has significantly higher security than these existing allocators. DieHarder only exceeds GUARDER's entropy in the first four size classes, when compared to its default configuration with 9 bits. However, our evaluation also shows that GUARDER guarantees virtually the same high entropy across different size classes, execution phases, applications, or inputs, making it the first secure allocator of this kind.

5.5 Effectiveness of Defending Against Attacks

We evaluate the effectiveness of GUARDER and other allocators using a collection of real-world vulnerabilities, including buffer over-writes, buffer over-reads, use-after-frees, and double/invalid frees. With the exception of Heartbleed, each of the reported bugs will typically result in a program crash. Heartbleed is unique in that it results in the silent leakage of heap data. GUARDER was shown to avoid the ill effects of these bugs, and/or report their occurrences to the user, as shown in Table 6. More information about these buggy applications is described below.

bc-1.06. *Arbitrary-precision numeric processing language interpreter*
The affected copy of this program was obtained from BugBench [25], and includes a buffer overflow as the result of an off-by-one array indexing error, caused by a specific bad input, which will produce a program crash. Based on their powers-of-two size classes, each secure allocator places the affected array in a bag serving objects larger than the needed size. As such, this small one-element overflow is harmlessly contained within unused space, thus preventing the crash.

ed-1.14.1. *Line-oriented text editor*
ed contains a simple invalid-free bug, caused by a call to `free()` that was forgotten by the developer after moving a buffer from dynamic to static memory. GUARDER guarantees detection of all double/invalid free problems, and thus provides an immediate report of the error, including the current callstack.

gzip-1.2.4. *GNU compression utility*
Gzip, obtained from BugBench [25], contains a stack-based buffer overflow. For testing purposes, it was

Application	Allocations (#)	Deallocations (#)	Memory Usage (MB)				
			Linux	DieHarder	OpenBSD	FreeGuard	Guarder
blackscholes	18	14	627	634	628	630	655
bodytrack	424519	424515	34	42	32	63	111
canneal	30728189	30728185	963	1153	828	932	1186
dedup	4045531	1750969	1684	1926	1020	2693	1474
facesim	4729653	4495883	327	377	324	374	491
ferret	137968	137960	66	94	71	100	132
fluidanimate	229992	229918	213	270	235	237	477
freqmine	456	347	1543	1344	1426	1631	1885
raytrace	45037352	45037316	1162	1724	1111	1511	1770
streamcluster	8908	8898	111	114	111	117	149
swaptions	48001811	48000397	6	12	7	12	383
vips	1422138	1421738	32	37	32	820	104
x264	71120	71111	491	506	497	494	604
Aget	49	24	69	59	32	51	82
Apache	102216	101919	4	5	2	6	12
Firefox	20874509	20290076	159	163	169	163	172
Memcached	7601	76	6	8	4	7	13
MySQL	491544	491433	126	135	277	158	535
Pbzip2	67	61	97	102	99	261	105
Pfscan	51	15	753	800	837	803	798
SQLite	1458486	1458447	41	64	35	125	331
Normalized Total			1.00	0.97	0.94	1.37	1.27

Table 5: The number of allocations, deallocations, and memory usage of secure allocators.

moved to the heap. This bug would normally corrupt the adjacent metadata, however, when testing each secure allocator, this crash is avoided due to their metadata segregation. Additionally, around 10% of Guarder and FreeGuard tests resulted in halting execution, caused by accessing an adjacent random guard page.

Libtiff-4.0.1. *TIFF image library*
A malformed input will cause the affected version of Libtiff's `gif2tiff` converter tool to experience a buffer overflow, normally resulting in a program crash. When verifying this bug with Guarder, this will always result in (1) an immediate halt due to illegal access on an adjacent random guard page, or (2) a report to the user indicating the discovery of a modified canary value. OpenBSD aborts with a "chunk info corrupted" error, while DieHarder produces no report and exits normally.

Heartbleed. *Cryptographic library*
The Heartbleed bug exploits a buffer over-read in OpenSSL-1.0.1f. Both Guarder and FreeGuard will probabilistically guard against this attack, with protection in proportion to the amount of random guard pages installed. By default, this is 10%. Neither OpenBSD nor DieHarder can provide protection against this bug.

PHP-5.3.6. *Scripting language interpreter*
A variety of malicious XML data are provided as input, resulting in use-after-free and double-free conditions. Guarder, FreeGuard, and OpenBSD halt and report each of these bugs, while DieHarder exits normally with no report made.

polymorph-0.4.0. *File renaming utility*
The affected version of polymorph suffers from a stack-based buffer overflow that was adapted to the heap for testing purposes, and results in a program crash due to corrupted object metadata. Due to their segregated metadata, all of the secure allocators allow the application to exit normally. However, both Guarder and FreeGuard also provide probabilistic protection in proportion to the amount of installed random guard pages.

Squid-2.3. *Caching Internet proxy server*
Squid 2.3 contains a heap-based buffer overflow caused by an incorrect buffer size calculation. Normally, this bug will cause the program to crash due to corrupting adjacent metadata. When tested with Guarder, the overwritten canary value at the site of the overflow is detected, and the program is immediately halted. FreeGuard exhibits similar behavior, while OpenBSD and DieHarder do not detect the overflow at all.

Summary. For all evaluated bugs, Guarder was capable of either probabilistically detecting the attack – such as through the use of random guard pages to thwart buffer

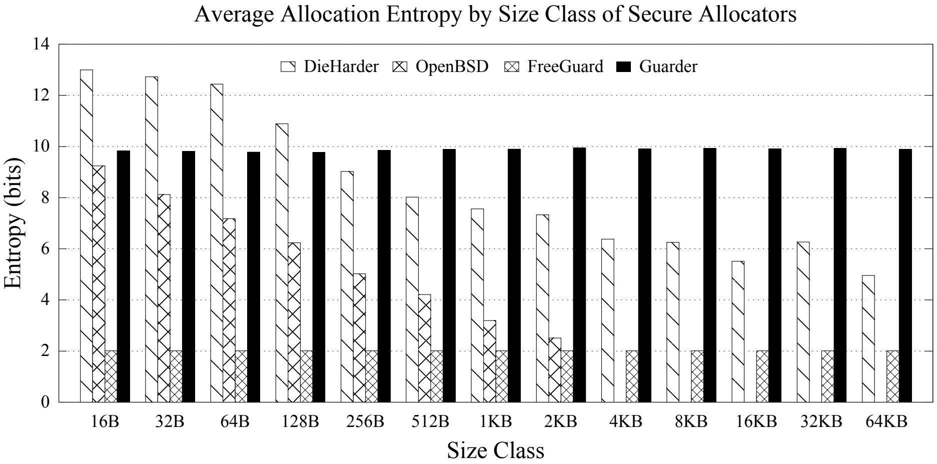

Figure 4: Average randomization entropies of existing secure allocators, grouped by object size class. GUARDER provides a consistently high entropy which other allocators cannot support.

Application	Vulnerability	Original	DieHarder	OpenBSD	FreeGuard	GUARDER
bc-1.06	Buffer Over-write	Crash	No crash	No crash	No crash	No crash
ed-1.14.1	Invalid-Free	Crash	No crash	Halt→report	Halt→report	Halt→report
gzip-1.2.4	Buffer Over-write	Crash	No crash	No crash	*p*-protect	*p*-protect
Heartbleed	Buffer Over-read	Data Leak	Data Leak	Data Leak	*p*-protect	*p*-protect
Libtiff-4.0.1	Buffer Over-write	Crash	No crash	Crash	Halt→report	Halt→report
PHP-5.3.6	Use-After-Free Use-After-Free Double-Free	Crash Crash Crash	No crash No crash No crash	Halt→report Halt→report Halt→report	Halt→report Halt→report Halt→report	Halt→report Halt→report Halt→report
polymorph-0.4.0	Buffer Overflow	Crash	No crash	No crash	*p*-protect	*p*-protect
Squid-2.3	Buffer Overflow	Crash	No crash	No crash	Halt→report	Halt→report

No crash:	Program completes normally	*Data Leak*:	Leakage of arbitrary heap data occurred
Halt→report:	Halts execution & reports to user	*p-protect*:	Probabilistic protection, p = 0.10 (default)

Table 6: Effectiveness evaluation on known vulnerabilities.

overflow – or immediately provided a report to the user when the error condition occurred (e.g., double-free). However, we also noticed that the results of GUARDER and FreeGuard are very similar. Based on our investigation, these evaluated bugs (mostly static) cannot show the benefit of the improved security of GUARDER, as described in Section 6.2, such as higher entropy and over-provisioning. For instance, it is not easy to evaluate higher randomization entropy providing more resistance to attacks, but in reality it does. Additionally, for example, if a one-element overflow is already contained within unused space, over-provisioning provides no additional benefit.

6 Discussion

6.1 Customization

(a) Why is Customization Helpful? GUARDER is the first allocator that supports customizable security. Based on our evaluation (see Section 5), higher security comes at the cost of increased performance overhead and memory consumption. Sometimes, this difference could be sufficiently large that it may affect users' choices. For instance, GUARDER's memory overhead using 7 bits of entropy is around 0% (not shown due to space limitations), while its memory overhead with 9 bits is around 27%. Therefore, users may choose a level of security that reduces memory consumption when required by resource-constrained environments, such as mobile phones. GUARDER provides this flexibility, without the requirement of changing and recompiling applications and the allocator.

(b) How many bits of entropy could GUARDER *support?* Currently, GUARDER supports up to 16 bits of entropy on machines with 48 address bits, in theory, although with the potential for higher overhead. In the current design, as shown in Figure 2, the number of supported threads may limit entropy choices, since there are 16 bags in each thread, and every bag has the same size.

If there are 128 threads in total, with a heap space of 128 terabytes, every bag will be 64 gigabytes, which can support up to 16 bits of entropy. Since there is room for at most 2^{17} objects of size 512 kilobytes in such a bag, it may only support 16 bits of entropy if over-provisioning and guard pages are also supported. In the future, we plan to allocate each bag on-demand, and may use different bag sizes, in order to support even higher levels of entropy.

6.2 Comparison with FreeGuard

In this section, we compare GUARDER with the current state-of-the-art secure allocator FreeGuard. On average, GUARDER imposes around 3% performance overhead and 27% memory overhead, while FreeGuard imposes around 1% performance overhead and 37% memory overhead.

However, GUARDER supports more security features and a higher level of entropy, due to its unique and novel design as described in Section 4: (1) GUARDER supports heap over-provisioning, which FreeGuard does not. This indicates that some buggy applications that may be attacked when using FreeGuard can be avoided with GUARDER. (2) Under the same overhead, GUARDER supports around 9.89 bits of entropy, which is more than 200 times that of FreeGuard. (3) GUARDER further randomizes the order of bags within each per-thread heap, while FreeGuard's deterministic layout is much easier to attack. (4) More importantly, GUARDER allows users to configure their desired security through entropy, guard page ratio, and over-provisional factors, which FreeGuard cannot support.

7 Related Work

Apart from the secure allocators previously examined in Section 2, several other works attempt to solve heap-related security problems, though often choosing to target only a particular class of vulnerability.

7.1 Allocators Protecting Object Metadata

Multiple allocators aim to secure object metadata. Robertson et al. utilize the placement of canary and checksum values, which will be relied upon to warn of potential buffer overflow. Younan et al. achieve fully-segregated metadata by incorporation of a hash table used to maintain their mappings [41]. Heap Server proposes the separation of memory management functions to a separate process, isolating the actual heap data in a different address space than its associated metadata [19].

dnmalloc dedicates a separately allocated area to house object metadata, and also utilizes a table to maintain mappings between these chunks and their metadata, an approach that is not unlike that of DieHarder or OpenBSD [40]. The metadata segregation achieved by these works can protect against metadata-based vulnerabilities, however, they cannot guard against attacks on the actual heap.

Blink, a rendering engine for the Chromium project, utilizes PartitionAlloc, a partition-based allocator with built-in exploit mitigations [15]. While PartitionAlloc provides a general allocator class suitable for supporting multithreaded applications, it is primarily optimized for single-threaded usage. It also lacks key protections offered by secure allocators, such as randomization. Lastly, its design could be significantly hardened; for example, its rudimentary detection of double/invalid frees, and free list pointers that occupy deallocated slots [16]. By comparison, GUARDER guarantees to detect all invalid/double frees, and fully segregates object metadata.

7.2 Protection Utilizing Compiler Instrumentation

Some works attempt to introduce randomness into the memory layout or allocation functions. Bhatkar et al. propose the concept of "address obfuscation", in which the address space is randomized [7]. Kharbutli further describes securing the sequence in which freed objects are reused, in an effort to introduce non-determinism to allocation functions [19]. GUARDER provides a higher entropy than these systems.

The reliance on managing additional metadata to guard against problems at runtime has been employed by many techniques toward increased security. These problems include protection against overflows through the validation of array accesses [3, 4], as well as performing type-checking of variable casting operations [21].

FreeSentry [39] also utilizes compiler instrumentation, but toward protecting against use-after-free problems. This is achieved by recording the application's use of pointer values, updating their status after the target objects have been freed. DangNULL similarly targets use-after-free and double-free vulnerabilities by tracking each pointer, nullifying it when the object it references is deallocated [20]. FreeSentry incurs approximately 25% performance overhead on average, while DangNULL ranges from 22% to 105%. DangSan utilizes a new lock-free design to reduce performance overhead, only introducing half the overhead of FreeSentry and DangNULL [36]. However, they cannot support the randomization of memory allocations.

Iwahashi describes a signature-based approach to detect and identify the cause of these and potentially other vulnerabilities [18]. Cabellero et al. describe Undangle, a runtime approach for detecting use-after-free vulnerabilities through the use of object labeling and tracking, which helps discover dangling pointers [10].

Rather than protecting against a single type of memory error, GUARDER defends against many common errors, achieving this with very little overhead on average. The GUARDER heap combines protections similar to those provided by the mechanisms introduced by these works, including fully-segregated metadata, randomized object reuse, and detection of double/invalid free vulnerabilities, among others.

The Low Fragmentation Heap (LFH) is a widely deployed heap policy for Windows-based platforms, introduced in Windows XP [27]. When enabled, LFH will utilize a bucketing scheme to fulfill similarly sized allocations from larger pre-allocated blocks. LFH is applied for objects of size 16 kilobytes or less, and its 128 buckets span five size classes of varying granularity. The LFH utilizes guard pages, randomization, and encoding of metadata pointers in order to add security to the heap. However, LFH has only 5 bits of entropy for new heap placement, as well as object selection [35, 38]. Furthermore, these entropy values are fixed, unlike those provided by GUARDER.

Apple's MacOS X operating system utilizes a scalable zone allocator from which to fulfill requests from the user-facing malloc layer. While this allocator has seen recent updating for multithreading improvements based on Hoard [5], Mac OS X is significantly lacking in memory security features as compared to other current operating systems [43]. For example, guard pages, segregated metadata, and randomization, are not incorporated. While metadata header checksums are present, they are merely intended to detect accidental corruption, rather than intentional, and can be easily bypassed.

7.3 Employing the Vast Address Space

Archipelego [26] randomly places objects throughout the vast 64-bit address space in order to trade the address space for security and reliability. Thus, the probability of overflowing real data can be effectively reduced. Cling also utilizes the vast address space to tolerate use-after-free problems [2].

8 Conclusion

This paper introduced GUARDER, a novel secure allocator that provides an unprecedented security guarantee among all existing secure allocators. GUARDER proposes the combination of allocation and deallocation buffers to support different customizable security guarantees, including randomization entropy, guard pages, and over-provisioning. Overall, GUARDER implements almost all security features of other secure allocators, while only imposing 3% performance overhead, and featuring comparable memory overhead.

Acknowledgments

We thank the anonymous reviewers for their invaluable feedback. We also thank Xueling Zhang for her participation in the evaluation. This material is based upon work supported by the National Science Foundation under Award CCF-1566154, CCF-1617390, CNS-1812553, CNS-1834215, AFOSR award FA9550-14-1-0119, and ONR award N00014-17-1-2995. The work is also supported by Mozilla Research Grant and Google Faculty Award.

References

[1] *Heartbleed*, 2014.

[2] AKRITIDIS, P. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 12–12.

[3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 263–277.

[4] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 51–66.

[5] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ACM Press, pp. 117–128.

[6] BERGER, E. D., AND ZORN, B. G. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 158–168.

[7] Bhatkar, E., Duvarney, D. C., and Sekar, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium* (2003), pp. 105–120.

[8] Bhatti, N., and Friedrich, R. Web server support for tiered services. *Netwrk. Mag. of Global Internetwkg. 13*, 5 (Sept. 1999), 64–71.

[9] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., and Boneh, D. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 227–242.

[10] Caballero, J., Grieco, G., Marron, M., and Nappa, A. 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* (New York, NY, USA, 2012), ISSTA 2012, ACM, pp. 133–143.

[11] Curtsinger, C., and Berger, E. D. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 219–228.

[12] CVE. Cve-2017-0144. `https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144`.

[13] Gross, D. TLS performance overhead and cost on gnu/linux. `http://david-grs.github.io/tls_performance_overhead_cost_linux`, 2016.

[14] Hanson, D. R. A portable storage management system for the icon programming language, 1980.

[15] Inc., G. Partitionalloc design. `https://chromium.googlesource.com/chromium/src/+/lkgr/base/allocator/partition_allocator/PartitionAlloc.md`.

[16] Inc., G. Partitionalloc source. `https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/PartitionAlloc.h`.

[17] Islam, A., Oppenheim, N., and Thomas, W. Smb exploited: Wannacry use of "eternalblue". `https://www.fireeye.com/blog/threat-research/2017/05/smb-exploited-wannacry-use-of-eternalblue.html`, 2017.

[18] Iwahashi, R., Oliveira, D. A., Wu, S. F., Crandall, J. R., Heo, Y.-J., Oh, J.-T., and Jang, J.-S. Towards automatically generating double-free vulnerability signatures using petri nets. In *Proceedings of the 11th International Conference on Information Security* (Berlin, Heidelberg, 2008), ISC '08, Springer-Verlag, pp. 114–130.

[19] Kharbutli, M., Jiang, X., Solihin, Y., Venkataramani, G., and Prvulovic, M. Comprehensively and efficiently protecting the heap. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 207–218.

[20] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W. Preventing use-after-free with dangling pointers nullification. In *NDSS* (2015).

[21] Lee, B., Song, C., Kim, T., and Lee, W. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), SEC'15, USENIX Association, pp. 81–96.

[22] Linux Comunity. *time - time a simple command or give resource usage*, 2015.

[23] Liu, T., Curtsinger, C., and Berger, E. D. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 911–922.

[24] Lu, K., Song, C., Lee, B., Chung, S. P., Kim, T., and Lee, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 280–291.

[25] Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., and Zhou, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[26] Lvin, V. B., Novark, G., Berger, E. D., and Zorn, B. G. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 115–124.

[27] MICROSOFT. Software defense: mitigating heap corruption vulnerabilities. https://blogs.technet.microsoft.com/srd/2013/10/29/software-defense-mitigating-heap-corruption-vulnerabilities/.

[28] MOERBEEK, O. A new malloc(3) for openbsd. https://www.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf, 2009.

[29] NIST. National vulnerability database.

[30] NOVARK, G., AND BERGER, E. D. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.

[31] OWENS, K., AND PARIKH, R. Fast random number generator on the intel® pentium® 4 processor. https://software.intel.com/en-us/articles/fast-random-number-generator-on-the-intel-pentiumr-4-processor/, March 2012.

[32] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 28–28.

[33] SILVESTRO, S., LIU, H., CROSSER, C., LIN, Z., AND LIU, T. Freeguard: A faster secure heap allocator. In *Proceedings of "24th ACM Conference on Computer and Communications Security (CCS'17)"*.

[34] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 48–62.

[35] VALASEK, C. Understanding the low fragmentation heap, 2010.

[36] VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 405–419.

[37] WIKIPEDIA. Dangling pointer. https://en.wikipedia.org/wiki/Dangling_pointer, September 2016. Last updated: September 1, 2016.

[38] YASON, M. Windows 10 segment heap internals. https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals-wp.pdf, 2016.

[39] YOUNAN, Y. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS* (2015).

[40] YOUNAN, Y., JOOSEN, W., AND PIESSENS, F. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the 8th International Conference on Information and Communications Security* (Berlin, Heidelberg, 2006), ICICS'06, Springer-Verlag, pp. 379–398.

[41] YOUNAN, Y., YOUNAN, Y., JOOSEN, W., JOOSEN, W., PIESSENS, F., PIESSENS, F., EYNDEN, H. V. D., AND EYNDEN, H. V. D. Security of memory allocators for c and c++. Tech. rep., 2005.

[42] ZHOU, J., SILVESTRO, S., LIU, H., CAI, Y., AND LIU, T. Undead: A featherweight deadlock detection and prevention system for production software. In *the submission of "the 39th International Conference on Software Engineering (ICSE'17)"*.

[43] ZOVI, D. Mac os xploitation. https://papers.put.as/papers/macosx/2009/D1T1-Dino-Dai-Zovi-Mac-OS-Xploitation.pdf, 2009.