FreeGuard: A Faster Secure Heap Allocator

Sam Silvestro
University of Texas at San Antonio
Sam.Silvestro@utsa.edu

Hongyu Liu
University of Texas at San Antonio
liuhyscc@gmail.com

Corey Crosser
United States Military Academy
Corey.Crosser@usma.edu

Zhiqiang Lin
University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Tongping Liu
University of Texas at San Antonio
Tongping.Liu@utsa.edu

ABSTRACT

In spite of years of improvements to software security, heap-related attacks still remain a severe threat. One reason is that many existing memory allocators fall short in a variety of aspects. For instance, performance-oriented allocators are designed with very limited countermeasures against attacks, but secure allocators generally suffer from significant performance overhead, e.g., running up to 10× slower. This paper, therefore, introduces FreeGuard, a secure memory allocator that prevents or reduces a wide range of heap-related attacks, such as heap overflows, heap over-reads, use-after-frees, as well as double and invalid frees. FreeGuard has similar performance to the default Linux allocator, with less than 2% overhead on average, but provides significant improvement to security guarantees. FreeGuard also addresses multiple implementation issues of existing secure allocators, such as the issue of scalability. Experimental results demonstrate that FreeGuard is very effective in defending against a variety of heap-related attacks.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; *Operating systems security*; • **Software and its engineering** → *Allocation / deallocation strategies*;

KEYWORDS

Memory Safety; Heap Allocator; Memory Vulnerabilities

1 INTRODUCTION

C/C++ programs (e.g., web browsers, network servers) often require dynamically managed heap memory. However, it is very challenging to guarantee heap security. Over the past decades, a wide range of heap-related vulnerabilities – such as heap over-reads, heap overflows, use-after-frees, invalid-frees, and double-frees – have been discovered and exploited for attacks, including denial-of-service, information leakage, and control flow hijacking [36]. Currently, heap vulnerabilities continue to emerge. For instance, as shown in Table 1, a significant number were still observed within the past three months. It is very unlikely that heap vulnerabilities will disappear in the near future, without significant advancement in detection techniques. Thus, efficient and effective techniques are still required to defend against these vulnerabilities.

Vulnerabilities	Occurrences (#)
Heap Over-reads	54
Heap Overflows	66
Use-after-frees	5
Invalid-frees	2
Double-frees	2

Table 1: Number of heap vulnerabilities in the past three months (collected on 08/26/2017 from NVD [30]).

One method used to secure the program heap is to add defenses within the memory allocator [31], which can be combined with other security mechanisms, such as non-executable segments and address space layout randomization (ASLR). However, existing allocators are either insecure or inefficient. In particular, existing memory allocators can be classified into two types, based on their implementation mechanisms.

One type belongs to bump-pointer or sequential allocators, which sequentially allocate different sizes of objects in a continuous range [31]. They maintain freelists for different size classes to assist fast allocations, and are also called freelist-based allocators. Representatives of these allocators include both the Windows and Linux allocators, as well as Hoard [4], whose design features very limited security countermeasures. Even worse, some implementations may directly conflict with the goal of security. For instance, they place metadata immediately prior to each object, and reutilize the first words of a freed object to store pointers used by their freelists [31]. These designs will significantly increase the attack surface, since attackers can easily overwrite freelist pointers or other metadata to initiate attacks. Further details are presented in Section 2.2.

BIBOP-style ("Big Bag of Pages" [17]) allocators belong to the second type of allocators. They allocate several pages to serve as a "bag", where each bag will be used to hold heap objects of the same size. The metadata of heap objects, such as the size and availability information, is stored in a separate area. These allocators, such as `jemalloc` [12, 13], Vam [14], Cling [1], the OpenBSD allocator (which may be referred to simply as "OpenBSD" in the remainder of this paper) [29], TCMalloc [16], and DieHarder [31], avoid corruption of the metadata through isolation mechanisms. To the best of our knowledge, all existing secure allocators utilize the BIBOP-style.

Existing secure allocators, such as OpenBSD [29], Cling [1], and DieHarder [31], avoid the use of freelists for small objects. Instead, they maintain a bag-based bitmap to indicate the availability of all

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10...$15.00
https://doi.org/10.1145/3133956.3133957

objects within the bag. Although the bitmap mechanism reduces the memory consumption associated with tracking the status of heap objects, using only one bit for each object, it may impose significant performance overhead. If allocators utilize randomized allocation, this may impose an even larger overhead. For instance, the OpenBSD allocator randomly chooses one possible object inside a bag, upon every allocation. However, if this object is not available, it will linearly search for another available object inside the same bag (refer to lines 997 through 1014 of `malloc_bytes()` in omalloc.c for OpenBSD-6.0). In the worst case, it may search the entire bag to allocate one object, where the number of iterations can be proportional to the number of objects inside a bag. Furthermore, both OpenBSD and DieHarder may introduce false sharing problems [4], since multiple threads are sharing the same heap. For these reasons, secure allocators are typically much slower than performance-oriented allocators, although Cling is an exception that only focuses on use-after-free problems. Based on our evaluation of open-source secure allocators, OpenBSD imposes 22% performance overhead, and DieHarder results in over 36% slower runtime, on average.

This paper introduces FreeGuard, a secure BIBOP-style allocator that overcomes the performance issues of existing secure allocators. FreeGuard may not impose the same randomization as existing secure allocators, but runs at nearly the same speed as one representative performance-oriented allocator—the Linux allocator.

First, **FreeGuard designs a novel memory layout that combines the benefits of both BIBOP-style and sequential allocators**. FreeGuard takes the approach of BIBOP-style allocators: each bag, consisting of multiple pages, will hold objects with the same size class, while the object metadata is placed in an area separate from the actual heap. This design helps prevent attacks caused by corrupted metadata. At the same time, FreeGuard designs a "sequential bag placement" by employing the vast address space of 64-bit machines: FreeGuard maps a huge chunk of memory initially, then divides it into multiple heaps. Each heap will be further divided into multiple subheaps, proportional to the number of threads, and bags with increasing size classes will be placed sequentially, starting from the minimum size class to the maximum size class. This layout enables constant-time metadata lookup. If one bag inside the current heap is exhausted, FreeGuard simply services new requests from the equivalent bag in the next available heap. The detailed design is shown in Figure 3. For the purposes of security, FreeGuard also randomizes the following parameters: bag size, heap starting address, and metadata starting address, all of which increase the difficulty of attacks. Also, guard pages are randomly inserted throughout, in order to defend against buffer overflows and heap spraying.

Second, **FreeGuard adopts the freelist idea from performance-oriented allocators, and applies the shadow memory technique based on its novel layout**. FreeGuard discards the bitmap and hashmap designs of existing secure allocators, as they are not suitable for performance. As described above, bitmaps may incur significant performance overhead, which could be proportional to the size of the bitmap. Instead, using freelists can guarantee constant-time memory allocations and deallocations. FreeGuard further utilizes single-linked lists in order to prevent cycles within the list, which avoids the issue of double frees. It utilizes freelists to manage freed objects, but places the freelist pointers into segregated shadow memory, such that they cannot be easily corrupted.

Third, **FreeGuard greatly reduces the number of `mmap` calls required for allocating both the bags, and the metadata required for managing these chunks**. This design not only avoids the performance overhead caused by performing a large number of system calls, but also saves kernel resources in managing numerous small virtual memory regions. For the purposes of security, FreeGuard selectively places internal guard pages within each bag, based on a user-specified budget.

Additionally, FreeGuard also fixes several implementation weaknesses of existing secure allocators.

Contribution. In short, this paper makes the following contributions.

- **A Faster Secure Allocator**. We developed FreeGuard to be a faster secure memory allocator. FreeGuard was designed with a novel memory layout. In addition, FreeGuard also applies the freelist and shadow memory techniques, and reduces the number of unnecessary system calls, in order to improve performance. FreeGuard also fixes multiple issues associated with existing secure allocators, including possible false sharing problems, and provides better reporting of double and invalid frees.
- **Extensive Analysis of Secure Allocators**. We have provided an extensive analysis of the performance and security issues of existing secure allocators, such as OpenBSD, DieHarder, and Cling. Some understanding is obtained directly through examination of their source code.
- **Extensive Evaluation**. We have performed a large number of experiments to verify the performance, memory overhead, and effectiveness of FreeGuard. Experimental results show that FreeGuard imposes less than 2% overhead when compared to the Linux allocator, while providing significantly better security. Furthermore, FreeGuard considerably outperforms representative secure allocators, OpenBSD and DieHarder.

Outline. The remainder of this paper is organized as follows. Section 2 presents background on heap-related vulnerabilities. Section 2.2 further examines the advantages and disadvantages of several representative allocators, which motivate our work. Based on our detailed analysis, we discuss the key ideas of FreeGuard and its threat model in Section 3. Then, Section 4 provides the detailed implementation of FreeGuard, while Section 5 evaluates its performance, memory usage, and effectiveness. Next, Section 6 discusses the limitations of FreeGuard. Finally, Section 7 lists relevant related work, and Section 8 concludes.

2 BACKGROUND

This section provides a background of relevant memory vulnerabilities, as well as an extensive analysis of existing allocators. Familiarity with these memory vulnerabilities helps to understand how FreeGuard defeats them, while the analysis also helps to recognize the differences between FreeGuard and these existing allocators.

2.1 Heap-related Memory Vulnerabilities

2.1.1 Heap Over-reads. A heap over-read occurs when a program overruns the boundary of an object, possibly reading adjacent memory that was not intended to be accessible. It includes heap under-reads, where memory locations prior to the target buffer are referenced. Heap over-reads can occur due to a lack of built-in bounds-checking on memory accesses, particularly for C/C++ programs. They can cause erratic program behavior, including memory access errors, incorrect results, or a crash. They can also lead to security problems, including information leakage and denial-of-service attacks.

2.1.2 Heap Overflows. A heap overflow occurs when a program writes outside of the boundary of an allocated object. As with heap over-reads, throughout the remainder of this paper, heap overflows will also be used to refer to the related problem of corrupting memory immediately prior to the allocated object. Buffer overflows can cause security problems such as illegitimate privilege elevation, execution of arbitrary code, denial-of-service, and heap smashing.

2.1.3 Use-after-frees and Double-frees. Use-after-free occurs whenever an application accesses a previously deallocated object. A recent study shows that use-after-free errors are the most severe vulnerabilities of the Chromium browser, in terms of both the number of occurrences, and the severity of security impacts [23]. Double-frees are considered to be a special case of use-after-free, and occur when an object has been freed twice. Depending on the design of the specific allocator, use-after-free may cause execution of arbitrary code, loss of integrity, and denial-of-service attacks.

2.1.4 Invalid frees. For invalid frees, applications invoke `free()` on a pointer that was not acquired using heap allocation functions, such as `malloc()`, `calloc()`, or `realloc()`. Invalid frees can cause the execution of arbitrary code, intentional modification of data, and denial-of-service attacks.

2.1.5 Other Heap Errors. Other heap-related security vulnerabilities exist, including: initialization errors, failure of return values, improper use of allocation functions, mismatched memory management routines (e.g., `malloc`/`delete`), and uninitialized reads, all of which can lead to exploitable vulnerabilities. We must note that FreeGuard is not designed to handle these vulnerabilities.

2.2 Existing Secure Allocators

As described in Section 1, memory allocators can be classified into two major types: bump-pointer and BIBOP-style allocators.

Bump-pointer Allocators. Bump-pointer allocators, including the Windows and Linux allocators, typically employ freelists for the purpose of improved performance: they maintain freed objects in various freelists, organized by their size classes [31]. Figure 1 provides an overview of Linux's default memory allocator [22]. However, as they were not designed for security, they actually increase the attack surface for malicious users. Metadata, such as size and status information, are prepended to heap objects, such that overflows can easily destroy their contents. To save space, they also embed freelist pointers directly within freed objects, which can be altered easily by buffer overflows and use-after-frees. The only security feature supported by the Linux allocator is the ability to detect double and invalid frees. However, even this feature is only partially achieved, as it checks a single bit embedded into the size field to confirm the status of the object. Furthermore, `DLmalloc` has the following problems: (1) When the metadata is corrupted, due to buffer overflows or use-after-frees following consolidation, `DLmalloc` may either miss problems or generate incorrect alarms. For example, it may report a normal free as an "invalid free" problem. (2) It cannot report invalid frees if the pointer is outside the range of valid heap addresses. (3) It may incorrectly report an invalid free as a double free problem if the pointer refers to an unallocated area.

BIBOP-style Allocators. BIBOP-style allocators, such as PHKmalloc [19], dnmalloc [39], Vam [14], `jemalloc` [12, 13], OpenBSD [29], TCMalloc [16], Cling [1], and DieHarder [31], typically allocate one or more pages at a time (known as a "bag"), where each bag is used to hold heap objects of the same size. Among them, Cling, OpenBSD, and DieHarder are considered to be secure allocators, while others focus on performance only. We further discuss the design, advantages, and shortcomings of these three secure allocators.

2.2.1 OpenBSD Allocator. The OpenBSD allocator originates from PHKmalloc [19], but features substantial improvements on the security [29]. It avoids the use of freelists and inline metadata (headers). All descriptions here are based on the allocator of OpenBSD-6.0.

The OpenBSD allocator handles objects with small sizes differently than those with large sizes. Objects with sizes greater than 2 kilobytes will be considered as large objects.

Management of Small Objects. For small objects, the size of a bag is simply a page, which allows for multiple objects with the same size. For instance, for the 32-byte size class, 128 objects will fit inside one bag. OpenBSD allocates every bag by utilizing an `mmap` system call. It saves the information of chunks/objects in a separate area that is also obtained via the `mmap` system call. Typically, the information of multiple bags can share the same page, since the memory required to store the information for a single bag will be less than one page. As shown in Figure 2, OpenBSD utilizes a bit in the bitmap (shown as "bits" in the figure) to indicate the status of every object, where 1 indicates freed status, and 0 represents in-use. Other bag information, such as the size and number of available objects, is stored in the area before the bitmap. The OpenBSD allocator utilizes a hash table to track the relationship between bags and their metadata information. Basically, given an address, we could obtain the starting address of the page, then use it to search the hash table to find the chunk information for this bag. The hash table will grow automatically, in order to reduce potential hash conflicts.

The allocation and deallocation of small objects are further described as follows. (1) During allocation, the OpenBSD allocator first randomly selects one-out-of-four bag lists for the given size class. If there are no available objects in the first bag, it will invoke `mmap` to first allocate another bag, then thread this bag into the bag list with the proper size class. When objects are available in the first bag, the allocator will select one object randomly from the bag, as discussed in "Randomized Allocation" below. (2) During

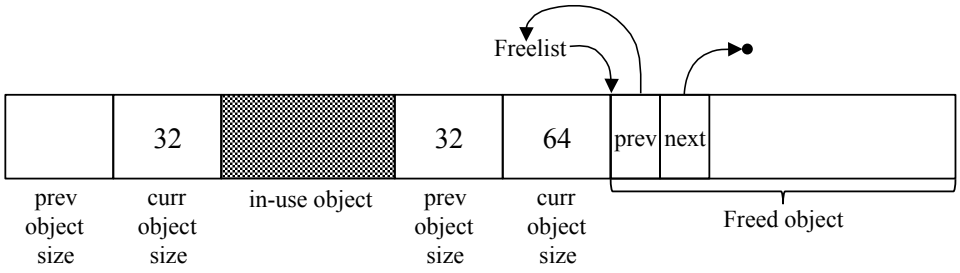

Figure 1: A fragment of the Linux allocator. Object headers are prepended to objects, which supports fast freeing and coalescing operations, but is vulnerable to overflows that can easily destroy the metadata.

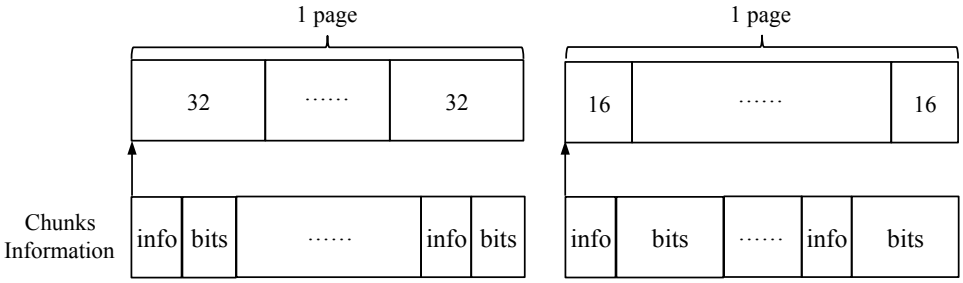

Figure 2: A fragment of the OpenBSD allocator. The mapping between bags and metadata (e.g. chunks information) is kept in a global hash table, and each bag has a bitmap to maintain the status of all objects inside. Metadata are typically stored in a separate location.

deallocation, a freed object will be randomly placed into a delayed array that can hold up to 16 freed objects. If a previously-freed object already exists in that slot, the previously-freed object will be actually freed, and the newly-freed object will take its place in the delayed array. If the previously-freed object is the first freed object of its bag – making the corresponding bag no longer full – the bag will be moved into the header of this bag list.

Management of Large Objects. For large objects (whose sizes are greater than 2 KB), OpenBSD applies a different policy. By default, OpenBSD keeps at most 64 pages in the `free_regions` cache in order to reduce the number of `mmap` system calls. Upon receiving an allocation request, OpenBSD will check whether it is possible to satisfy the request from the cached pages. If the requested size is less than the available pages in the cache, OpenBSD will check the entries in the cache, starting from a randomly-selected entry. If it can find one object whose size is equal to, or larger than, the requested size, OpenBSD allocates the object from the cache, and reinserts the remaining pages back into the cache. When there are no available objects capable of satisfying the current request from the cache, OpenBSD invokes the `mmap` system call directly.

For deallocation, OpenBSD first checks whether the size of the freed object is larger than the size of the preset cache (64 pages). If so, then this object will be deallocated directly by invoking the `munmap` system call. Otherwise, the current object is added to a random location in the cache. Note, that if the current freed object increases the total size of freed objects in the cache beyond 64 pages, then some existing cached objects will be unmapped in order to limit the total size of freed objects in the cache to no more than 64 pages.

Overall, the OpenBSD allocator implements the following approaches toward augmenting security, as shown in Table 2 as well.

No freelist, no object headers, BIBOP-style. These properties are inherited from the original design of PHKmalloc [19]. Since the OpenBSD allocator completely disposes with the use of freelists, it avoids possible corruptions to metadata (such as linked-list pointers) related to any freelists.

Fully-segregated metadata. Metadata information is maintained in an area separate from heap objects. This design is a departure from PHKmalloc, which stores metadata in the header of every chunk [19]. Obviously, fully-segregated metadata helps augment security.

Sparse page layout/Guard pages. Rather than using `sbrk`, such as PHKmalloc, the OpenBSD allocator employs the `mmap` system call to allocate a page from the underlying operating system each time it is required for small objects. In effect, this mechanism effectively places unmapped "guard pages" between regions, which limits the exploitability of both overflow attacks and heap spraying attacks.

Destroy-on-free. Destroy-on-free overwrites the contents of freed objects, filling them with random data. This policy is expected to locate some memory errors within applications. Currently, OpenBSD can also clean up an object prior to the memory being used, however,

this feature is disabled by default due to performance considerations.

Randomized allocation. OpenBSD employs two types of randomization during allocation. Firstly, it maintains four lists for each size class, and chooses one randomly from among them. Secondly, inside a bag, it will determine the index of an allocation randomly. If the object with that index is in-use, it will linearly search for the next available object, starting from the current position. The corresponding implementation is located between line 997 and line 1014 of `omalloc.c` of `OpenBSD-6.0`. Obviously, this second step may greatly compromise performance. In the worst case, when only a single free object exists in a bag, the number of searches is proportional to the total number of objects in the bag (e.g., a page).

Delayed memory reuse. During memory deallocation, a freed object is placed into a delay buffer that can hold up to 16 objects with the same size class. It computes an index into this array randomly. If a previously-freed object is occupying the corresponding entry of the delay buffer, that object will actually be freed, making room for the currently-freed object to be placed into the delay array.

Prevent invalid frees. The OpenBSD allocator could detect and prevent invalid frees with no false positives. Basically, it could identify the starting address and size of every object. If the corresponding object does not exist, or if the address is no longer a valid starting address, an invalid free is detected. Then, the allocator can stop execution for the purposes of attack prevention.

Prevent double frees. The OpenBSD allocator has a **very low** probability of detecting double frees, due to an implementation issue. Currently, it only checks for a double free problem whenever a freed object is placed into the delay buffer. Only when the object in the selected slot of the delay buffer shares the same address as the newly-freed object will a double free problem be detected. However, it can tolerate double frees, since one bit is used to record the status of an object. It will not cause the same linked-list problem that is inherent in freelist-based allocators.

2.2.2 DieHarder. DieHarder adapts many protections used by the OpenBSD allocator, but improves upon the randomized placement and randomized reuse by employing the randomization mechanism of DieHard [5]. DieHarder sparsely utilizes the pages in a continuous range of virtual address space, which is different from DieHard. However, DieHarder does not place guard pages inside a continuous region. A buffer overflow cannot be detected, even if it may be tolerated by its over-provisioning mechanism.

To further tolerate the vulnerabilities imposed by buffer overflows, DieHarder guarantees that the ratio of allocated objects, to the number of total objects, will never exceed $1/M$, where M represents the heap over-provisioning factor used to control this proportion. Thus, the entropy of choosing a random object is $O(\log N)$ (where N represents the number of allocated objects), which is much larger than that of the OpenBSD allocator. Similarly, DieHarder guarantees memory reuse to be less predictable. These two properties decrease the probability of overflow attacks. However, due to performance concerns, the default setting of M is less than 2 (actually 8/7), which indicates DieHarder will not waste half of the heap space to achieve better security.

DieHarder manages large objects differently from OpenBSD. Basically, it always allocates large objects using the `mmap` system call, then unmaps these objects by invoking the `munmap` system call. It does not utilize the cache mechanism of OpenBSD, thus helping defeat use-after-free problems. However, it may impose much larger performance overhead, caused by numerous system calls. An object with size larger than 64 KB will be treated as a large object by DieHarder. Our experiments also confirmed that the size of large objects will have a significant impact on the performance of applications.

However, based on our evaluation, DieHarder cannot detect or report double-frees and invalid-frees. However, it can tolerate these problems, which is the same as the OpenBSD allocator. DieHarder also has a scalability problem, as it uses a global lock to manage memory allocations/deallocations. This explains why the performance overhead reported here is higher than that of its original publication [31], since all evaluated applications of Section 5.1 are multithreaded ones, instead of single-threaded applications, as in their paper.

2.2.3 Cling. Cling is designed to defeat use-after-free problems, but does not protect against other types of vulnerabilities [1]. It also utilizes the BIBOP-style for managing small objects, and its bitmap is located outside of the actual heap for security reasons. It borrows the type-safety memory allocation idea from existing work [11], but without the use of compiler analysis. Basically, memory reuse is confined to only objects with the same type and same alignment (called "confining memory reuse"). It avoids the use of freelists to mitigate the problem of metadata corruption. In order to avoid the scanning of bitmaps, Cling borrows the idea of "reaps" [6], when multiple objects are allocated from the same allocation site. Basically, the allocations of objects inside the same bag will be in a sequential order. However, Cling does not introduce any randomization mechanism to increase the difficulty of other types of attacks. Also, the prevention of use-after-frees may fail if the object type is difficult to determine by allocation site [9].

3 OVERVIEW

3.1 Key Ideas

To the best of our understanding, the OpenBSD allocator and DieHarder have the following issues or limitations:

- **Inefficiency caused by the use of bitmaps:** The bitmap design clearly reduces memory consumption, but compromises efficiency. For instance, the worst case when searching the bitmap for a free object is proportional to the number of objects inside a bag, due to their randomized allocation policy.
- **Reduced randomization for larger size classes:** Depending on the size class, the effective level of randomization in OpenBSD may not be uniform. A smaller size class will provide better randomization, since an object will be chosen randomly from among the many objects within a page. However, when the size class is large, such as 2KB,

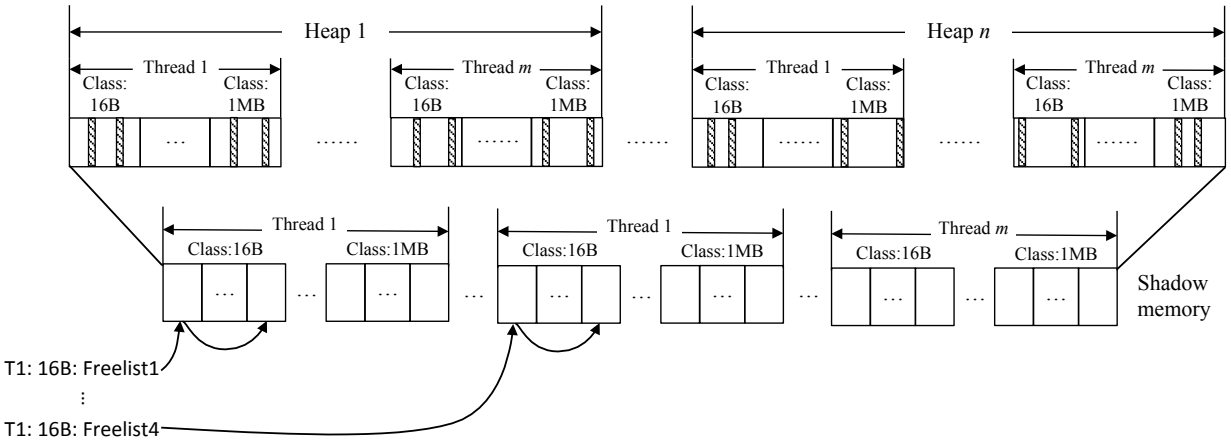

Figure 3: One example of FreeGuard's layout.

only two objects are present in each page, and one will be selected randomly from one-out-of-four bags.

- **Inefficiency and extra memory overhead caused by page-based `mmap`:** The OpenBSD allocator invokes an `mmap` system call to allocate a chunk, as well as storage for its `chunk_info`, every time. This method may place some guard pages between different chunks, due to the ASLR mechanism of the underlying operating system. However, a large number of system calls can significantly increase performance overhead. Also, the underlying OS may create a separate virtual memory region for each page, which consumes around 80 bytes of additional memory overhead in Linux. For small objects, DieHarder invokes `mmap` on the level of the miniheap, larger than one page, which reduces the number of system calls. However, it invokes `mmap` and `munmap` for large objects, whose size is larger than 64 kilobytes. In total, DieHarder also invokes a large number of `mmap` calls, as seen in Table 4.
- **False sharing performance problem:** In OpenBSD and DieHarder, multiple threads utilize the same heap simultaneously, which can cause false sharing problems that may substantially hurt the performance of applications [4]. False sharing is a usage pattern in which two or more threads simultaneously access different objects co-located within the same cache line. If one of these threads modifies the data, this will result in the entire cache line being invalidated for the other threads, despite the fact that they were not using the exact data being modified. This cache invalidation will result in a costly re-fetch of the required data, degrading application performance [26].
- **Other problems:** DieHarder cannot detect double and invalid frees, based on our evaluation. The OpenBSD allocator can only report a very small portion of double frees, since it reports double-frees only when the object being inserted into the delay buffer shares the same address as the prior object occupying the same slot in the buffer. When configured to utilize canaries, the OpenBSD allocator will only check for overflow of freed objects, which is insufficient to stop many possible buffer overflow attacks.

Our Design. Due to the above-mentioned problems, FreeGuard designs a novel allocator aiming to balance performance and effectiveness.

FreeGuard adopts almost all security features listed in Table 2, although with a lower entropy for randomization, as discussed in Section 6. The only feature not implemented by FreeGuard is DieHarder's over-provisioned allocation. Over-provisioned allocation is useful to increase randomization and reduce attacks caused by buffer overflows, since overflows may occur in unallocated free space. However, over-provisioned allocation may significantly increase memory consumption, and largely decrease performance due to lower cache and memory utilization, combined with higher TLB pressure. Instead, FreeGuard checks for the occurrence of overflow on neighboring objects at each deallocation, not just the item being freed, which is not supported by DieHarder. Then, if an overwrite is detected, FreeGuard can stop the program immediately. This method helps thwart attacks caused by overflows in a more timely manner.

For performance reasons, FreeGuard adapts the freelist mechanism that is widely utilized in performance-oriented allocators, such as the allocators of Linux and Windows systems. Freelists excel at performance, since each allocation and deallocation can be completed in constant time. Also, the freelist maintains the order of deallocations, which helps reduce attacks caused by use-after-frees, the most serious type of security attacks in Microsoft products recently [23]. Different from existing freelist allocators (see Section 2.2), FreeGuard allocates these freelist pointers in a separate space, and uses only a single-linked list, to reduce memory consumption, shown as the shadow memory in Figure 3. To save space, object status information is stored within the same word: if the object is available, then its lowest-order bit will be 0 (this will hold true whether the location contains a pointer to the next available object, or whether it is null, indicating no next-available object exists). Conversely, if the object is in-use, its status will exactly equal 1.

The **second design element** is to reduce performance overhead and memory consumption caused by page-based `mmap` operations. In order to reduce calls to `mmap`, FreeGuard allocates a huge block initially, and places guard pages randomly inside each bag (shown

as boxes with diagonal lines in Figure 3). Currently, guard pages will be placed randomly to occupy 10% of each bag. This method reduces the number of mmap calls to less than 10%, since OpenBSD invokes additional mmap system calls to allocate storage for chunk_info structures, as well.

The **third design element** is to improve the performance of fetching corresponding metadata. Currently, OpenBSD and DieHarder create a hash table in which to map the page address of heap objects to a specific index, and grows the total size of this hash table whenever necessary. However, this still imposes significant performance overhead, especially when multiple pages are mapped to the same bucket. Instead, FreeGuard relies on the fact that 64-bit machines have a vast address space, and utilizes the shadow memory technique to save metadata [41]. For any given heap address, FreeGuard can quickly compute the location of its metadata, and vise versa. The layout of the allocator is shown as Figure 3, and further described in Section 4.

3.2 Scope, Assumptions, and Threat Model

The security properties supported by FreeGuard are listed in Table 2, as well as those of other allocators. Overall, FreeGuard has the same performance overhead as the glibc allocator, but provides a better security guarantee than all existing allocators. Next, we discuss the attacks that can and cannot be stopped by FreeGuard, and explain the fundamental reasoning.

Scope. For attacks based on invalid and double frees, FreeGuard can prevent all such attacks, as long as the status of an object is never corrupted. Because the status information is kept in a separate location, this will greatly reduce the possibility of success for these attacks. Even if the status were to be modified by the attacker, some invalid frees caused by an invalid address can be prevented due to FreeGuard's special allocator design.

Buffer overflow/over-read attacks will fail if the access touches one of the guard pages inserted randomly by FreeGuard. Additionally, buffer overflows can be detected if one of the implanted canaries is found to have been corrupted. Implanting canaries will result in additional verification steps at the time the object (or one of its adjacent neighbors) is freed. At the same time, the difficulty of issuing these two types of attacks is increased due to randomized allocations, since the address of a target object is much harder to guess.

Attacks based on use-after-frees are reduced by utilizing delayed memory reuses. If an object is not re-utilized, the attacker may fail to exploit use-after-frees, since it will not cause any ill effect. Also, memory reuses are randomized to increase the difficulty of successful attacks.

Assumptions. FreeGuard assumes that the starting addresses of both the heap and the shadow memory are kept hidden from the attacker. If an attacker has knowledge of these addresses, he can possibly change the status of an object, and force the allocator to make an incorrect decision. To avoid the predictability of these addresses, FreeGuard allocates this memory using the mmap system call, which is guaranteed to return a random address if ASLR is enabled on the underlying OS. However, if the attacker has permission to run a program on the machine, he may be able to guess the location of the metadata, then take control of memory allocation. More discussion is provided in Section 6.

4 IMPLEMENTATION

This section explains the detailed implementation of FreeGuard. Basically, FreeGuard focuses on the management of small objects, and adopts the same mechanism of DieHarder for managing larger objects. However, FreeGuard defines large objects differently, such that only those objects with sizes larger than 1MB will be treated as "large objects".

4.1 Managing Small Objects

Section 3.1 describes the basic idea of managing small objects. First, FreeGuard utilizes the BIBOP-style in order to place the metadata in another location, avoiding possible metadata-based attacks. This achieves the "fully-segregated metadata" target shown in Table 2. Second, FreeGuard utilizes freelists for better performance, rather than using a bitmap. Third, FreeGuard supports the fast fetching of metadata (such as freelist pointers) using a novel heap layout, shown as Figure 3.

FreeGuard initially maps a huge block of memory, and divides this block into multiple heaps in the beginning. Inside each heap, FreeGuard employs a per-thread subheap design so that memory allocations from different threads will be satisfied from different subheaps, in order to avoid possible false sharing problems [4]. All bags belonging to a thread, which hold objects with different size classes, are located together. The bag size, starting address of the heap, and the starting address of the shadow memory that keeps the metadata of heap objects, are randomly chosen for each execution for the purpose of increased security.

The rest of this section focuses on the implementation of other security features, as listed in Table 2.

4.1.1 Randomized Guard Pages. FreeGuard initially utilizes the mmap system call to allocate a large chunk of memory, where the starting address of the heap is randomized between executions, a feature enabled by the ASLR mechanism of the underlying OS. The bag size utilized throughout each execution, which remains the same across the different size classes, is randomized with every execution, and ranges ranges between 4MB and 32MB. These mechanisms guarantee that the starting address of each bag is random across multiple executions.

FreeGuard inserts guard pages randomly within each bag. Prior to allocating objects from a new page, FreeGuard determines whether this page should be utilized as a guard page. This decision is based on a predetermined user budget, such as 10%. Thus, 10% of pages inside each bag will be chosen as guard pages. When a page is randomly selected to be a guard page, FreeGuard invokes the mprotect system call to make this page inaccessible, such that all memory accesses on this page will be treated as invalid, and trigger segmentation faults. For a bag with a size class larger than one page (4KB), the size of its guard pages will be the same as the size class. That is, multiple pages will be utilized as guard pages in order to avoid misalignment of the metadata. Guard pages are useful for stopping buffer overflows, buffer over-reads, and heap spraying, as access on guard pages will immediately stop execution.

Security Features	Security Properties	glibc	Cling	DieHarder	OpenBSD	FreeGuard
No/segregated freelist	Prevent attacks on freelist related pointers		✓	✓	✓	✓
No object headers	Prevent metadata related attacks		✓	✓	✓	✓
BIBOP style	Prevent metadata related attacks		✓	✓	✓	✓
Fully-segregated metadata	Prevent metadata related attacks		✓	✓	✓	✓
Confining memory reuse	Prevent use-after-free attacks		✓			
Destroy-on-free	Help finding some memory errors			✓	◇	◇
Guard pages	Reduce attacks of buffer overflows and over-reads Reduce heap spraying attacks			✓	✓	✓
Randomized allocation	Increase difficulty of attacks caused by use-after-frees			✓	✓	✓
Over-provisioned allocation	Reduce possible attacks caused by overflows			✓		
Delayed/randomized reuse	Reduce possible attacks caused by use-after-frees			✓	⊖	✓
Detect invalid frees	Prevent attacks caused by invalid frees	⊖			✓	✓
Detect double frees	Prevent attacks caused by double frees	⊖			⊖	✓
Check overflows on frees	Stop attacks caused by overflows timely				⊖	✓

Table 2: Security features of existing secure allocators, while adding glibc for the comparison. "✓" indicates that the allocator has this feature. "◇" indicates that this is an option, but is disabled by default. "⊖" indicates that the implementation has some weakness.

4.1.2 Randomized Allocation and Delayed Reuse. FreeGuard takes a different approach from all existing allocators, by balancing randomization and performance.

FreeGuard maintains four bump pointers for each size class of each per-thread heap, which always point to the first never-allocated object [14, 21]. Objects will be allocated in a sequential order. After an object is allocated, the corresponding pointer will be bumped up to the next one. Whenever a bump pointer refers to the start of a new page, FreeGuard determines whether this new page should be utilized as a guard page, as discussed above. FreeGuard uses this sequential order for the purposes of performance, though it may compromise security. More discussion can be seen in Section 6.

FreeGuard also maintains four freelists to manage freed objects for each size class of each per-thread heap. A freed object will be added into one-out-of-four freelists randomly. Objects in a freelist will be reused in a first-in/first-out (FIFO) order. In this way, some use-after-free problems can be prevented automatically, since a freed object may be reallocated only after a long period, in which any use-after-free problems appearing in this period can be tolerated automatically. However, this method may slightly reduce performance compared with allocators using the last-in/first-out (LIFO) order. For the LIFO order, there is a significant chance that a newly allocated object is still inside the cache, which can avoid fetching from memory. However, our method will be superior to LIFO implementations in terms of security. It will significantly increase the difficulty of guessing the address of an allocation, due to the combination of FIFO and randomization, as discussed below. Overall, the FIFO mechanism increases both reliability and security. This mechanism cannot easily be supported when using bitmaps, such as the OpenBSD allocator or DieHarder. Bitmap-based allocators only use one bit to indicate the state of an object, either in-use or free. After a freed object is returned to the bitmap, there is no way to maintain the temporal information. Due to the use of FIFO, there is no need to utilize a delay buffer, which is different from OpenBSD.

FreeGuard introduces randomization into its memory allocations. An allocation request could be satisfied either from one-of-four bump pointers, or one-of-four freelists, based on the value of a random number. This randomization is achieved through the following steps. First, we generate a random number R using the Intel SSE2 number generator, as discussed below. We then take the modulus value N by calculating $R\%4$. N will decide which freelist or bump pointer will be utilized. We will always check the N^{th} freelist first, and if freed objects are available, it will reuse them to satisfy the request. However, if there are no free objects in this freelist, the allocation will fall back to the N^{th} bump pointer. Furthermore, we will always check if the expression $R\%W$ is equal to zero, where W represents a weighting factor. If so, FreeGuard will strictly utilize the N^{th} bump pointer, regardless of whether the N^{th} freelist contains any objects available for reuse. Therefore, in terms of W, we will have a 1-in-W chance of overriding the freelist and using the bump pointer instead. This method may slightly increase memory consumption and cause some slowdown, due to the increased memory footprint. However, it actually increases randomization, which is different from OpenBSD. OpenBSD will never allocate from a new bag, when there are freed objects it can reuse in the chosen bag.

Incorporation of Fast Random Number Generator. In our initial design, we utilized the `glibc rand` function to generate a random number. However, this method is found to be very slow due to lock conflicts. The invocation of `rand` will acquire a global lock, which may prevent another thread from simultaneously obtaining a random number. To improve performance, FreeGuard utilizes a **fast** pseudo-random number generator (RNG)[32]. This faster RNG was optimized using Intel's SSE2 extensions, and further, does not require the use of synchronization primitives internally. Adopting this fast RNG reduced the performance overhead of `swaptions` by up to 65%.

4.1.3 Checking Overflows at Deallocation. FreeGuard borrows another mechanism of OpenBSD to thwart possible attacks caused by buffer overflows. However, the OpenBSD allocator disables this mechanism, by default. In fact, based on our evaluation, this mechanism is very lightweight and helpful toward stopping attacks in a timely manner.

FreeGuard also increases the number of checks upon every deallocation. Currently, it will check the neighboring four objects as well, two before the current object and two after, instead of just one object. To support this, every allocation request will add one

additional byte, at the end of the object, in which to hold a canary. Upon deallocation, if one of these five canaries has been changed to other values, FreeGuard can halt execution of the current program. Note, that adding one byte to the end of an object may significantly increase memory consumption, since FreeGuard always manages objects within size classes featuring powers of two. Thus, one additional byte may double the size of the memory consumption in the worst case.

4.1.4 Preventing Double and Invalid Frees. For both of these problems, FreeGuard will halt the execution immediately, and report the problem precisely, with 100% guarantee.

FreeGuard prevents the following invalid frees: (1) If a free pointer lies outside the address range of the heap, a case which is easy to detect, and that most allocators can possibly detect. (2) If a free pointer falls within the range of the heap, but was never allocated. This could be discovered easily by checking its corresponding status. However, the Linux allocator may wrongly consider this problem to be a double-free error. FreeGuard avoids this issue and reports it correctly. (3) If a free pointer is not aligned to the object's specific size class. FreeGuard detects this problem easily based on its "information computable" design. FreeGuard avoids false alarms and false negatives present in the Linux allocator, and caused by corruption of metadata, since FreeGuard maintains the status of each object in shadow memory that is segregated from the actual heap.

FreeGuard also relies on the status information to detect possible double-frees upon deallocations. FreeGuard always reports possible double frees, avoiding the implementation faults of the OpenBSD allocator. The segregation of metadata ensures that FreeGuard can always detect double frees, unlike the Linux allocator.

4.2 Managing Large Objects

FreeGuard borrows the same mechanism as DieHarder to handle large objects, which is discussed in Section 2.2.2. Both provide better protection on "large" objects than OpenBSD. They can significantly reduce possible use-after-free attacks, since any access occurring after the `munmap` operation may actually cause the program to crash. They could defeat most buffer over-writes and over-reads, since the ASLR mechanism will effectively place guard pages before and after a mapped area, in most situations. Instead, OpenBSD maintains a cached list to track freed objects, which makes it fail to defeat use-after-frees. It has an option to protect the area of freed objects, but is disable by default due to performance reasons. We enabled this option and found that it may significantly affect performance, due to the increased number of system calls. OpenBSD treats objects with sizes larger than 2,048 bytes as large objects, resulting in many of its objects being treated as large objects.

FreeGuard defines "large" objects differently than DieHarder, which treats objects with sizes exceeding 64 kilobytes as large. This provides better protection than FreeGuard, but with increased overhead due to the increased number of system calls.

5 EXPERIMENTAL EVALUATION

This section focuses on the following research questions.

- What is the performance overhead of FreeGuard, in comparison to the representative general-purpose allocator (the Linux allocator), and other secure allocators, such as the OpenBSD allocator and DieHarder?
- What is the memory overhead of FreeGuard? Also, we compare it against the allocators mentioned above.
- How effectively can FreeGuard reduce or prevent real attacks?

We performed all experiments on a 16-core quiescent machine, with two sockets installed with Intel(R) Xeon(R) CPU E5-2640 processors. This machine has 256GB of main memory, with 256KB L1, 2MB L2, and 20MB L3 cache. The experiments were performed using the unchanged Ubuntu 16.04, installed with Linux-4.4.25 kernel. We used GCC-4.9.1 with `-O2`, `-g` flags to compile all applications and all evaluated allocators appearing in this paper.

We utilized the default settings for both the Linux allocator and DieHarder. For the OpenBSD allocator, we utilized a junking level of 0, in order to provide a fair comparison with FreeGuard. For DieHarder, we utilized the version of 08/05/2017, where an object with size larger than 65,536 bytes (64 kilobytes) will be considered a large object, and with the heap multiplier M set to 8/7. We experienced much higher performance overhead when M is set to 2, or higher. Both FreeGuard and OpenBSD do not enable destroy-on-free, which is enabled by DieHarder.

5.1 Performance Overhead

We evaluated 19 applications, and show the average results of ten executions in Figure 4, where all values are normalized to the `glibc` library. A taller bar indicates a larger overhead. Among them, eleven are from the PARSEC suite of applications, while others are real applications, including `Apache httpd-2.4.25`, `Firefox-52.0`, `MySQL-5.6.10`, `Memcached-1.4.25`, `SQLite-3.12.0`, `Aget`, `Pfscan`, and `Pbzip2`. All evaluated applications are multithreaded applications, making them more relevant toward gauging performance on modern multicore machines than single-threaded benchmark suites, such as SPEC. Both DieHarder and OpenBSD utilize a single heap to satisfy requests, instead of per-thread heaps, with a scalability issue.

PARSEC applications were exercised using native inputs [8]. `MySQL` was tested using the `sysbench` application, with 16 threads and 100,000 max requests, the throughput of which is shown. `Memcached` was tested using the `python-memcached` script [34], but changed to loop 20 times in order to obtain sufficient runtime. `SQLite` was tested using a program called "threadtest3.c" [10]. `Apache` was tested by sending 10,000 requests via `ab` [15]. For `Aget`, we collected the execution time of downloading 600MB of data from another quiescent server located on the local network. For `Pfscan`, we performed a keyword search within 800MB of data. For `Pbzip2`, we performed compression on a file containing 150MB of data. Finally, `Firefox-52.0` was evaluated in a headless configuration using a Python script to instruct the browser, via geckodriver, to fetch a fixed set of 71 web pages cached on a proxy server located on the local network. The `time` utility was used to measure the runtime required to perform these operations, as well as the maximum resident set size.

Figure 4 shows the normalized runtime of different allocators. Compared to the Linux allocator, FreeGuard's performance overhead is only 1.8% using the arithmetic average, and 1.4% using

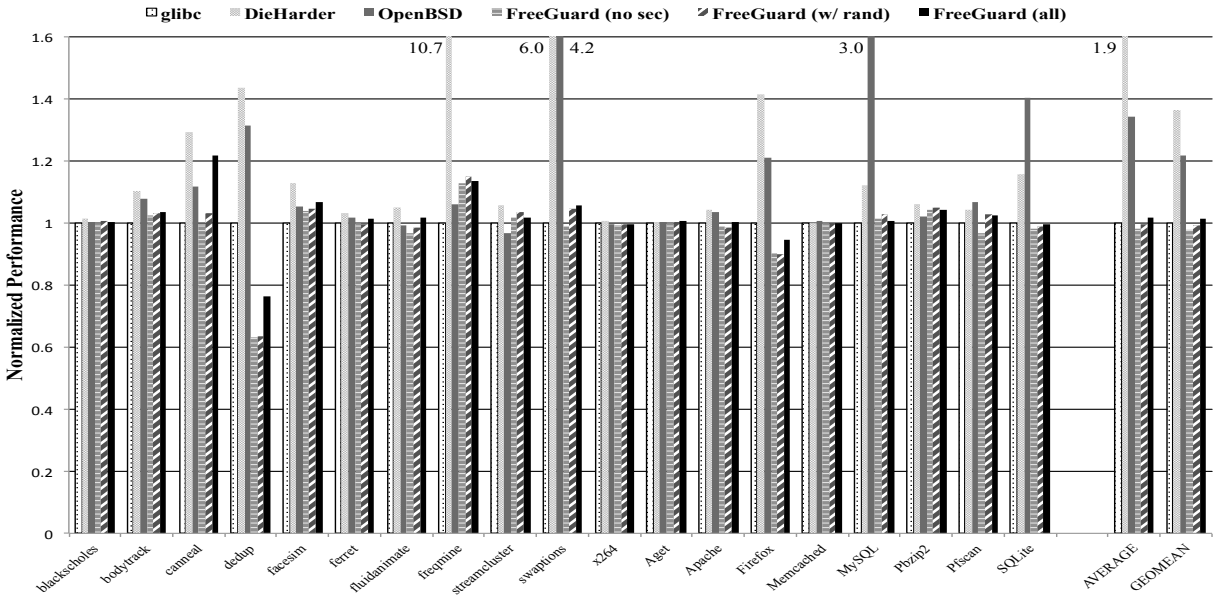

Figure 4: Normalized runtime with different allocators, where a higher bar indicates a higher overhead.

the geometric mean, with a number of security features enabled (Table 2). In comparison, the OpenBSD allocator has an overhead of around 34% (arithmetic mean) or 22% (geometric mean), while DieHarder runs around 88% (arithmetic mean) or 36% (geometric mean) slower than the default Linux allocator. This indicates that FreeGuard significantly outperforms the existing allocators.

With the exception of `canneal` and `freqmine`, FreeGuard imposes less than 10% performance overhead. FreeGuard has additionally enabled the delayed memory reuse feature by default, which adds around 2% performance overhead.

As shown in Table 3, `canneal` involves a large number of memory allocations and deallocations, around 30 million. Thus, the performance slowdown is caused by the additional overhead of these allocations and deallocations. Since the glibc allocator prepends metadata before the actual objects, it takes virtually no time to fetch the metadata when there are no errors. Although FreeGuard can compute the object's metadata location easily, it still imposes a larger overhead than the glibc allocator. For each deallocation, FreeGuard must identify the placement of the metadata in order to add the entry into the freelist. When an object is allocated from a freelist, it must compute the corresponding heap address for the metadata. Currently, the freelist only contains the metadata address of the free objects. Afterwards, the metadata should be changed to reflect the object's updated status. The allocation of an object will actually involve two cache lines, instead of only one, which also adds some overhead.

For `freqmine`, as seen in Table 3, a considerable proportion of these allocations were large objects, which FreeGuard handles by invoking the `mmap` system call in response to each such request. Therefore, a clear performance penalty will be associated with this method of handling large objects, and explains the degraded performance of FreeGuard for this application.

Figure 4 also shows that FreeGuard considerably outperforms the Linux allocator for the `dedup` application. Based on our analysis, the Linux allocator's default configuration invokes a large number of `madvise` calls (over 500,000), in order to return memory back to the OS. However, FreeGuard does not invoke such `madvise` system calls, which explains why FreeGuard significantly outperforms the Linux allocator in this case. Consequently, FreeGuard shows larger memory consumption on this application, with around 64% more memory used.

5.2 Memory Overhead

We have evaluated memory overhead of different allocators on the same applications. For server applications like `MySQL` and `Memcached`, we executed a script to periodically collect the `/proc/PID/status` file, and utilize the maximum value of the VmHWM field to represent its maximum memory consumption. Memory consumption of other applications was collected using the `maxresident` output of the `time` utility, which reports the maximum amount of physical memory consumed by an application [25].

The physical memory overhead of running Linux, OpenBSD, and FreeGuard, is shown in Table 3. Overall, OpenBSD has almost the same memory overhead as Linux, since it always prefers freed objects. Comparing to the Linux allocator, the total memory overhead of FreeGuard is around 20%, while OpenBSD actually uses 6% less memory, and DieHarder's memory overhead is about 11%.

We have investigated to determine the cause of this. FreeGuard utilizes four bump pointers and four freelists. Memory reuse randomization may significantly reduce the re-utilization of a particular object. For instance, consider an application performing eight allocations, each of size 1MB, where one allocation follows after another deallocation. The Linux allocator will immediately re-utilize the freed object, which only increases the memory footprint by 1MB, in total. However, FreeGuard may utilize up to 8MB

Programs	Runtime (s)	Total Allocs (#)	Large Allocations (#)			Memory Usage (MB)			
			DieHarder	OpenBSD	FreeGuard	glibc	DieHarder	OpenBSD	FreeGuard
blackscholes	37.26	22	4	4	4	613	619	613	615
bodytrack	26.6	437572	13053	15417	0	33	42	31	62
canneal	55.39	30728188	1	1720	1	943	1131	808	1281
dedup	16.23	4073908	359	1152213	7	1639	2076	1007	2830
facesim	70.41	4746623	16970	33393	26	323	393	341	376
ferret	4.58	139013	1557	7374	1	66	90	67	101
fluidanimate	29.67	229928	6	8	2	408	464	429	433
freqmine	44.31	6638	6103	6227	3068	1859	1785	1821	1996
streamcluster	62.08	113271	30	1758	3	111	115	111	115
swaptions	19.98	48001811	0	16000129	0	9	12	7	12
x264	53.23	35771	241	35483	35	485	516	502	483
Aget	5.5	50	0	18	0	6	6	3	5
Apache	–	495	0	4	0	5	5	5	6
Firefox	78.01	21126988	5393	413270	264	158	158	161	160
Memcached	4.62	118	5	17	0	6	9	6	9
MySQL	–	1898867	51697	391172	5	123	132	271	154
Pbzip2	1.46	1229	851	1022	0	94	100	95	255
Pfscan	1.46	43	0	2	0	735	782	784	821
SQLite	20.62	1345226	9635	1075648	0	40	62	34	122

Table 3: Program characteristics related to different memory allocators. (Apache and MySQL were measured by transactions per second rather than runtime.)

of memory, since the randomization may choose to allocate only from bump pointers, or a chosen freelist may not have any available objects, causing it to fall back to the bump pointer again. Although, when there are already multiple objects available in the freelists, memory overhead will be not significantly increased. FreeGuard compromises memory overhead in order to achieve better randomization.

Currently, FreeGuard imposes more memory overhead than the OpenBSD allocator. There is a balance between memory overhead and performance overhead: (1) The OpenBSD allocator treats memory allocations larger than 2KB as large objects, which will be allocated utilizing `mmap` every time (see Table 4). For each allocation larger than 2KB, but less than 1MB, the OpenBSD allocator will waste, at most, one page. Instead, FreeGuard utilizes power-of-two size classes to manage objects less than 1MB. Thus, it is possible to have larger internal fragmentation, with an upper bound of 50%. In its default setting, FreeGuard adds one byte for the canary, which helps find possible overflows in a timely manner, and stop the program accordingly. This additional byte may waste almost 50% of the allocated space if the original size was already aligned to a power of two. (2) The OpenBSD allocator utilizes one bit to indicate the status of an object, which also minimizes memory consumption, but with higher performance overhead. FreeGuard will use one word for each object in order to thread an object into the freelist. (3) FreeGuard utilizes four freelists and four bump pointers, and may randomly choose to allocate an object from bump pointers despite free object availability. This also adds some memory overhead, but provides better protection due to increased randomization.

Application	Vulnerability	Reference	Original	FreeGuard
bc-1.06	Buffer Overflow	Bugbench [27]	Crash	Mitigation
ed-1.14.1	Invalid Free	CVE-2017-5357	Crash	Prevention
gzip-1.2.4	Buffer Overflow	Bugbench [27]	Crash	Mitigation
Heartbleed	Buffer Over-Read	CVE-2014-0160	Data Leak	Mitigation
Libtiff-4.0.1	Buffer Overflow	CVE-2013-4243	Crash	Mitigation
PHP-5.3.6	Use-After-Free	CVE-2016-6290	Crash	Mitigation
	Use-After-Free	CVE-2016-3141	Crash	Mitigation
	Double Free	CVE-2016-5772	Crash	Prevention
polymorph-0.4.0	Buffer Overflow	Bugbench [27]	Crash	Mitigation
Squid-2.3	Buffer Overflow	CVE-2002-0068	Crash	Prevention

Table 5: Verifying FreeGuard on several vulnerabilities.

5.3 Effectiveness

To verify the effectiveness of FreeGuard, we have tested it on several different real-world vulnerabilities, as shown in Table 5. Note that these vulnerabilities have also been evaluated in other works, such as FreeSentry [38].

We confirmed whether FreeGuard can prevent or mitigate latent problems in these applications. "Prevention" indicates that FreeGuard completely avoids the problem, such as with double or invalid frees. "Mitigation" indicates that the possibility of successful attack is reduced, although there is no full guarantee that such a problem will always be avoided.

All vulnerabilities were confirmed in the original applications prior to linking with the FreeGuard library. All applications, except `OpenSSH`, resulted in program crash. These problems include use-after-free, double-free, and buffer overflow problems; OpenSSH experiences an information leak.

bc-1.06. bc, an arbitrary precision numeric processing language, contains a heap buffer overflow. We obtained a buggy version of this program from BugBench, a C/C++ bug benchmark suite [27]. Bad input can trigger the buffer overflow, and will normally result

	glibc			DieHarder			OpenBSD			FreeGuard		
	mmap	munmap	mprotect	mmap	munmap	mprotect	mmap	munmap	mprotect	mmap	munmap	mprotect
blackscholes	36	17	24	126	17	32	59	17	35	38	5	11023
bodytrack	6585	6555	125	20469	19600	33	19661	20466	36	6557	6526	11177
canneal	52	24	42	215307	27	31	154856	152849	34	33	2	33763
dedup	650	862	273989	258295	363	28	434648	499668	31	35	2	42631
facesim	105	39	42	33106	16940	31	14062	13705	584	38	6	12249
ferret	319	297	183	8119	1839	34	7037	8532	549	294	263	11483
fluidanimate	43	19	28	20663	30	32	14929	14725	35	34	3	12532
freqmine	212	164	167	6848	6322	49	6187	6035	52	3165	3124	11460
streamcluster	107	80	90	257	102	92	201	136	95	33	2	11042
swaptions	53	25	220	992	20	32	365	14	35	32	1	87760
x264	286	269	39	1424	503	32	1016	943	35	34	3	11319
Aget	54	27	44	99	14	32	87	14	35	33	2	11023
Apache	239	32	141	417	34	140	295	30	143	225	32	10125
Firefox	12248	8916	207834	70947	14545	209854	93006	143198	208699	11845	8685	202429
Memcached	39	8	25	214	5	23	97	1	24	34	1	11030
MySQL	154	33	326	17239	14449	62	22248	49876	65	67	17	12079
Pbzip2	120	92	143	1114	1037	37	939	880	40	33	1	11088
Pfscan	41	2	30	83	2	34	76	2	37	36	2	11025
SQLite	65	33	4160	14152	9665	33	239746	254387	36	38	7	14994

Table 4: System call counts, including both the application and the allocator.

in a program crash. An array requiring 512 bytes is requested, and an object of size 1,024 bytes is returned by FreeGuard. The bug causes bc to access one element beyond the boundary of the array. FreeGuard prevents a crash from occurring, as the overflow occurs within the slack/unused portion of the object, but does not reach either the canary or guard page located at the end of the allocated space.

ed-1.14.1. ed has an invalid free problem that can cause the program to crash, since the developers changed a `malloc`'d buffer for a static one, but forgot to remove the corresponding free operation. FreeGuard can always report and prevent this problem, and print the call stack of the invalid free inside.

gzip-1.2.4. gzip, the GNU compression and decompression program, contains a stack overflow problem. We converted this problem into a heap overflow. When it occurs, it causes the program to crash, since adjacent metadata will be corrupted. FreeGuard, however, avoids the crash, due to its segregated metadata.

Heartbleed. FreeGuard's protection against buffer over-read was validated against the Heartbleed bug, which results in the leakage of up to 64KB of data occurring from the heap. During our evaluation, we observed that the OpenSSL library will request approximately 33KB to use for receiving the client heartbeat request. Due to the nature of a BIBOP-heap, this results in FreeGuard fulfilling the request with a 64KB object, as this is the smallest available bag (whose sizes follow powers-of-two) capable of satisfying the request. The malicious heartbeat request contains a falsified payload length value, indicating the payload is 64KB long, when in fact, it is empty. The server then allocates a new buffer in which to construct the heartbeat reply, and proceeds to copy 64KB from the start of the payload region within the request buffer, the amount indicated by its falsified header value. However, because the request data is stored in a buffer of size 64KB, and the payload section is not located at the beginning of the object, this results in a buffer over-read occurring. Normally, this would result in the leakage of uninitialized heap data. But, with FreeGuard's random guard pages enabled, the buffer over-read can result in a program crash immediately upon accessing the random guard page (if present) placed at the end of the object. The proportion of random guard pages inserted onto the heap is configurable, and was set to 10% for our evaluation. As such, the Heartbleed attack was prevented 1-in-10 times, resulting in a program crash.

Libtiff-4.0.1. To validate FreeGuard's protection against buffer overflow vulnerabilities, a heap-based buffer overflow found in gif2tiff, a GIF-to-TIFF image conversion tool found in the libtiff library, was tested. When supplied with a specially-crafted image, gif2tiff will attempt to process the file, resulting in a crash. An attacker might exploit this vulnerability, and could potentially execute arbitrary code under the privilege level of the account used to run the process. We reproduce the exploit by supplying a crafted GIF image as input. After linking to FreeGuard, the program avoids the crash, instead reporting, "illegal GIF block type".

PHP-5.3.6. For PHP, two use-after-free vulnerabilities are triggered by dedicated XML data. They would allow attackers to cause a denial-of-service attack by crashing the program. We apply FreeGuard to PHP and rerun the vulnerable code. One program prints the correct data, while the other prints a null value. Although the output is not correct in these cases, FreeGuard prevents the program crash and, therefore, successfully prevents the denial-of-service attack. FreeGuard provides protection due to its delayed and randomized reuse mechanisms; by reutilizing freed objects in

FIFO order, as well as randomly choosing an object to return, certain use-after-free errors will not result in the corruption of in-use object data.

We additionally tested FreeGuard with PHP when experiencing a vulnerability caused by unserializing malicious XML data, an issue that results in program crash due to a double-free error. We use a malicious PHP script to reproduce this vulnerability. With FreeGuard, the program reports the complete call stack upon the second free.

polymorph-0.4.0. Polymorph, a filename converter, has a stack buffer overflow problem that was modified to use the heap for the purposes of our evaluation. This overflow is actually very similar to that of `gzip`, which will change the metadata and cause the program to crash when using the `glibc` library. As with `gzip`, FreeGuard avoids the crash due to its segregated metadata.

Squid-2.3. The affected version of Squid – a caching Internet proxy – contains a heap buffer overflow. Squid allocates memory from which to build a title URL string, however, it fails to account for the extra space needed to URL-escape the string. Thus, a buffer overflow occurs when the program attempts to escape the string, and writes the result to an inadequately-sized heap buffer. For this case, FreeGuard will always detect the overflow before the program crashes, since FreeGuard can always find out the corrupted canaries in single-threaded programs, although conceptually, FreeGuard can probabilistically prevent buffer overflows.

Conclusion: In each of these instances, FreeGuard allows the programs to either run normally with no ill effects, or immediately halts execution and reports the problem to the user. As described above, FreeGuard can always detect double and invalid frees. FreeGuard prevents or mitigates buffer overflows due to the following mechanisms: first, FreeGuard's metadata segregation prevents the corruption of metadata; second, due to FreeGuard's power-of-two object class sizes, it tolerates a certain level of corruption, and; third, when the canary within an adjacent object has been detected to be corrupt, FreeGuard immediately produces a warning and calls `abort`. FreeGuard also mitigates use-after-free problems due to its reuse of freed objects in FIFO order.

6 LIMITATIONS

Both FreeGuard and DieHarder utilize the same mechanism for the management of large objects, which is safer than that of OpenBSD. Currently, OpenBSD cannot effectively defend against use-after-free vulnerabilities due to its cache mechanism, since freed objects are not protected after their deallocations. Instead, accessing a freed object in FreeGuard and DieHarder will cause an access violation, since a freed object will be unmapped directly.

For small objects, FreeGuard has some limitations in randomized allocation, randomized memory reuse, and freelist protection, which are discussed as follows.

Randomized allocation. FreeGuard's randomized placement is not as strong as that of DieHarder and OpenBSD. DieHarder provides $O(\log N)$ bits of entropy for its randomized placement [31], where N represents the number of freed objects for this size class in existing miniheaps. OpenBSD-6.0 first chooses one-out-of-four lists, then allocates an object randomly inside a bag. Currently, the size of a bag is just one page. Thus, a bag can hold 256 objects with the size 16 bytes, and 2 objects with the size 2,048 bytes. Thus, the entropy associated with OpenBSD is currently between 3 bits to 10 bits. FreeGuard only has 2 bits of entropy, as it will choose one-out-of-four lists randomly.

Randomized memory reuse. DieHarder has the same entropy as its randomized allocation, $O(\log N)$ bits [31]. OpenBSD-6.0 has a delayed buffer with 16 entries, which will provide an entropy of 4 bits. After that, no randomization exists: a freed object will be always placed back into the same bag; it will always allocate an object from the first bag, if any objects are available there. FreeGuard does not use the delayed buffer, but will put a freed object into one-out-of-four lists randomly. FreeGuard introduces another mechanism to increase the complexity of attack upon memory reuses: freed objects will be utilized in a FIFO order, and FreeGuard may still allocate never-allocated objects, even when there are some freed objects in freelists. We cannot easily quantify this entropy, but it should not be worse than OpenBSD.

Freelist protection. The biggest problem of FreeGuard is that its freelists are not protected. Thus, if an attacker modifies them, they may take control of the heap allocator, and issue successful attacks afterwards. The relationship between FreeGuard's metadata and heap objects is computable, when the starting address of the metadata is known. If the attacker has permission to run a program on the target machine, he may be able to examine the contents of `/proc/PID/maps`, and identify the placement of the metadata. However, if the attacker is unable to run programs on the target machine, the randomization increases the complexity of such attacks. Both OpenBSD and DieHarder improve the protection of their metadata, since the relationship between heap objects and their metadata is actually stored in a hash map. OpenBSD dramatically increases the difficulty of attacks, since every bag and bag metadata have the same storage unit of one page.

7 RELATED WORK

7.1 Secure Heap Allocators

There are other secure heap allocators, apart from those discussed in Section 2. However, most focus primarily on a particular type of security issue.

Some previous work has focused on securing only the object metadata. Robertson et al. proposed to prepend canaries and checksums for the metadata in order to detect possible overflows [35]. Younan et al. proposed to secure the metadata, utilizing a hash table that is placed in a separate location [40]. Heap Server places the metadata in the separate address space of another process for better protection [20]. The dnmalloc allocator allocates a separate chunk to hold the metadata, and utilizes a separate lookup table to map the chunks to their metadata, which is similar to OpenBSD and DieHarder [39]. Although these works can prevent some metadata-related attacks, they cannot mitigate attacks toward the heap itself, such as use-after-free attacks.

Furthermore, some works aim to increase the non-determinism of memory allocations and reuses, including the changing of starting addresses [7, 37], and shuffling the reuse order of freed objects [20]. FreeGuard adopts some of these techniques, but provides more protections than these works.

7.2 Defending One Specific Type of Errors

Many security hardening techniques maintain and lookup the metadata at runtime to defend against certain problems. These examples include the checking of bounds information for validating array references [2, 3], the confirmation of type information to validate cast operations [24], and the collection of object pointer information to perform garbage collection [33].

Iwahashi et al. proposed a Petri-net based signature that helps to understand and detect double-free vulnerabilities [18]. Undangle assigns a unique label to each object, and tracks the propagation of these labels, employing dynamic taint analysis. Upon deallocation, Undangle determines unsafe dangling pointers based on the lifetime of dangling pointers [9]. FreeSentry protects against use-after-free vulnerabilities through compiler instrumentation [38]. FreeSentry tracks pointers pointing to every object, then invalidates these pointers when an object is freed, which imposes around 25% overhead on average. DangNULL prevents use-after-free and double-free vulnerabilities [23]. Similarly, DangNULL traces the relationship between pointers and objects, and nullifies those pointers when their pointing-to objects are freed. DangNULL also utilizes compiler instrumentation in order to collect the relationship between pointers and objects. Overall, DangNULL's performance overhead is between 22% to 105%.

However, these countermeasures typically defend against only one type of approach. In contrast, FreeGuard provides more comprehensive protection against a range of errors, with lower performance overhead.

7.3 Employing the Vast Address Space

Archipelego [28] trades the address space for security and reliability by randomly placing objects in the vast address space. Thus, the probability of overflowing real data can be effectively reduced. Cling also utilizes the vast address space to tolerate use-after-free problems [1]. FreeGuard utilizes the vast address space to map heap objects to their metadata through the shadow memory technique, in order to achieve better performance.

8 CONCLUSION

This paper presents a novel secure heap allocator, FreeGuard, which provides significantly better performance than existing secure allocators. FreeGuard designs a novel memory layout, reduces a large number of `mmap` calls, and borrows the "freelist" idea from performance-oriented allocators. Overall, FreeGuard imposes negligible performance overhead (less than 2%) over the Linux allocator, but features a range of additional security features. In contrast, the OpenBSD allocator (the representative secure allocator), imposes around 22% overhead on average, with the worst case running 3.9× slower than FreeGuard. Finally, we have released the source code of FreeGuard, which is available at https://github.com/UTSASRG/FreeGuard.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Hamed Okhravi, and anonymous reviewers for their valuable suggestions and feedback, which significantly helped improve this paper. We are also thankful to Emery Berger for his invaluable comments, especially regarding DieHarder and OpenBSD, and suggestions on an early draft of this paper. The work is supported by UTSA, Google Faculty Award, and the National Science Foundation under Grants No. 1566154 and 1453011. It is also supported by an AFOSR grant, FA9550-14-1-0119. The opinions, findings, conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1929820.1929836

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 263–277. https://doi.org/10.1109/SP.2008.30

[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 *Proceedings of the 18th conference on USENIX security symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 51–66. http://dl.acm.org/citation.cfm?id=1855768.1855772

[4] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. 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*. ACM Press, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[5] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. https://doi.org/10.1145/582419.582421

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

[8] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

[9] 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 (ISSTA 2012)*. ACM, New York, NY, USA, 133–143. https://doi.org/10.1145/2338965.2336769

[10] SQL Developers. How SQLite Is Tested. https://www.sqlite.org/testing.html.

[11] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. 2003. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES '03)*. ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/780732.780743

[12] Jason Evans. jemalloc. http://www.canonware.com/jemalloc/.

[13] Jason Evans. Scalable memory allocation using jemalloc. https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/.

[14] Yi Feng and Emery D. Berger. 2005. A Locality-improving Dynamic Memory Allocator. In *Proceedings of the 2005 Workshop on Memory System Performance (MSP '05)*. ACM, New York, NY, USA, 68–77. https://doi.org/10.1145/1111583.1111594

[15] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[16] Sanjay Ghemawat and Paul Menage. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[17] David R. Hanson. 1980. A portable storage management system for the icon programming language. , 489–500 pages. https://doi.org/10.1002/spe.4380100607

[18] Ryan Iwahashi, Daniela A. Oliveira, S. Felix Wu, Jedidiah R. Crandall, Young-Jun Heo, Jin-Tae Oh, and Jong-Soo Jang. 2008. Towards Automatically Generating Double-Free Vulnerability Signatures Using Petri Nets. In *Proceedings of the 11th*

International Conference on Information Security (ISC '08). Springer-Verlag, Berlin, Heidelberg, 114–130. https://doi.org/10.1007/978-3-540-85886-7_8

[19] Poul-Henning Kamp. malloc (3) Revisited. http://www-public.tem-tsp.eu/~thomas_g/research/biblio/2015/gidra15asplos-numagic.pdf.

[20] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. 2006. Comprehensively and Efficiently Protecting the Heap. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/1168857.1168884

[21] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 129–142. https://doi.org/10.1145/1065010.1065027

[22] Doug Lea. The GNU C Library. http://www.gnu.org/software/libc/libc.html.

[23] 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*.

[24] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 81–96. http://dl.acm.org/citation.cfm?id=2831143.2831149

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

[26] Tongping Liu and Emery D. Berger. 2011. SHERIFF: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048066.2048070

[27] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*.

[28] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2008. Archipelago: Trading Address Space for Reliability and Security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 115–124. https://doi.org/10.1145/1346281.1346296

[29] Otto Moerbeek. 2009. A new malloc(3) for OpenBSD. https://www.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf.

[30] NIST. National Vulnerability Database. http://nvd.nist.gov/

[31] 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

[32] Kipp Owens and Rajiv Parikh. 2012. 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/.

[33] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/1542431.1542438

[34] Sean Reifschneider. "Pure python memcached client". https://pypi.python.org/pypi/python-memcached.

[35] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. 2003. Run-time Detection of Heap-based Overflows. In *Proceedings of the 17th USENIX Conference on System Administration (LISA '03)*. USENIX Association, Berkeley, CA, USA, 51–60. http://dl.acm.org/citation.cfm?id=1051937.1051947

[36] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

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

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

[39] Yves Younan, Wouter Joosen, and Frank Piessens. 2006. Efficient Protection Against Heap-based Buffer Overflows Without Resorting to Magic. In *Proceedings of the 8th International Conference on Information and Communications Security (ICICS'06)*. Springer-Verlag, Berlin, Heidelberg, 379–398. https://doi.org/10.1007/11935308_27

[40] Yves Younan, Yves Younan, Wouter Joosen, Wouter Joosen, Frank Piessens, Frank Piessens, Hans Van Den Eynden, and Hans Van Den Eynden. 2005. *Security of memory allocators for C and C++*. Technical Report.

[41] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. 2011. Dynamic Cache Contention Detection in Multi-threaded Applications. In *The International Conference on Virtual Execution Environments*. Newport Beach, CA. http://groups.csail.mit.edu/commit/papers/2011/zhao-vee11-cache-contention.pdf