Preventing Use-After-Free Attacks with Fast Forward Allocation

Brian Wickman[†] Hong Hu[‡] Insu Yun Daehee Jang
JungWon Lim Sanidhya Kashyap[*] Taesoo Kim

[†]GTRI [‡]PennState GeorgiaTech [*]EPFL

Abstract

Memory-unsafe languages are widely used to implement critical systems like kernels and browsers, leading to thousands of memory safety issues every year. A use-after-free bug is a temporal memory error where the program accidentally visits a freed memory location. Recent studies show that use-after-free is one of the most exploited memory vulnerabilities. Unfortunately, previous efforts to mitigate use-after-free bugs are not widely deployed in real-world programs due to either inadequate accuracy or high performance overhead.

In this paper, we propose to resurrect the idea of one-time allocation (OTA) and provide a practical implementation with efficient execution and moderate memory overhead. With one-time allocation, the memory manager always returns a distinct memory address for each request. Since memory locations are not reused, attackers cannot reclaim freed objects, and thus cannot exploit use-after-free bugs. We utilize two techniques to render OTA practical: batch page management and the fusion of bump-pointer and fixed-size bins memory allocation styles. Batch page management helps reduce the number of system calls which negatively impact performance, while blending the two allocation methods mitigates the memory overhead and fragmentation issues. We implemented a prototype, called `FFmalloc`, to demonstrate our techniques. We evaluated `FFmalloc` on widely used benchmarks and real-world large programs. `FFmalloc` successfully blocked all tested use-after-free attacks while introducing moderate overhead. The results show that OTA can be a strong and practical solution to thwart use-after-free threats.

1 Introduction

Memory-unsafe languages, like C and C++, are widely used to implement key programs such as web browsers and operating systems. Therefore, we have seen innumerable memory safety issues detected and abused in these systems [20]. Among all memory safety issues, the *use-after-free* bug is one of the most commonly reported and exploited security problems [15, 20].

A use-after-free bug occurs when a program tries to deference a *dangling* pointer that points to a freed object. The consequence of a use-after-free bug depends on the implementation of the memory allocator and the following code of the program. In the worst case, attackers may reclaim the freed object and update its content with malformed values. When the program accidentally uses the content, its behavior will be under the control of attackers, potentially allowing arbitrary code execution or sensitive information leakage.

Researchers have proposed different methods to detect or mitigate use-after-free bugs. The first method is to label each memory chunk to indicate whether it is allocated or freed. Before each memory access, a label is checked to detect after-free use [21–23, 28, 33]. However, this approach introduces high overhead to program execution and thus has not been widely adopted in released applications. A second way is to actively invalidate dangling pointers once the object is freed, like setting them to a NULL value [7, 15, 35, 37]. These tools must maintain the inverse points-to relationship between an object and its references, which significantly slows down the protected execution. As a special case, Oscar [7] makes the freed objects inaccessible to achieve the same goal. A more recent approach ignores the normal free request, and utilizes spare CPU cores to independently identify and release garbage objects (*i.e.*, objects without any reference) [2, 18, 30]. This method requires extra computation resources, and may have limited scalability.

The limitations of previous proposals led us to rethink the defense against use-after-free bugs. Fundamental to exploiting a use-after-free bug is the attackers' ability to reclaim the freed memory and modify its content before the program uses it. As almost all memory managers reuse released memory for subsequent requests to improve efficiency [10, 12, 13], attackers can usually acquire the freed memory with trivial effort [9, 34, 36]. For example, glibc caches freed chunks in multiple linked lists (called *bins*) and reuses them to quickly respond to future requests. However, if a memory manager does not reuse any released memory, attackers will lose the ability to control the memory content associated with a dangling pointer. The program may run normally or crash (e.g., the freed memory has been unmapped), but all exploitation of use-after-free bugs will fail.

Inspired by this observation, we propose to resurrect *one-time allocation (OTA)* to prevent successful exploitations of use-after-free bugs. For any virtual address, an OTA allocator will assign it to the program only once and will *never reuse* it for other memory requests. In other words, every request will get a distinct memory chunk and no one will ever overlap with another. Note that an OTA allocator does not eliminate use-after-free bugs, but renders all of them unexploitable.

Although the idea is straightforward, developing a practical OTA manager is not easy. We identify three challenges that have to be handled properly. First, OTA may introduce high memory overhead. As the kernel manages memory at the page level (i.e., 4096 bytes by default), OTA cannot release the page as long as any byte is in use. In the worst case, it may waste 4095 bytes per page. Second, OTA is limited by the number of VMA structures in kernel. The Linux kernel creates a VMA structure for each set of continuous pages, and allows up to 65535 VMAs for each process. As OTA does not reuse memory, the in-use pages will scatter sparsely, until the process reaches the VMA limit. After that, the kernel cannot release any pages that would split a VMA into two. Finally, OTA may slow down the execution due to frequent system calls. Without address reuse, the program will continuously exhaust the pages allocated from the kernel, and have to make more system calls (e.g., mmap) to request new pages.

Our solution to these challenges is two-fold. First, we blend two allocation strategies to reduce the waste of memory and mitigate the VMA issue. For small requests, we use a size-based binning allocator to group similarly sized objects together; for large requests, we handle them in a continuous manner from a discrete region. By coalescing small allocations, we avoid the worst-case memory usage: tiny islands in between large allocations that hinder releasing pages and lead to heavy overhead. Meanwhile, continuously allocating large objects limits excess allocation to no more than minimal padding to comply with alignment requirements. Similar solutions are adopted by existing memory managers. However, we demonstrate that it makes OTA, a commonly believed impractical method, possible and useful. Second, we strategically batch memory mapping and unmapping to minimize the number of system calls. When FFmalloc requests memory from the kernel, it will ask for a much larger region than immediately necessary to handle the application's requirement. The additional amount of memory is cached internally to answer future requests. When the program frees memory, FFmalloc will not immediately invoke system calls to release the region. Instead, it will wait for several sequential freed pages and return them together with one system call.

We implemented Fast Forward Memory Allocation (FFmalloc), a prototype OTA, in 2,117 lines of C code. FFmalloc requests 4MB memory at a time from the kernel, and only releases freed memory when there are eight or more contiguous pages. For memory requests of less than 2K bytes, we use the binning allocator to group them together. For larger requests, we simply return available memory sequentially.

We applied FFmalloc on common benchmarks and real-world programs to understand its practicality and security. Specifically, we used FFmalloc to protect nine programs with eleven exploitable use-after-free bugs. With the protection of FFmalloc, all exploits failed. Upon manual inspection, we confirmed no overlap between any allocated objects. This result shows that FFmalloc can effectively prevent use-after-free attacks. To measure the overhead, we tested FFmalloc on SPEC CPU2006 benchmarks, the PARSEC 3 benchmark suite, the JavaScript engine ChakraCore, and the web server NGINX. On average (geometric mean), FFmalloc introduces 2.3% CPU overhead and 61.0% memory overhead to SPEC CPU2006 benchmarks. By comparison, the state-of-the-art tool, MarkUs, adds 14.8% CPU overhead and 28.1% memory overhead to the same set of benchmarks. Meanwhile, FFmalloc has 33.1% CPU overhead and 50.5% memory overhead on PARSEC 3 benchmarks, while MarkUs introduces 42.9% CPU overhead and 13.0% memory overhead. FFmalloc brings negligible overhead to ChakraCore, and provides similar performance as other secure allocators. These results show that FFmalloc is a practical solution to protect real-world programs against use-after-free exploits.

In summary, we make the following contributions:

- We propose to revive the idea of one-time allocation (OTA) to prevent use-after-free attacks. OTA provides efficient protection with a strong security guarantee.
- We designed and implemented FFmalloc, the first practical prototype of OTA, which supports both Linux and Windows applications.
- We extensively evaluated FFmalloc. The results demonstrate that OTA can be a practical way to protect real-world applications against use-after-free attacks.

We will release the source code of FFmalloc at https://github.com/bwickman97/ffmalloc.

2 Problem Definition

2.1 A Motivating Example

Figure 1 shows an example of the use-after-free vulnerability. The code defines two structures: Array to hold the user input, and Parser for the parser function. Both structures have the same size while Parser contains a function pointer handler. Function handle_net_input first dynamically allocates an instance of Parser and initializes its members, like setting the handler to function net_parser. Then, it tries to get a command from the client (line 18). If the command is PARSE, it will allocate an instance of Array (line 23), and will read *untrusted* user input from the client to the internal buffer of array (line 24). Finally, it invokes the parsing func-

```
1  typedef struct {long used; char buf[24];} Array;  // 32-byte
2
3  typedef struct {                                  // 32-byte
4    long status; void *start, *current;
5    int  (*handler)(void *buf);
6  } Parser;
7
8  enum Command { INVALID, PARSE, ... };
9  int net_parser(void *buf);
10
11 int handle_net_input(int client_fd) {
12   Parser *parser = (Parser *)malloc(32);      // allocation
13   parser->status = INIT;
14   parser->start = parser->current = NULL;
15   parser->handler = &net_parser;
16
17   enum Command cmd = INVALID;
18   read(client_fd, &cmd, sizeof(cmd));
19
20   switch (cmd) {
21     case INVALID: free(parser);               // missing break;
22     case PARSE:
23       Array *array = (Array *)malloc(32);     // re-allocation
24       read(client_fd, array->buf, 24);        // content changes
25       parser->handler(array->buf);            // use-after-free
26       free(parser); break;
27 }}
```

Figure 1: An example of use-after-free bugs. The `parser` object is freed at line 23 if the command is `INVALID`, but is used at line 25 for the indirect call. This bug is exploitable as attackers can change the object content at line 24 due to the memory reuse.

tion through `parser->handler` (line 25). If the command is `INVALID`, the code will free the object `parser` (line 21). Due to the lack of a `break` statement at line 21, the code at line 25 will use the freed `parser`, leading to a use-after-free vulnerability.

This use-after-free bug is exploitable and attackers can remotely execute arbitrary code. Specifically, when the object `parser` is freed at line 21, the memory manager (*e.g.*, glibc) will not immediately return the memory occupied by `parser` to the operating system. When the code at line 23 requests an `Array` object, the memory manager will reallocate the memory originally used by `parser` to `array`, as `Array` has the same size as `Parser`. Now `array` and `parser` point to the same memory location. The `read` function at line 24 will fill the `array->buf` with the untrusted user input, which will effectively overwrite the members of `parser`, including the function pointer `handler`. When the code invokes the parser handler, it will jump to any location specified by the attacker, resulting in a control-flow hijacking attack.

2.2 Use-After-Free Bugs and Exploits

Use-after-free bugs may lead to different consequences depending on the logic of the program and the memory manager. We summarize the possible consequences in Table 1. If the system has removed the permission to access the corresponding memory (S1), the after-free use will trigger an access violation and cause the program to crash. If the memory is still accessible and the memory has not been reallocated to other objects (S2), the obsolete content of the freed object will be used. If in the interim the memory has been allocated to other objects (S3), the content of the new object will be used instead. In the last two cases, depending on the retrieved value and its usage, the program may crash, produce wrong results, or work "well" without any observable anomaly. The example in Figure 1 falls into S3, where the new object `array` occupies the memory originally allocated to `parser`.

Table 1: Consequences of use-after-free bugs. Depending on the memory allocator and the program logic, attackers may launch severe attacks, including code injection and information leakage.

	Corresponding memory	**After-free use**	**Exploitable?**
S1	Inaccessible	Crash	No
S2	Accessible & never reused	Get old content	No (w/o metadata writing)
S3	Accessible & reused	Get new content	Possible

A use-after-free in S3 is likely to be exploitable. A bug in S1 is not exploitable as it always causes the program to crash. In S2, the exploitability of the bug depends on the memory manager and the program logic. If the memory manager makes no change to the freed region, the program will remain well-behaved as the freed region continues to have a validly formed object. Until the memory manager unmaps the page (moving into S1), it is as if the application never freed the object. However, if an allocator alters the freed block, like storing some metadata, an attacker may abuse this behavior to exploit the bug. For example, they might be able to modify free list metadata to achieve arbitrary memory write [29]. By contrast, in S3, an attacker can reclaim the memory and fill in new content, thus affecting the following usages.

To exploit a use-after-free bug in S3, attackers have to follow the pattern of `free-reallocate-use`. In the first step, they trigger the program to free a vulnerable object. Then, they request a similar-sized object to obtain the freed memory. They fill the memory with contextually appropriate data. For example, in Figure 1, attackers will overwrite the function pointer `handler` in `parser` to a different address, like `system`. Finally, when the program reads the memory, the malicious content will be retrieved and used to launch the attack. In Figure 1, the `free-reallocate-use` pattern can be mapped to line 21, lines 23-24 and line 25.

2.3 Approach Overview

Of the three steps of a successful use-after-free exploit, `free-reallocate-use`, `reallocate` is the most unique behavior triggered by attackers. If we can prevent the reuse of freed objects, attackers will not be able to re-occupy the freed memory and cannot change the content. In that case, an exploitable use-after-free bug will not be exploitable any more. While the program may run well, abnormally, or even simply crash, it is out of the attacker's control. We call this memory management method *one-time allocation (OTA)*.

Although the idea of OTA is straightforward, it is nontrivial to build a practical OTA allocator. Previous works explored ideas similar to OTA, but they either failed to provide sufficient security or imposed unacceptable performance penalty. DieHarder allocates memory at randomized addresses [25], but this only provides a probabilistic assurance that memory chunks will not overlap with each other. Archipelago places each allocation on a distinct physical page [19], while Oscar simulates the same object-per-page strategy by masking the allocation through virtual pages [7]. However, creating these shadow pages can introduce more than 40% overhead due to frequent system calls. Cling prevents memory reuse between mismatched types [3], but leaves space to exploit use-after-free bugs within compatible types. In the original paper, the author of Cling discussed the idea of one-time allocation, but he treated it as an impractical solution due to heavy memory usage.

Despite the unpleasant history, we notice that OTA still has genuine merit: a straightforward design and strong security guarantee. Without needing complicated intelligence or external system dependencies, OTA can eliminate the threat of use-after-free bugs. The design also helps avoid careless errors in implementation. Therefore, we explored different choices to mitigate the remaining challenge of overhead while retaining the security benefit of OTA. Fortunately, we found a set of solutions that enable a practical OTA implementation. Our results in §6 show that the overhead of `FFmalloc` is minimal in the vast majority of cases.

2.4 Threat Model

Before exploring the design space, we define the threat model where OTA aims to protect benign programs. We assume that a program contains one or more use-after-free bugs, and some of them are exploitable. Other vulnerabilities, like buffer overflows or type errors, are out of the scope of this work. We assume attackers can only exploit use-after-free bugs. Other bugs cannot be used to bypass or corrupt the OTA memory manager. Our threat model is consistent with previous proposals on use-after-free defense [2, 15, 23, 30, 35, 37].

3 Design Space Exploration

3.1 Forward Continuous Allocation

In our first design attempt, we implemented a *forward continuous* allocator, called `FCmalloc`. `FCmalloc` uses a pointer to track the end of the last allocation. For a new memory request, it simply advances the pointer by the requested size and returns the old value. Since the pointer is only incremented, any call to `malloc` will get a distinct region. When the pointer reaches the end of the mapped pages, `FCmalloc` will request additional pages from the operating system through the `mmap` system call. For each `free` request, `FCmalloc` releases all completely free pages (*i.e.*, no byte is in use) to the system with the `munmap` system call. The simple design of `FCmalloc` enables compact allocation, where each allocated chunk immediately follows the previous one.

Table 2: `VMA` issue of `FCmalloc` on SPEC programs. Due to the forward allocation, programs with `FCmalloc` require more `VMA` structures. Batch processing can help mitigate this issue. FC-X means we only release at least X continuous freed pages.

Benchmark	glibc	FCmalloc	FC-2	FC-8	FC-32
perlbench	4,401	58,737	46,299	23,171	9,321
bzip2	23	35	35	35	35
gcc	2,753	6,525	4,665	3,120	1,854
mcf	20	31	30	30	30
milc	46	65	65	65	65
namd	128	57	56	56	56
gobmk	25	61	57	52	48
dealII	4,760	2,322	1,052	338	326
soplex	152	99	96	93	89
povray	51	109	89	74	57
hmmer	35	197	183	145	114
sjeng	20	32	32	32	32
libquantum	29	38	38	35	35
h264ref	228	89	83	80	80
lbm	23	34	34	33	33
omnetpp	1,164	15,933	15,040	13,728	12,521
astar	1,762	6,726	5,370	3,703	2,790
sphinx3	168	31,409	31,382	31,064	9,022
xalancbmk	2,705	**68,606**	48,526	34,826	23,434

`FCmalloc` is the most straightforward way to implement OTA. However, after applying it to the SPEC CPU2006 benchmarks, we identify three challenges that limit its practicality.

Performance Overhead. `FCmalloc` has high performance overhead due to the frequent `mmap/munmap` system calls. For example, given the input file `c-typeck`, the `gcc` benchmark will send 831,410 `mmap/mumap` system calls to the Linux kernel, leading to 40.8 seconds spent in the kernel space. With glibc, `gcc` only issues 57 such system calls which merely cost 0.59 seconds. The overall overhead is 60.2% for `c-typeck`.

Memory Overhead. `FCmalloc` can lead to significant memory overhead compared to the standard C allocator. Since the OS only allocates or releases memory on page granularity, all 4096-bytes of a page cannot be returned as long as one byte is still in use. Even worse, if a small allocation straddles the boundary between two pages, then neither page can be freed.

VMA Limit. Frequent memory release with `munmap` could exhaust the `VMA` kernel structures. The Linux kernel creates a `VMA` structure for each set of contiguous pages. When `FCmalloc` releases a page that is in the middle of some continuous pages, the Linux kernel will split the corresponding `VMA` in two. By default, Linux allows at most 65,535 `VMA` structures for each process. Once this limit is reached, no pages can be released unless they are at the margin of an existing `VMA`. This behavior exacerbates the memory overhead of the process. Table 2 shows the number of `VMAs` used by SPEC benchmarks. As we can see, `FCmalloc` increases the number of `VMAs` for most of the programs. In the worst case, it causes `xalancbmk` to use 68,606 `VMAs`, exceeding the default limit of the Linux kernel, while the original glibc only requires 2,705 `VMAs`. Therefore, `FCmalloc` introduces high memory overhead to `xalancbmk`. Other programs incurring high `VMA`-usage include `perlbench`, `omnetpp` and `sphinx3`.

3.1.1 Mitigation: Batch Processing

We find that batch mapping and unmapping can help mitigate the aforementioned challenges. When requesting memory from the kernel, we can specify a large number of pages using mmap at one time. FCmalloc then handles malloc with this region until this batch of pages is used up, at which time FCmalloc will issue another mmap request. When the program tries to free a chunk of memory, FCmalloc checks if this will create a set of continuous freed pages. If so, we can release them together with one munmap system call. Batch processing effectively reduces the performance overhead of FCmalloc, as it can significantly reduce the number of system calls. For the example of gcc, when FCmalloc only releases at least 32 freed pages, we can save 471,144 munmap system calls (58.7%). Reduced system calls can also help mitigate the VMA issue. As shown in Table 2, the VMA overhead of FCmalloc keeps decreasing if we release memory less often but with more pages. For example, when FCmalloc only releases 32 pages, we can save 45,172 VMAs from xalancbmk (65.8% reduction). For perlbench and omnetpp, batch processing helps reduce the VMA counts to a normal range. However, batch processing will enlarge the memory usage of the protected execution, as batch mapping introduces mapped-but-not-allocated memory and batch unmapping brings freed-but-not-unmapped pages.

3.2 Forward Binning

Our second design attempt was a *forward binning* allocator, called FBmalloc. In contrast to FCmalloc, FBmalloc groups allocations of similar sizes into a bin. This design is usually called a *BiBop* allocator - a big bag of pages [3, 10, 12, 25]. FBmalloc creates several bins for allocations with less than 4096 bytes. All allocations within a bin will get the same-size chunks. For allocations greater than 4096 bytes, FBmalloc rounds the size up to the next page size, and directly uses mmap to request new pages. FBmalloc uses one page per small bin at a time to serve the malloc request. Once a bin is fully allocated, FBmalloc uses mmap to request an additional page from the kernel. Only when the bin is fully freed, will it be released to the system. Requests for different sizes will get memory from different bins, and thus the return value of malloc might not be strictly increasing. However, FBmalloc is still a valid OTA, as it never reuses bins and takes the forward continuous allocation to manage memory within a bin.

FBmalloc helps mitigate the memory overhead issue of FCmalloc, especially when small allocations have a longer lifetime than large ones. In the omnetpp benchmark of SPEC CPU2006, certain small objects are rarely freed, while around them are large objects with shorter lifetime. This leads to heavy memory overhead with FCmalloc. With FBmalloc, these particular allocations are co-located on a much smaller number of pages (bins), which effectively limits the overhead.

3.3 Allocator Fusion

Our final design, called FFmalloc, marries the ideas of forward continuous allocation and forward binning allocation to take advantage of their strengths. FFmalloc handles allocations up to 2048 bytes via the binning allocator. This prevents small long-lived allocations holding onto freed pages. FFmalloc serves large allocations from the continuous allocator to minimize allocation waste due to alignment requirements.

FFmalloc assigns each allocator a *pool*, which contains several continuous pages. A pool is the basic memory unit that FFmalloc requests from the OS, and its size is configurable during compilation. Currently, we set the pool size to be 4MB. For allocations larger than the pool size, FFmalloc will create a special pool that is just large enough for the request, called a *jumbo* pool. Only one pool can be assigned to the binning allocator, while the continuous allocator can have multiple pools. FFmalloc associates each CPU core with a distinct pool to reduce lock contention. When the remaining space in a pool is insufficient to satisfy a request, FFmalloc creates a new pool. FFmalloc will continue to place future smaller allocations in the original pool, until the pool is filled or too many pools have been created. We create the first pool offset from the end of the existing heap region. This allows an application to use both the glibc allocator and FFmalloc at the same time. This design also makes the starting address of FFmalloc randomized, preserving the security benefits of ASLR. FFmalloc ensures that the kernel supplies pages at increasingly higher addresses by specifying the MAP_FIXED_NOREPLACE flag for mmap. With this flag, the kernel tries to map the memory at the specified location and returns an error if such placement conflicts with an existing mapping. FFmalloc keeps probing forward until it finds an available address for the next pool.

4 Implementation

4.1 Metadata of Allocations

FFmalloc tracks its allocations in different metadata structures. Each pool of the continuous allocator has an array of allocated addresses. For each allocation, FFmalloc appends the return value of malloc to the array. Since FFmalloc allocates memory forward, the array is naturally sorted and allows for efficient searching. We can obtain the size of each allocation by subtracting its address from the next entry in the array. As all allocations from FFmalloc are 8-byte aligned, the last three bits are always zeros. We use the last two bits to indicate the status of a memory chunk: 00 means the memory is in use; 01 means the memory is freed and safe to release; 11 means FFmalloc has released some pages in this allocation. The pool of the binning allocator has an array of structures, with one entry per page. This structure records the allocation size, the next unused chunk, and a bitmap to maintain the status of each chunk, where 1 means in use and 0 means freed.

FFmalloc connects each memory chunk to its metadata through a central three-level trie. The key of the mapping is the address of each memory chunk, while the value points to the metadata structure of the pool. That structure records the start and end address of the pool, the type, the next available address or page, pointers to the type-specific metadata, and unmapped regions. Jumbo pools only have type information, start address, and end address. All metadata is stored separately from the pool, like other secure allocators.

4.2 Freeing Memory

When the program calls free to free memory, FFmalloc first locates the metadata of the pool, and marks the corresponding slot in a bitmap of the binning pool, or updates the pointer in the array of a continuous pool. Following the principle of batch processing, it will not immediately release the memory to the system. Instead, it waits for enough continuous pages to be freed and returns them with one munmap system call. Before compilation, we can change the minimum number of continuous pages for a munmap system call. Increasing this threshold trades memory for speed. As will be shown in §6.4, we evaluated the tradeoff and empirically picked eight as the default value in the current implementation.

Detecting double-free and invalid-free. While not an original design goal, the metadata of FFmalloc helps us detect double-free or invalid-free bugs. Double-free bugs free a dangling pointer, which may corrupt the allocator's metadata and lead to further attacks, like arbitrary memory access. Invalid-free bugs instead free an address not allocated by malloc. When the program invokes free, FFmalloc first locates the metadata corresponding to that address, and then checks whether the underlying memory has been freed. It will find the problem and terminate the execution if the program tries to free a dangling pointer, or an invalid address.

Lazy free. The straightforward way to return pages to the system is to invoke the munmap system call, where the kernel will immediately release the memory. However, starting from version 4.5, the Linux kernel provides an alternative way for lazy memory release. Specifically, if we provide the MADV_FREE flag to the madvise system call, the kernel will only reclaim the pages during heavy memory pressure. We provide both implementations to use madvise or munmap to release freed pages. When madvise is used, FFmalloc will completely unmap the pages once all allocations in a pool are freed. In §6.4, we will evaluate the benefits and limitations of each method by evaluating them on the SPEC CPU2006 benchmarks.

4.3 Reallocation under OTA

Function realloc allows an application to change the size of an existing allocation. If the new size is smaller than the old one, a memory allocator can just shrink the size and return quickly. But if the new size is larger, the allocator must check whether the memory after the current allocation has enough space for the extra bytes. If so, the allocator will just increase the size in its metadata and immediately return. In the worst case, the allocator has to allocate a new, large-enough memory chunk, copy the existing content into the new one, free the old allocation, and return the new address.

FFmalloc has to take a different approach since we want to avoid reusing any memory. As all allocations in a bin share the same size, growing an allocation beyond the size always requires a new allocation. For a continuous allocation, growing the allocation proceeds as the traditional method, except that the following memory must have not been allocated. Due to the no-reuse constraint, realloc in FFmalloc is more likely to perform a reallocation than other allocators. If the program has many invocations to realloc, we can expect FFmalloc to impose noticeable overhead.

4.4 Supporting Multi-threaded Applications

To improve the allocation speed on multi-threaded applications, the binning allocator in FFmalloc borrows the *thread caching* technique from tcmalloc [12] for lock-free allocation. Thread caching creates a distinct cache for each thread of the process. Each call to malloc is served by the corresponding thread-specific cache. Since caches are not shared between threads, there is no risk of contention and thus no need for locks. As noted earlier, FFmalloc creates one pool at a time for all binning allocators of all threads. Then, we split the pool into same-sized pieces, and assign pieces when threads are started or consume all previously assigned pages. In this way, each thread has its own memory space for binning allocation.

Marking a small allocation as free is also lockless. To clear the corresponding bit in the bitmap, FFmalloc uses an atomic bitwise and operation to guarantee the operation safety. When we see a particular number of free pages, FFmalloc frees small allocations after requiring a lock. Allocations from the continuous pool are not handled by the thread cache. Since simultaneous allocations from multiple threads could attempt to read and update the next allocation pointer at the same time, a race condition is possible. Therefore, we protect these allocations via locks. However, to reduce lock contention, FFmalloc has a configurable option to have a continuous pool per CPU core, turned on by default. The allocator will identify the CPU core currently executing the thread, and allocate the memory from the continuous pool of that core. This allows multiple threads to execute in parallel with reduced risk of contention at the cost of additional memory overhead.

5 Security Evaluation

We tested FFmalloc on real-world vulnerable programs to demonstrate its ability to prevent use-after-free attacks (§5.1).

Table 3: Preventing UAF attacks. We collected working exploits from six real-world UAF vulnerabilities and five CTF challenges. ✔ means attackers can successfully launch the attack, while ✗ means `FFmalloc` prevents the exploit. `FFmalloc` successfully prevents all of them.

Program	ID	Bug Type	Link	Original Attack	With the Protection of `FFmalloc`
`babyheap`	CTF challenges	UAF ⟶ DF	↗	✔ Arbitrary code execution	✗ Exception due to failed info-leak
`childheap`		UAF ⟶ DF	↗	✔ Arbitrary code execution	✗ DF detected
`heapbabe`		UAF ⟶ DF	↗	✔ Arbitrary code execution	✗ DF detected
`ghostparty`		UAF	↗	✔ Arbitrary code execution	✗ Exception due to failed info-leak
`uaf`		UAF	↗	✔ Arbitrary code execution	✗ Exploit prevented due to new realloc
`PHP 7.0.7`	CVE-2016-5773	UAF ⟶ DF	↗	✔ Arbitrary code execution	✗ Exploit prevented & DF detected
`PHP 5.5.14`	CVE-2015-2787	UAF	↗	✔ Arbitrary code execution	✗ Assertion failure (uncontrollable)
`PHP 5.4.44`	CVE-2015-6835	UAF	↗	✔ Arbitrary memory disclosure	✗ Exploit prevented & run well
`mruby` 191ee25	Issue 3515	UAF	↗	✔ Arbitrary memory write	✗ Exploit prevented & run well
`libmimedir 0.5.1`	CVE-2015-3205	AF ⟶ UAF	↗	✔ Arbitrary code execution	✗ Exploit prevented & run well
`python 2.7.10`	Issue 24613	UAF	↗	✔ Restricted memory disclosure	✗ Exploit prevented & run well

UAF: Use-After-Free, **DF**: Double Free, **AF**: Arbitrary Free

We also performed a study to understand its design benefit and implementation security (§5.2).

5.1 Preventing Use-After-Free Attacks

We broadly searched in public vulnerability databases, like Exploit-DB ↗ and HackerOne ↗, and collected six exploits for four real-world applications, including the PHP language interpreter `PHP`, the Ruby language runtime `mruby`, the MIME directory parser `libmimedir` and the Python language interpreter `python`. In addition, we also selected five vulnerable challenges from popular capture-the-flag (CTF) games. We fed each exploit to the original and the `FFmalloc`-protected programs and analyzed the consequence.

Table 3 shows our protection result: `FFmalloc` successfully prevents all `11` use-after-free attacks. The `Type` column shows the type of bug, where → means the first bug leads to the second one. For example, UAF→DF indicates that the original use-after-free bug leads to a double-free vulnerability. Originally, each exploit successfully launches a malicious action, like arbitrary code execution, taking over the instruction pointer, or arbitrary memory writing. With the protection of `FFmalloc`, the execution either runs "normally" to the end, or crashes in the middle. We manually inspected the final execution state running with `FFmalloc` to understand the reasons. When the freed page had been unmapped, the after-free use triggered an invalid access exception. Otherwise, the after-free use succeeded and the execution continued. In that case, since the retrieved value remained unmodified, the program could run well to the end (five executions). However, since the attackers made certain assumptions on the value type, like a pointer, the executions also crashed due to failed pointer dereferences (two executions), or by assertion failure (one execution). For the other three executions, `FFmalloc` detected the double-free issues and proactively terminated the processes.

Case Study: PHP. CVE-2015-6835 is a use-after-free bug in the `unserialize` feature of `PHP` ↗. With a crafted session string, an attacker can keep a reference of a `zval` object even after the memory is freed. A proof-of-concept (PoC) exploit of this vulnerability is available online ↗. With the PoC, `PHP` first frees the object, and then reallocates the freed memory to hold the input data, which overwrites the object with the malicious content. When the program uses the freed memory, function `session_decode` will return arbitrary memory contents, like pointer values, where further attacks (e.g., code execution) could be constructed accordingly. After we used `FFmalloc` to replace the default memory allocator `Zend`, the vulnerable `PHP` failed to retrieve any data and showed identical behavior to the patched version.

Case Study: mruby. `mruby` is a lightweight runtime for the Ruby language ↗. It has an exploitable use-after-free vulnerability in commit 191ee25 ↗. The original PoC exploit from HackerOne causes execution to crash as it modifies a pointer inside the freed object to an invalid address. We updated the PoC to make it an arbitrary memory write primitive such that the new PoC makes the `proc` object point to a fake data segment controlled by attackers. From there `mruby` finally jumps into the `OP_MOVE` opcode handler with our fake virtual register values and writes the memory of our choice. We ran `mruby` with `FFmalloc` and launched the attack. This time, the runtime shut down gracefully as it could not parse the supplied exploit code. We confirmed that the use-after-free bug was triggered during the execution. However, since `FFmalloc` does not reuse memory, the old content of the freed object was used, and thus nothing critical happened.

CTF Challenges. We applied `FFmalloc` to five Capture-The-Flag (CTF) challenges that have use-after-free bugs. Although CTF problems are smaller than real-world programs, their authors often add uncommon challenges to increase the difficulty of the exploitation. We tested `FFmalloc` on CTF problems to cover edge cases that are missed in real-world programs. The original PoCs abused allocator-specific structures to execute shell commands. We further developed three new PoCs that purely utilized the program's structure to achieve arbitrary memory write and control-flow hijacking, which were independent of characteristics of specific heap alloca-

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void* p[256];
4  int main() {
5    p[0] = malloc(842373);    // allocate p[0]
6    p[1] = malloc(842389);
7    free(p[1]);
8    free(p[0]);               // free p[0], but not nullify
9    p[2] = malloc(842373);    // the same as p[0]
10   return 0;
11 }
```

Figure 2: PoC of the bug in MarkUs. By design, MarkUs does not release an object O if it can find any reference of O from stack or global memory. In this PoC, although `p[0]` still points to the first allocated object, MarkUs returns the same address at line 9.

tors. As shown in Table 3, `FFmalloc` successfully prevented all exploits. Even with a deep understanding of each problem, we could not find ways to bypass `FFmalloc`.

5.2 Secure for Deployment

`FFmalloc` provides robust security by the virtue of its straightforward design. Its security guarantee is based on a simple, easy to reason about proposition. Its implementation is straightforward, avoiding the complicated logic of memory-recycling code. By contrast, other defenses against use-after-free bugs, such as pointer invalidation [15,35] and garbage collections [2,18,30], rely on the soundness and completeness of their analyses for security. Unfortunately, it is very challenging to correctly implement such sophisticated techniques, and any mistakes in implementations can lead to severe security flaws, some even breaking their guarantee.

To understand the security status of different secure allocators, we ran `ArcHeap` [39] on each implementation for 24 hours to find potential issues. `ArcHeap` can automatically find heap exploitation techniques of an allocator, which can be developed further into powerful primitives (e.g., arbitrary writes). After the 24 hours of testing, `ArcHeap` did not discover any security issue from `FFmalloc`, demonstrating `FFmalloc`'s robustness on memory corruption vulnerabilities.

Other tools that rely on complex analyses are not as robust as `FFmalloc` in their security guarantee given certain implementation issues. For example, we discovered that MarkUs [2], which uses garbage collection techniques to prevent use-after-free vulnerabilities, failed to protect large-size blocks. Figure 2 shows the proof-of-concept (PoC) to trigger this bug. By design, MarkUs will not release an object O if it can find any reference of O from the stack, global memory, and registers. In the example code, `p[0]` holds the pointer value of the first allocated object by line 5. Even if this object is freed at line 8, the global array `p` still contains its reference at the 0th element. In theory, MarkUs should hold the memory and not allocate it to another request. However, when we requested the same-size memory at line 9, MarkUs returned the same address as `p[0]`, which means the memory had been released and reused. In the case when the dangling pointer `p[0]` is used after free, attackers can exploit the bug to launch attacks.

Moreover, after additional manual analysis, we found that MarkUs's management for quarantined objects simplified the exploitation of use-after-free bugs. To manage the quarantined objects, MarkUs writes an encoded pointer to the first eight bytes to freed objects to track the next chunk in the quarantine list. Unfortunately, the first eight bytes of a polymorphic object in C++ is the pointer to its virtual function table (vftable). Thus, the freed object's vftable will become the encoded pointer by MarkUs. The encoded pointer had predictable most significant bits because its `xor` encoding used a magic value (since corrected to point to an invalid address region). Therefore, attackers did not have to reclaim the freed object as in an ordinary exploitation scenario. Instead, they just had to spray the memory pointed by the encoded pointer with fake function pointers. When the program used freed memory, it would take the fake, malicious function pointers for indirect calls, leading to an arbitrary code execution attack. Figure 8 in the Appendix shows a proof-of-concept code about the exploitation. We responsibly reported these issues to the developers and they have since been patched [16,17,38]. `FFmalloc` will not have this kind of critical failure as it does not write any metadata to freed chunks.

6 Performance Evaluation

We evaluated `FFmalloc` on commonly used benchmarks and real-world programs to understand its overhead on single-threaded applications (§6.1), multi-threaded applications (§6.2) and large applications (§6.3). We also explored different values for certain internal settings to find the optimal configuration that makes `FFmalloc` achieve a balance between performance and memory usage (§6.4).

Benchmarks. To measure the overhead of `FFmalloc` and find the optimal settings, we selected four sets of benchmarks: SPEC CPU2006 with 19 single-threaded C/C++ programs, PARSEC 3.0 with 15 multiple-threaded workloads the JavaScript engine `ChakraCore` and the web server `NGINX`. We excluded the `raytrace` workload from PARSEC 3.0, as the compiled binary hangs on our system [1].

Setup. We performed our evaluations on a 64-core machine running Ubuntu 18.04, with Intel CPU E7-4820 at 2.00GHz and 256GB memory. We compiled all benchmarks with their default configurations, except the x264 benchmark in PARSEC 3.0. The default `-O3` optimization lead to some crashes, and we used `-O2` instead to avoid the problem [11]. We set the environment variable `LD_PRELOAD` to the OTA library so that the same compiled binary was used with our hardened memory manager and other allocators. We set `FFmalloc` to its default setting, which releases at least eight consecutive free pages to the system using the `munmap` system call. §6.4 evaluates the different settings and discusses our choice. During the

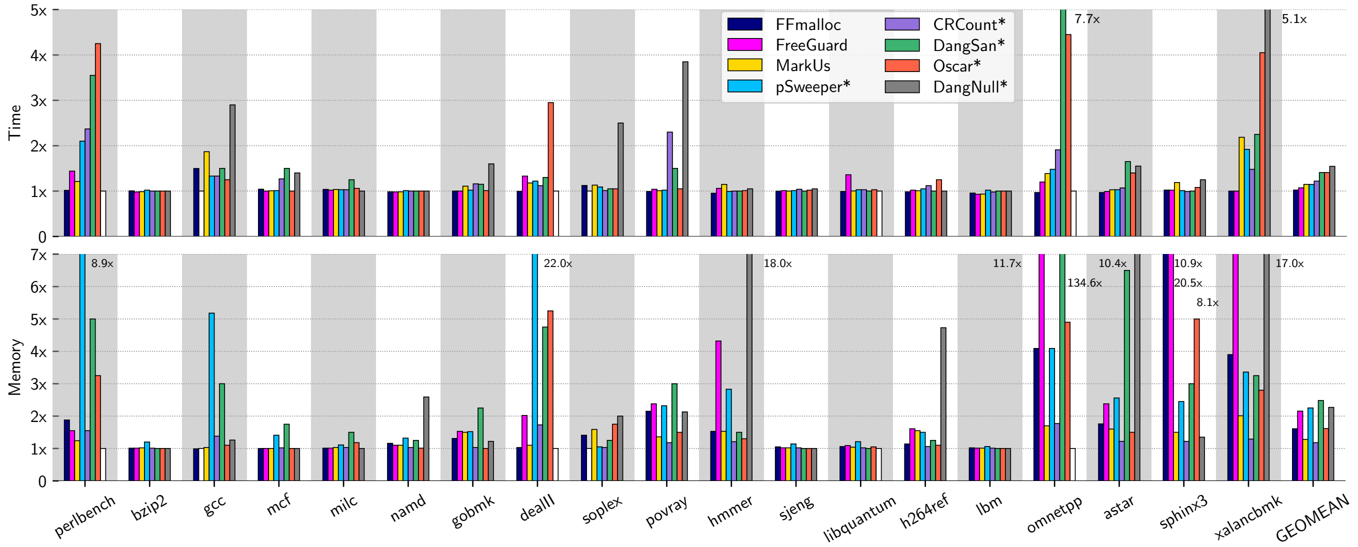

Figure 3: Overhead for SPEC CPU2006. 1x means no overhead. `FFmalloc` uses `munmap` to release memory to the system, 8 pages at a time. For tools with *, we use the results reported in the literature. DangNull did not report overhead for `perlbench`, `dealII`, `libquantum` and `omnetpp`. The `gcc` and `soplex` tests crash when running with FreeGuard. Placeholders for missing results are in white.

execution, we used the utility `time` to get the execution time and the maximum resident set size (RSS), which describes the maximum memory usage. If one program has multiple inputs, we use the maximum one among all RSS values.

6.1 Single-threaded Benchmarks

We measured the overhead of `FFmalloc` on SPEC CPU2006 benchmarks and compared it with seven previous works: MarkUs [2], FreeGuard [31], CRCount [30], pSweeper [18], Oscar [7], DangSan [35] and DangNull [15]. We successfully reran MarkUs and FreeGuard on our machine. Although DangSan is open-sourced, we could not get it to compile on Ubuntu 18.04. Since other works have not released their source code, we have elected to reuse the reported numbers from the literature. We ran each benchmark three times and averaged the results. Figure 3 shows the performance and memory overhead of each tool on each SPEC C/C++ benchmark. A white bar means either the original paper did not include the number, or the program crashed during the execution.

Performance Overhead. Considering the geometric mean, `FFmalloc` introduces 2.3% overhead to SPEC benchmarks, the lowest one among eight tools. On the same platform, MarkUs imposes 14.8% overhead and the overhead of FreeGuard is 7.2%. However, FreeGuard causes two programs to crash, specifically `gcc` and `soplex`. Since `gcc` usually shows very high overhead, the overhead of FreeGuard could be higher. pSweeper reported a similar slowdown as MarkUs, while CRCount claimed 22.0% overhead. Both Oscar and DangSan reported about a 40.0% cost. DangNull reported about 54.6% overhead for 15 out of 19 benchmarks. This result shows that with our careful design, one-time allocation can have counterintuitively low performance overhead.

`FFmalloc` introduces consistent overhead to 19 SPEC benchmarks, with the standard deviation 0.12, while other tools have standard deviations from 0.15 (FreeGuard) to 1.25 (DangNull). Among all benchmarks, `gcc` is an outlier where `FFmalloc` makes it slower by 49.8%. We investigated `gcc`'s execution, and found that it consumes the largest amount of memory per second (see the last column of Table 7 in Appendix A). Therefore, the execution with `FFmalloc` leads to significantly more system calls for memory management. For example, for the c-typeck input, `FFmalloc` issues 28,767 `mmap` and 500,213 `munmap` system calls, and spends 39.8 seconds in the kernel space. The original memory allocator `glibc` only requires 34 `mmap` and 23 `munmap`, which takes 0.59 seconds to finish. Although the user-space execution with `FFmalloc` is faster (reduced from 59.0 seconds to 53.9 seconds), the overall overhead is 53.7%. In this extreme case, we may need to optimize our settings to make a new balance between memory usage and performance overhead (see §6.4). Fortunately, we have not seen another program like `gcc` that so quickly allocates a large amount of memory. Further, other tools also demonstrate higher overhead for `gcc`, although the underlying reasons could be different.

Memory Overhead. Considering the geometric mean, `FFmalloc` introduces 61.0% memory overhead to SPEC benchmarks, similar to that of Oscar. Two previous tools achieve less overhead than `FFmalloc`: 18.0% for CRCount, and 28.1% for MarkUs (rerun). Another four tools consume more memory: 115.4% for FreeGuard (rerun), 125.2% for

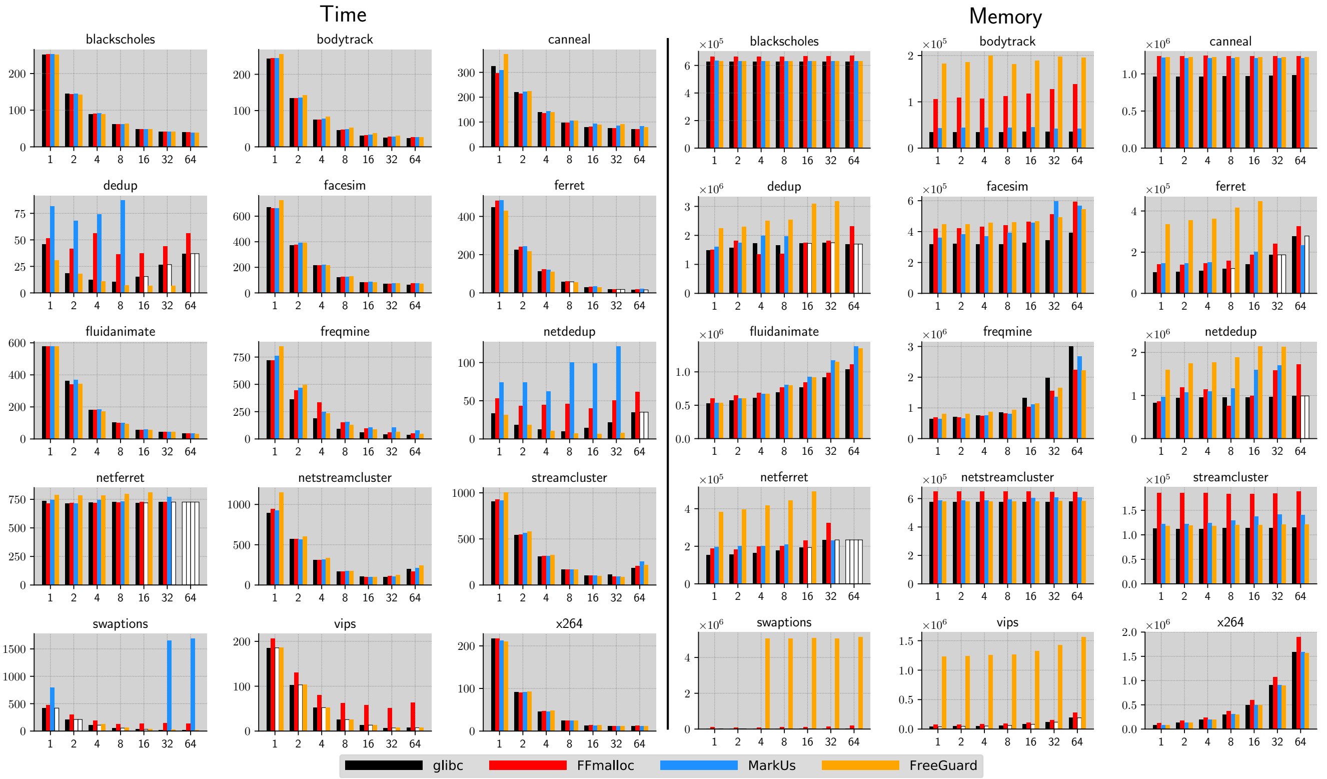

Figure 4: Overhead on PARSEC 3 with various CPU cores. White bar means the execution hangs or crashes.

pSweeper, 127.1% for DangNull and 148.1% for DangSan. In this evaluation, we configured `FFmalloc` to release memory only if there are eight consecutive freed pages. If we release memory more aggressively, `FFmalloc` will have lower memory overhead and higher performance overhead. We will discuss the tradeoff between time and memory in §6.4.

An apparent observation from Figure 3 is that the memory overhead is more diverse than the performance overhead. The values of the standard deviation range from 0.24 (CRCount) to 30.37 (DangSan), while the maximum standard deviation of the performance overhead is merely 1.25 (DangNull). The common pattern is that, for some benchmarks, most of the tested tools show significantly higher memory overhead than that on other benchmarks. Taking `omnetpp` as an example, `FFmalloc` and pSweeper consumes about 4.0× of the original memory; FreeGuard requires 11.7×; DangSan takes 134.6×; Oscar spends 4.9×; CRCount and MarkUs consume about 1.7× of the original memory. The extreme memory overhead is likely due to the characteristic of the program. There is no tool that always outperforms others on memory usage.

6.2 Multi-threaded benchmarks

We ran 15 benchmarks with seven different core counts, specifically, 1, 2, 4, 8, 16, 32 and 64, together with four memory managers: glibc, our `FFmalloc`, MarkUs, and FreeGuard. All benchmarks and core count combinations ran successfully using glibc or `FFmalloc`, except `netferret` with 64 cores which alway hung. In comparison, MarkUs failed 19 executions, while FreeGuard failed eight. All failures happen while running `dedup`, `ferret`, `netdedup`, `netferret`, `swaptions` and `vips`. In fact, MarkUs and FreeGuard had multiple random crashes during other executions. To get meaningful results, we ran each instance ten times, and reported the first three successful executions. While still widely used in the literature, the PARSEC 3.0 benchmarks are no longer in active development. When they failed to run on Ubuntu 18.04, we chose to accept this rather than attempting to patch the benchmarks which would result in incomparable results. Figure 4 shows our evaluation results, including the time overhead and the memory overhead. Eighty-three instances are supported by all memory managers. Failed executions are represented as white bars.

Performance Overhead. Considering all successful executions, `FFmalloc` introduces 33.1% performance overhead (geometric mean), compared with 42.9% for MarkUs and -0.18% for FreeGuard. However, if we only consider the 83 instances supported by all tools, the overhead is 21.9% for `FFmalloc`, 43.0% for MarkUs and 1.68% for FreeGuard. `FFmalloc` only introduces relatively high overhead to four out of 15 programs – `dedup`, `netdedup`, `swaptions` and `vips`, where the geometric mean is 157.8%, compared with 584.6% for MarkUs and -

Table 4: CPU overhead of secure allocators on `ChakraCore`. The numbers are overall average score from all benchmarks (with 10 iterations). Red values mean the performance decreased, while green values indicate performance improved.

Benchmark	glibc	`FFmalloc`	MarkUs	FreeGuard
WebTooling	25.53	-0.16%	-0.43%	1.26%
Octane	9706.8	-0.73%	-4.81%	3.22%
Kraken	603.0	0.07%	0.28%	0.03%
SunSpider	21.86	1.88%	3.28%	1.30%
JetStream	97.6	0.20%	-3.07%	1.27%

Table 5: Memory overhead of secure allocators on `ChakraCore`. The numbers are average peak memory use from all benchmarks (with 10 iterations). We show glibc memory usage in kilobytes, and show others as changes over glibc's.

Benchmark	glibc	`FFmalloc`	MarkUs	FreeGuard
WebTooling	454,980	**6.09%**	70.99%	35.35%
Octane	148,220	142.53%	252.55%	**84.28%**
Kraken	63,344	**536.43%**	872.17%	765.30%
SunSpider	103,252	**378.36%**	405.63%	440.97%
JetStream	195,552	162.71%	**74.75%**	241.61%

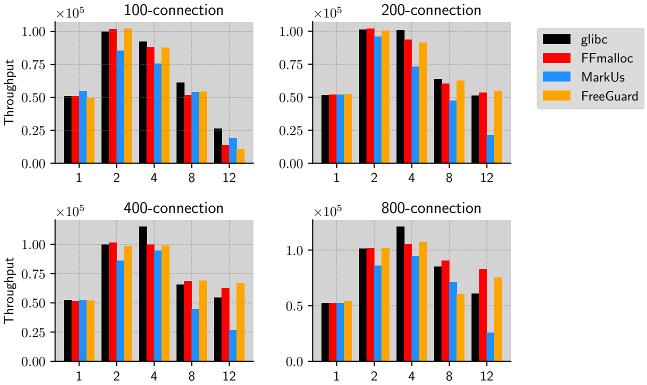

Figure 5: `NGINX` throughput with various allocators. X-axis shows the thread number; Y-axis presents the connection number per second. β-connection means `NGINX` accepts β parallel connections.

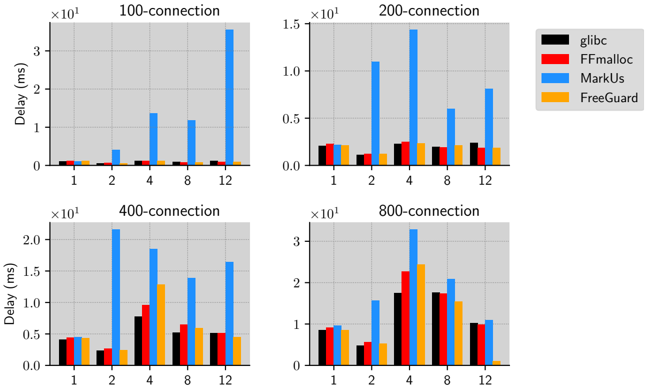

Figure 6: `NGINX` latency with various allocators. X-axis shows the thread number, and y-axis presents the connection number per second. β-connection means `NGINX` accepts β parallel connections.

18.0% for FreeGuard. `FFmalloc` demonstrates merely 4.3% overhead for others.

We observed several interesting facts about the memory overhead. First, `FFmalloc`'s overhead monotonically increased from 5.7% to 50.9% when we used one to 64 cores. This is expected as `FFmalloc` uses locks to prevent race conditions and to synchronize the status of the memory manager. The Linux kernel also has a global lock for `mmap/munmap` system calls, which further increases the overhead for multicore executions. MarkUs followed a similar pattern, but with several exceptions due to unsupported executions. Second, FreeGuard could sometimes improve performance over glibc. For example on `dedup`, the execution with FreeGuard is 3.8× faster when running with 64 cores. This is due to the significantly fewer number of `madvise` system calls compared to that of glibc [31].

Memory Overhead. On the geometric mean, `FFmalloc` introduces 50.5% memory overhead to all successful executions, compared with 13.0% for MarkUs and 141.2% for FreeGuard. For the 83 executions supported by all four allocators, the overheads are 35.6%, 13.3% and 67.5%, respectively. `FFmalloc` brings slightly higher overhead to `bodytrack` (3.3× of original usage) and `swaptions` (10.5×). We find that compared with others, these two programs use relatively little memory ($<$ 50 MB). As `FFmalloc` reserves page pools for different bins, the overhead mainly comes from the allocated-but-not-used memory. However, the overall memory usage is still low, even with `FFmalloc`. MarkUs shows consistent overhead to all benchmarks. FreeGuard introduces the highest memory overhead, and shows extremely high memory usages for `swaptions` (416×), `vips` (8.3×) and `bodytrack` (5.5×). We believe the 50.5% overhead of `FFmalloc` is acceptable to many real-world applications, especially considering its simple design and strong security guarantee against use-after-free bugs.

6.3 Real-world Applications

We also evaluated `FFmalloc` on two real-world large programs: the JavaScript engine `ChakraCore` developed for the Edge browser, and the web server `NGINX`.

6.3.1 ChakraCore

We ran `ChakraCore` on five sets of benchmarks, specifically, Octane, Kraken, SunSpider, Jetstream, and the Web Tooling Benchmark, together with four memory allocators: glibc, `FFmalloc`, MarkUs and FreeGuard. Table 4 and Table 5 show the results of our evaluation, including the performance overhead and memory overhead. In Table 4, as scores in different benchmarks have different meanings, we use green text to show performance improvement, and red text to indicate performance decrease. `FFmalloc` introduces less than 2% run-

Table 6: Comparing munmap with madvise. We inspected the execution of three representative programs and recorded the changes after switching from munmap to madvise. Both instructions and cache misses play an important role in determining the execution time.

Benchmark	Time Diff	# Insn	Cache Miss
xalancbmk	+4.13%	+0.55%	15.54% -> 27.67%
gcc (g23)	+11.90%	+5.70%	27.76% -> 33.07%
mcf	-3.70%	-0.02%	27.86% -> 27.85%

time overhead to four benchmarks, and even improves the performance of JetStream by 0.20%. MarkUs adds the most overhead, 4.81% to Octane and 3.28% to SunSpider while FreeGuard improves the performance for three out of five benchmarks. Regarding memory usage, Table 5 shows that FFmalloc imposes the least overhead among the three secure allocators for three of the programs. MarkUs adds the least overhead on JetStream while FreeGuard was the best on Octane. Overall, FFmalloc, like the other secure allocators, introduces consistently negligible performance overhead to ChakraCore, but typically does so with significantly less memory use.

6.3.2 NGINX

We tested the NGINX webserver through the wrk benchmarking tool [14] with different settings. A setting with α threads and β connections means that wrk uses α threads to send requests to NGINX in parallel, and keeps β connections open at any moment. We ran each setting for 60 seconds and repeated the evaluation using glibc, FFmalloc, MarkUs and FreeGuard. Figure 5 and Figure 6 show the evaluation results. The y-axis of Figure 5 is NGINX throughput in requests-per-second; a higher number indicates better performance. FFmalloc and FreeGuard add negligible overhead in multiple settings, and only show notably higher overhead for the 12-thread, 100-connection setting (47.6% decrease for FFmalloc and 58.8% decrease for FreeGuard). MarkUs has the lowest throughput for most settings. As the number of threads increase, its performance consistently decreases and reaches 65.5% less throughput for the 12-thread, 800-connection setting.

Figure 6 shows NGINX connection latency measured on the client side. Both FFmalloc and FreeGuard introduce minor overhead to the latency. MarkUs introduces significant latency, especially for multi-thread connections.

We also measured the memory overhead for each NGINX thread. On average, FFmalloc consumes 5.24× more memory than glibc, similar to the 5.48× overhead of FreeGuard. However, MarkUs requires 77.72× more memory, which is much higher than FFmalloc and FreeGuard. Overall, FFmalloc introduces negligible overhead to NGINX, and outperforms MarkUs for most of the settings.

6.4 Optimal Settings of FFmalloc

We explored multiple options of releasing memory to find the one enabling the optimal performance and memory usage. First, we configured FFmalloc to release consecutive freed memory with at least α pages (details in §4.2). We tested three different α values, specifically, 32, 8 and 2. In theory, a smaller α means FFmalloc will release memory more frequently, and thus will have higher performance overhead and lower memory overhead. A larger α will have the opposite effect. Second, we configured FFmalloc to use munmap or madvise to return memory to the system. munmap forces the kernel to immediately release the memory, while madvise leaves the kernel to release the memory during high memory pressure. We expected that munmap would have higher performance overhead and lower memory overhead than madvise. Figure 7 shows the performance and memory overhead of FFmalloc on SPEC CPU2006 C/C++ benchmarks, with six different settings.

Munmap vs Madvise. Figure 7 confirms our expectations of the two system calls on memory overhead, but shows a counter-intuitive result on performance overhead. The **Memory** figure shows that FFmalloc with madvise can have significantly higher memory overhead than that of munmap. For example, FFmalloc consumes 198.5× more memory than the original execution if it postpones the memory release via madvise, while the overhead is only 7.6× with munmap. However, the **Time** figure indicates munmap also outperforms madvise on performance, from 0.28% to 0.76%. Although the difference is not significant, considering the high memory overhead, it is clear that we should use munmap instead of madvise to release freed memory to the kernel.

We inspected three programs to understand why madvise sometimes is slower than munmap. The results in Table 6 indicate that both cache misses and extra instructions contribute to the slower execution of madvise. With the madvise system call, the Linux kernel does not immediately reclaim pages due to the low memory pressure in our system. Therefore, future mmap syscalls will likely get a new physical page that is not present in the cache. In contrast, munmap forces the kernel to immediately release the physical page and future mmap calls can reuse the in-cache physical pages, leading to fewer cache misses. For SPEC program gcc, running with madvise executes 5.70% more instructions, causing the most significant overhead on madvise.

Minimum Freed Pages. Figure 7 shows that the minimum consecutive freed pages α is more correlated to memory overhead than to the performance. The performance overhead of FFmalloc is 1.71%, 2.21% and 2.22%, respectively for α values of 32, 8, and 2. Although this is consistent with our intuition that smaller α leads to higher overhead, the difference is not very large. Additionally, not all executions exactly follow this pattern. For example, mcf shows the slowest execution when α is set to 8, not 2. On the other hand, α has a strong impact on the memory overhead when FFmalloc re-

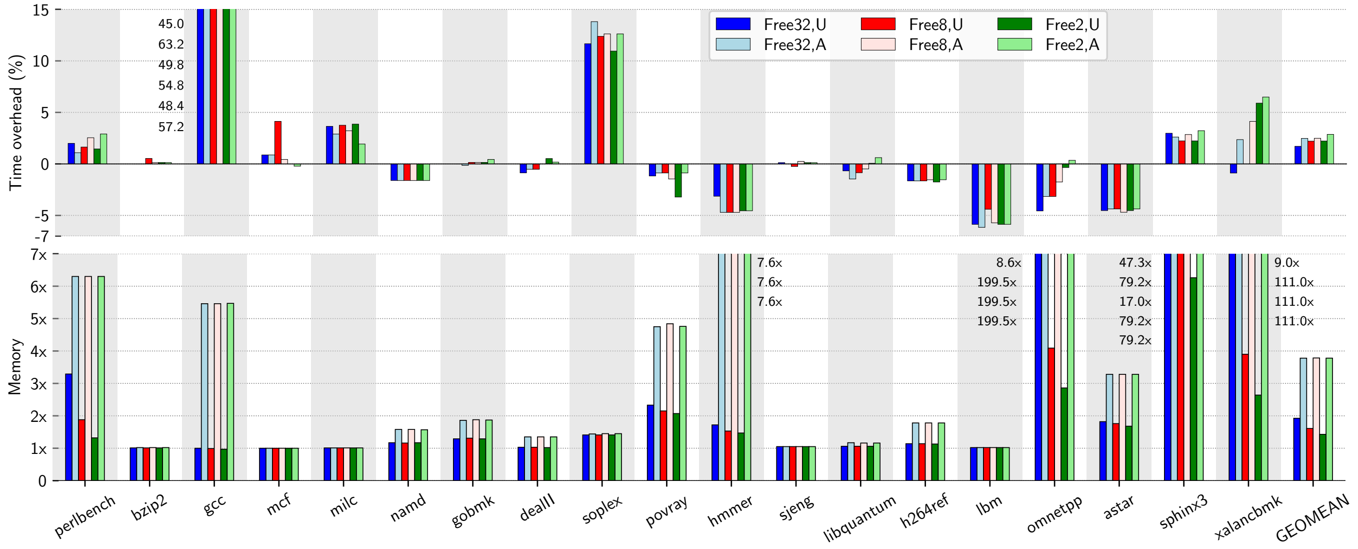

Figure 7: Overhead of FFmalloc with different settings. FreeX means that to release memory, FFmalloc returns at least X consecutive freed pages to the system, through either munmap (U) or madvise (A).

leases memory with munmap. Especially for programs with extremely high overhead, like omnetpp, setting a smaller α can reduce the memory overhead to a reasonable range (from 8.6× to 2.9×). The value of α has no impact on the overhead with madvise, as madvise does not immediately release the memory by design. Therefore, the α value 2 is the best choice among all three values.

During our evaluation, we used the α value 8 and munmap to test all benchmarks and programs. Therefore, FFmalloc's performance overhead can be further reduced if we release memory less aggressively. Alternately, its memory overhead can be reduced by releasing memory more agressively.

7 Discussion

7.1 Other Technical Details

Supporting More Functions. Currently, FFmalloc covers the standard C library functions for memory management including malloc, free, realloc, reallocarray, calloc, posix_align, memalign, aligned_alloc and malloc_usable_size. FFmalloc does not contain wrappers of system calls like mmap and munmap. If an application directly calls mmap and munmap to get memory, a use-after-free bug may escape the protection of FFmalloc. In this case, FFmalloc would be unaware of the address space previously occupied by these mappings and might use them again (only once) for its own allocation. This escape would also affect any other secure allocator, but we have not seen it addressed elsewhere in the literature.

For simple requests to mmap (private, non-file backed, no fixed address), a future version of FFmalloc could handle them via the existing jumbo allocation code path. However, it is less clear what the correct behavior would be for more complex invocations of mmap. For example, how should FFmalloc handle a request that contained the MAP_FIXED flag? If the specified region was not previously used, FFmalloc could allow the call to succeed and then remember to not re-use that range in the future. But, if the desired address range overlapped with a region previously returned by mmap or was previously used by FFmalloc, should FFmalloc fail the call? Blocking the call and returning a failure code could break perfectly legitimate functionality and would negate the ability of FFmalloc to be a drop-in replacement for the glibc allocator. Allowing the call to succeed risks a bypass of its protection which calls into question the value of wrapping mmap at all.

Randomization. Address space layout randomization (ASLR) is widely deployed on modern systems to provide probabilistic protection against various attacks. One can be concerned that FFmalloc may diminish the effectiveness of ASLR due to its sequential allocation scheme. However, such concern does not exist for FFmalloc. ASLR randomizes memory on the *module* granularity, which contains a large number of pages, including code and data. FFmalloc allocates its first pool offset from the randomly assigned default heap. In this way, FFmalloc is fully compatible with ASLR and delivers the security benefits of ASLR to its users. Different from ASLR, the deterministic allocation could be a weakness in case the attack abuses relative heap layout. For example, in case of a heap buffer overflow attack, crafting an exploit would become easier if the adjacent heap chunks affected by the overflow stay at a deterministic relative location. We note that with a small modification to the allocation algorithm, FFmalloc could render the relative heap layout in a non-deterministic way without conflicting our original design goals. We leave the changes to future work.

7.2 Suitability

While FFmalloc is only a prototype of an OTA allocator, its success at running all SPEC and PARSEC benchmarks, unlike many of the other tested systems, demonstrates the feasibility of using it with real workloads. However, the results of tests like gcc and dedup indicate that it may not be appropriate for all applications.

Strengths. Compared to many other systems, FFmalloc provides a hard, rather than probabilistic, guarantee that it can prevent use-after-free exploitation unless the entire application address space is used. This guarantee is useful in remotely accessible applications since an attacker may have repeated opportunities to trigger an exploit. In many cases, this protection comes with one of the lowest CPU overheads relative to alternate systems.

FFmalloc should also be a good choice for use in embedded systems with limited CPU resources, provided they utilize a 64-bit address space. Unlike multiple other systems, FFmalloc does not require an auxiliary thread, typically assumed to be running on a different core than the main application thread, for garbage collection or similar pointer analysis to achieve its performance level.

Weaknesses. Admittedly, the performance of FFmalloc suffers significantly under certain scenarios. FFmalloc batches calls to munmap by waiting until a configurable number of consecutive pages have been freed before returning them to the operating system. Necessarily, this means that applications that free larger allocations will require more munmap calls than those with small allocations. For example, given the default 8-page threshold, a minimum of 2048 16-byte allocations would need to be freed before munmap was called versus only 32 freed 1KB allocations.

Additionally, applications that very frequently allocate and deallocate objects of similar sizes will be slower than with glibc. In this case, glibc can recycle the same few allocation sites repeatedly which reduces cache line misses and avoids system calls for additional address space. In contrast, FFmalloc, will be slowed down by significantly higher number of system calls, even with the batching mechanisms in place.

In addition to the above, FFmalloc can struggle to scale on multi-threaded applications. Even if FFmalloc eliminated all locks from its design, calls to mmap or munmap get serialized within the kernel. Applications that frequently allocate and deallocate objects across threads will end up getting serialized as a result. Proposed revisions to break up the mmap-sem lock in the kernel [6] will likely strongly benefit FFmalloc in this scenario if implemented.

In terms of memory overhead, FFmalloc is weakest when applications allocate objects with different lifetimes simultaneously. When a long lived object is allocated alongside short lived objects it could eventually become the lone allocation preventing unmapping an otherwise unused page or even run of pages. The smaller the object, the higher the impact on memory overhead will be.

Finally, as currently implemented, FFmalloc only provides protection against use-after-free, double-free, and invalid-free vulnerabilities. Protecting against various overflow, overread, or underread type attacks was intentionally omitted to focus on engineering a solution to the OTA problem.

Comparision with Alternatives. FFmalloc compares favorably to other use-after-free focused solutions. By focusing on attack prevention rather than detecting vulnerable pointers, FFmalloc's design is simpler and results in generally higher performance.

Most competitive with FFmalloc is MarkUs. It incorporates garbage collection techniques to verify that there are no dangling pointers to freed allocations. As discussed earlier, in our test environment MarkUs has broadly better memory overhead but somewhat worse performance overhead than FFmalloc, though the gap can be narrowed by having FFmalloc return pages more frequently

In contrast to use-after-free specific solutions, FreeGuard provides tunable probablistic protection against a broad range of attack types. Its strong performance characteristics and breadth of claimed defended attacks on the surface make it an attractive alternative to glibc or single focus secure allocators. However, this probablistic protection could often be of limited value. For example, FreeGuard protects against buffer overreads via use of randomly placed guard pages. In their paper, the authors note that under default settings, FreeGuard only terminated execution of a vulnerable OpenSSL server when attacked by Heartbleed ten percent of the time. Additionally, they claim to mitigate against certain heap overflow attacks by virtue of not placing heap metadata inline as with glibc. This is essentially standard practice by secure allocators in the literature. By this standard, FFmalloc could also claim limited protection against heap overflow. Instead we consider this metadata segregation to be for the security of the *allocator* rather than the heap.

8 Related work

Secure Allocators. Object-per-page allocators such as Archipelago [19] and Electic Fence [27] place each object on an individual page to detect memory safety issues. They can prevent buffer overflows by placing inaccessible pages between objects and can limit use-after-free exploitation by randomizing the reuse of freed pages. However, these approaches are limited to an application that has few live objects or as debugging tools due to their large overhead resulting from the page-granularity of an object.

DieHarder [25] is the security-focused evolution of DieHard [4], which was designed to prevent programs from crashing due to memory corruption bugs. DieHarder simulates an infinite heap where no allocations adjoin each other.

The gaps between allocations resist buffer overflow attacks, and randomized allocation can guarantee address space not being reused probabilistically. FreeGuard [31] provides better performance than DieHarder by adopting techniques from performance-oriented allocators (e.g., free lists per size class). As a result, FreeGuard can achieve similar performance to the default Linux allocator with significant security improvement, but failed to reach a similar security level to DieHarder's. Recently, Guarder [32] is proposed to bridge this gap as an evolved version of FreeGuard. It substantially increases its randomization entropy, but has similar performance overhead compared to FreeGuard by introducing new techniques to manage small objects and adjusting tradeoffs between performance and security. Unlike these approaches that probabilistically prevent use-after-free bugs, OTA can completely stop them by guaranteeing one-time allocation. However, OTA is more modest in only attempting to prevent use-after-free bugs and its variations such as invalid free and double free.

Cling [3] restricts memory reuse within objects of the same type rather than completely disabling it. It argues that this design severely limits an attacker on exploiting use-after-free vulnerabilities while retaining efficient memory usage. However, this does not completely block use-after-free exploitation. Rather, it requires the attacker to control a new matching type object rather than any suitably sized one. Similar to OTA, Oscar [7] also prevents use-after-free by employing a forward only allocation principal. It simulates the object-per-page style allocator using shadow memory to overcome the high memory overhead of placing each object on discrete pages. Despite its improvement on earlier work [8], it still imposes significant overhead in the form of expensive memory mapping system calls compared to OTA.

Pointer Invalidation. An alternative approach to preventing use-after-free attacks is to invalidate dangling pointers when the object is freed. DangNull [15] keeps track of all pointers to all allocated objects, and explicitly nullifies pointers once the pointed-to object is freed. FreeSentry [37] takes a similar approach as DangNull, except that it flips the top-bit of the pointer to make it an invalid address. This helps preserve context when reviewing crash dumps. DangSan [35] improves the performance of this technique on multi-thread applications, with the help of an append-only per-thread log. pSweeper [18] avoids live tracking of pointers by concurrently scanning memory to find dangling pointers. Instead of proactively destroying dangling pointers, CRCount [30] waits for the program to reset all such pointers; it frees an object only if the reference counter for the pointer becomes zero. MarkUs [2] is similar to CRCount, except that it starts scanning from the stack, global memory and registers. These schemes usually impose significant CPU and memory overhead due to the difficulty of tracking pointers in C code.

Use-After-Free Detection. CETS [23] inserts additional metadata at the program runtime, a lock for each object and a key for each pointer. During the object creation, it initializes the lock for the object and assigns the corresponding key to the pointer. During the program execution, the key is propagated together with the pointer and the lock is reset when its corresponding object is freed. Thus, any memory access with a dangling pointer will be detected and blocked by checking its key. Since CETS needs to maintain a key for each pointer and to compare key and lock for each memory access, it introduces substantial overhead.

Undangle [5] utilizes dynamic taint analysis to track the propagation of pointers, and detects the use-after-free bug if the source of the pointer has been freed. Due to the heavy runtime overhead of taint analysis, Undangle is impractical for real-world deployment.

Valgrind [24] and AddressSanitizer [28] can detect memory errors including use-after-free by checking the validity of memory accesses. Since they are designed for debugging, not for security, an advanced attacker can easily bypass their mechanisms. For example, researchers already have shown that use-after-free is still exploitable under AddressSanitizer by exhausting its fixed-size quarantine for freed memory [15].

Project Snowflake [26] adds manual memory management to a garbage-collected runtime. It introduces the notion of a *shield* which tracks references to unmanaged memory and can only be created by the reference owner. Even after the owning reference is deleted, the memory will not be reused until all shields have been destroyed as well.

9 Conclusion

We designed and implemented a memory allocator based on the one-time allocation (OTA) principal, aiming to prevent exploitation of use-after-free bugs. OTA provides a distinct memory chunk for each memory request, where attackers cannot reclaim the freed memory and thus cannot overwrite the content for exploitation. We explored several design choices and found the optimal ones to reduce the overhead of our prototype. The evaluation shows that OTA can prevent real-world use-after-free exploits effectively and efficiently.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Andrea Lanzi, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

References

[1] Muhammad Abid. Raytrace running infinitely. https://lists.cs.princeton.edu/pipermail/parsec-users/2010-January/000620.html.

[2] Sam Ainsworth and Timothy Jones. MarkUs: Drop-in Use-after-free Prevention for Low-level Languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[3] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, August 2010.

[4] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[5] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Minneapolis, MN, July 2012.

[6] Jonathan Corbet. How to get rid of mmap_sem. https://lwn.net/Articles/787629/.

[7] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[8] Dinakar Dhurjati and Vikram Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

[9] Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-spraying Attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[10] Jason Evans. jemalloc: Memory Allocator. http://jemalloc.net/.

[11] Antonio Franques. Can anyone provide detailed steps to fix Host x264? https://github.com/cirosantilli/parsec-benchmark/issues/3.

[12] Sanjay Ghemawat. TCMalloc : Thread-Caching Malloc. https://gperftools.github.io/gperftools/tcmalloc.html.

[13] Wolfram Gloger. Wolfram Gloger's Malloc Homepage. http://www.malloc.de/en/.

[14] Will Glozer. wrk - A HTTP Benchmarking Tool. https://github.com/wg/wrk, 2019.

[15] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[16] JungWon Lim. freeing list operation leads to exploitable write-after-free. https://github.com/SamAinsworth/Markus-sp2020/issues/2.

[17] JungWon Lim. large chunk (with dangling pointer) is forgotten and can be reclaimed. https://github.com/SamAinsworth/Markus-sp2020/issues/1.

[18] Daiping Liu, Mingwei Zhang, and Haining Wang. A Robust and Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[19] V Lvin, G. Novark, E. Berger, and B Zorn. Archipelago: Trading Address Space for Reliability and Security. In *ACM SIGPLAN Notices*, volume 43, 2008.

[20] Matt Miller. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape.pdf. https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf, 2019. BlueHat IL.

[21] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[22] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[23] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[24] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[25] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, October 2010.

[26] Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. Project snowflake: Non-blocking safe manual memory management in .net. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[27] Bruce Perens. Electric Fence. https://linux.softpedia.com/get/Programming/Debuggers/Electric-Fence-3305.shtml.

[28] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[29] Shellphish. how2heap: A repository for learning various heap exploitation techniques. https://github.com/shellphish/how2heap.

[30] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[31] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[32] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A Tunable Secure Allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[33] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Softw. Pract. Exper.*, 43(1), January 2013.

[34] Alexander Sotirov. Heap Feng Shui in JavaScript. https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf, 2007. BlackHat Europe.

[35] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[36] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[37] Yves Younan. FreeSentry: Protecting against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[38] Insu Yun. Another unexpected behavior in markus allocator. `https://github.com/SamAinsworth/Markus-sp2020/issues/3`.

[39] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *Proceedings of the 29th USENIX Security Symposium (Security)*, August 2020.

A Statistics of SPEC CPU2006 Benchmarks

Table 7: Statistics of SPEC CPU2006 benchmarks. **Function Count** lists the number of calls to memory management functions, including `malloc`, `free`, `realloc` and `calloc`. Note that some invocations of `malloc` come from `realloc` and `calloc`. **Memory Usage** column shows the memory requested by and allocated to the program. `total_req` means the sum of all requested memories, without considering any `free` operation. `total_alloc` is similar to `total_req`, but each allocation size is aligned to 16 bytes. `max_alloc` shows the largest memory usage along the execution, if every allocation is aligned to 16 bytes. **Time** provides the time for each program complete the execution. **Alloc Freq** shows the average memory request per second.

Program	Function Count				Memory Usage			Time	Alloc Freq
	#malloc	#realloc	#calloc	#free	total_req	total_alloc	max_alloc	(s)	(MB/s)
400.perlbench	353M	12M	3	347M	15.97 G	18.28 G	1.08 G	551	33.2
401.bzip2	174	0	0	144	3.64 G	3.64 G	3.53 G	758	4.8
403.gcc	28M	45K	4726K	28M	741.09 G	741.32 G	4.18 G	516	1436.7
429.mcf	8	0	3	7	1.76 G	1.76 G	1.76 G	460	3.8
433.milc	6521	8	6513	6466	88.32 G	88.32 G	719.37 M	934	94.6
444.namd	1328	0	0	1323	47.15 M	47.15 M	47.13 M	612	0.1
445.gobmk	607K	52K	317K	607K	1.07 G	1.07 G	130.49 M	689	1.6
447.dealII	151M	0	1	151M	11.38 G	12.47 G	827.44 M	576	21.6
450.soplex	247K	75K	4	236K	49.31 G	49.31 G	849.80 M	419	117.7
453.povray	2443K	46K	0	2416K	82.92 M	95.24 M	3.60 M	341	0.3
456.hmmer	2419K	369K	123K	2107K	2.52 G	2.70 G	36.62 M	638	4.2
458.sjeng	6	0	0	2	180.00 M	180.01 M	180.01 M	809	0.2
462.libquantum	142	58	121	121	1.04 G	1.04 G	100.67 M	1641	0.6
464.h264ref	178K	0	171K	178K	1.40 G	1.40 G	112.16 M	903	1.6
470.lbm	7	0	0	6	428.81 M	428.82 M	428.81 M	684	0.6
471.omnetpp	267M	0	8	267M	44.65 G	46.76 G	154.68 M	579	80.8
473.astar	4802K	0	6	4802K	4.39 G	4.40 G	451.55 M	610	7.2
482.sphinx3	14M	0	14M	14M	16.15 G	16.23 G	43.04 M	807	20.1
483.xalancbmk	135M	0	8	135K	6.28 G	66.66 G	383.47 M	339	196.6

B Proof-of-Concept of MarkUs Exploit

```
class Victim {
  public:
    virtual void good() { puts("Hello World"); };
};
void evil() {
  puts("[!] Spawning shell...");
  execve("/bin/sh", NULL, NULL);
}
int main() {
  Victim *a = new Victim();
  Victim *b = new Victim();
+ printf("[+] a = %p, a.vftable = 0x%lx\n",
+                a, ((uintptr_t*)a)[0]);
  free(a);
  free(b);
+ printf("[+] a = %p, a.vftable(corrupted) = 0x%lx\n",
+                a, ((uintptr_t*)a)[0]);

+ const size_t kSpraySize = 0x300000000;
+ uint8_t *spray = (uint8_t*)malloc(kSpraySize);
+ uintptr_t offset = ((uintptr_t*)a)[0] & 0xfff;
+ assert(offset == 0x3de); // offset is always constant
+ for (size_t i = 0; i < kSpraySize; i += 0x1000)
+   ((uintptr_t*)(spray + offset))[i / 8] = (uintptr_t)evil;
+ printf("[+] Spray at %p - %p\n", spray, spray + kSpraySize);

+ // Make sure that 'a' is not reclaimed
+ assert((void*)a < spray || (void*)a >= spray + kSpraySize);
+ puts("[+] Triggering UAF (virtual function call)!");
  a->good();
}
```

Figure 8: A use-after-free bug and its exploitation for MarkUs. The code snippets after the plus sign (+) represents exploitation and shows internal information to make PoC more clear.

```
$ lsb_release -d
Description:    Ubuntu 18.04 LTS
$ g++ -o poc poc.cpp
$ LD_PRELOAD=$MARKUS ./poc
[+] a = 0x55c80d35eff0, a.vftable = 0x55c80ce3bd70
[+] a = 0x55c80d35eff0, a.vftable(corrupted) = 0x55cad2e9f3de
[+] Spray at 0x55c80d39a000 - 0x55cb0d39a000
[+] Triggering UAF (virtual function call)!
[!] Spawning shell...
$
```

Figure 9: Results after executing PoC code in Figure 8 in Ubuntu 18.04. This demonstrates arbitrary code execution by invoking `evil()`. Unlike an ordinary allocator that requires to reclaim the freed object, this PoC program uses heap spray to control the virtual function table, which is corrupted by MarkUs's quarantine management. Note that `$MARKUS` is the environment variable to make the PoC program to use MarkUs as its underlying allocator.