S2malloc: Statistically Secure Allocator for Use-After-Free Protection And More

Ruizhe Wang, Meng Xu, and N. Asokan

University of Waterloo
{ruizhe.wang, meng.xu.cs}@uwaterloo.ca asokan@acm.org

Abstract. Attacks on heap memory are ever-increasing. Existing entropy-based secure memory allocators can provide statistical defenses against various heap-based exploits, including use-after-free (UAF). However, although UAF mitigation is in scope, such allocators are not tailored to *detect* failed UAF attempts. Consequently, an attacker can beat entropy-based protection by repeating the same attack, possibly in combination with heap spraying, to improve their chance of success further.
We introduce S2MALLOC, aiming to enhance UAF-attempt detection without compromising other security guarantees or introducing significant overhead. S2MALLOC consists of three new constructs in the secure allocator design space: **free block canaries (FBC)** to detect UAF attempts, **random in-block offset (RIO)** to stop the attacker from accurately overwriting the victim object, and **random bag layout (RBL)** to impede attackers from estimating the block size based on its address. We show that S2MALLOC can detect UAF attempts with a reasonable probability and is practical, with only a 2.8% CPU overhead on PARSEC and an 11.5% CPU overhead on SPEC.

Keywords: Secure Memory Allocator · Use-After-Free.

1 Introduction

Heap-related vulnerabilities are severe and common threats that can be leveraged to launch attacks resulting in arbitrary code execution or information leakage. Heap overflow, double and invalid free, and use-after-free (UAF) are among the most common types of these vulnerabilities. According to the Common Vulnerabilities and Exposures (CVEs) report of 2022, they were ranked 19th, 11th, and 7th in terms of bug prevalence [25].

Secure memory allocators play a crucial role in defending against heap exploitations. They can be broadly categorized into two types: entropy-based generic allocators, which use randomness to enhance security, and UAF-mitigating allocators, which specifically target UAF vulnerabilities.

UAF occurs when a previously freed memory block is used to store data. It receives special attention in secure allocator design due to its prevalence and the powerful exploitation primitives (e.g., arbitrary read/write) it enables. Chromium has reported that more than a third of its security issues are related to UAF, which is more prevalent than other types of memory errors combined [36]. Prior

works have demonstrated that effective UAF mitigation is not only possible but can be achieved in several ways, such as tracking and nullifying pointers on de-allocation (e.g., MarkUs [2]), forward-only allocation (e.g., FFmalloc [41]), and context-based allocation (e.g., Cling [3]). However, these approaches also have drawbacks, such as substantial CPU overheads (e.g., about 40% for MarkUs), downgraded protection in edge cases (e.g., same-context UAF is still possible with Cling), or new attack vectors exposed due to complexity in implementation (e.g., FFmalloc) albeit preventable.

Entropy-based heap allocation is another way to mitigate UAF exploits. Allocators in this theme share one limitation: probabilistic protection. However, the relaxed security requirement also enables them to protect most, if not all, common heap vulnerabilities with simpler and binary-compatible designs. Specifically on UAF mitigation, entropy-based memory allocators typically use delayed free-lists [30,20] to prevent the same memory block from being *immediately* reallocated after being freed. Attackers now face a moving target even when they obtain a dangling pointer, as they have less confidence in knowing when this pointer becomes valid again and/or which object it might point to.

While achieving relatively low CPU overhead, especially compared with UAF-mitigating allocators (details in §2), existing entropy-based allocators still face the challenge of entropy loss, further compromising the probabilistic protection. To illustrate how entropy loss can occur: 1) an attacker could control multiple dangling pointers via heap spraying, increasing the chance of corrupting a target object through one of the dangling pointers; 2) an attacker could infer block size or type based on pointer addresses; 3) an attacker could retry the same attack repetitively if a failed attempt only causes silent failure (more details in §3).

To overcome these challenges, we propose S2MALLOC, an allocator that reduces entropy loss by detecting UAF *attempts* with low CPU and memory overhead on par with state-of-the-art entropy-based memory allocators. S2MALLOC achieves its promises by combining several new realizations of existing concepts: **randomized in-slot offset (RIO)**, **free-block canary (FBC)**, and **random bag layout (RBL)**. RIO mitigates UAF attacks by allocating blocks with random offsets, obstructing the attacker from locating the target field in a data structure. FBC puts cryptographically secure canaries in free blocks to detect illegal writes, turning a failed UAF exploitation attempt into a clear signal. RBL organizes blocks of the same size range using sub-bags. Only blocks within the same memory page are guaranteed to be in the same sub-bag.

Summary. We claim the following contributions:

- We analyze current entropy-based allocators in real-world UAF attack scenarios and show their protection is weaker than claimed (§3).
- We present S2MALLOC, a drop-in solution that addresses the above weaknesses while protecting against other commonly observed heap memory vulnerabilities. S2MALLOC does not require special hardware, program recompilation, or elevated privileges and works on x86 and AARCH (§4).
- Through various real-world CVEs and benchmarks, we show that S2MALLOC can successfully detect all attacks (§5) while incurring 2.8% CPU overhead and

27% memory overhead on the PARSEC benchmark and 11.5% CPU overhead and 37% memory overhead on the SPEC benchmark (§6).

Both the software artifact and an extended version of the paper are available. In the extended version, we 1) discuss the implementation details more thoroughly, 2) document finer-grained evaluation results, and 3) provide a more formal analysis on the security of S2MALLOC.

2 Background and Related Work

2.1 Entropy-based allocators

Entropy-based heap allocators strive to protect against almost all heap-based exploits mentioned above by minimizing the attacker's success rate. Ideally, the success rate should be low enough to deter attackers from trying to attack the system. However, as demonstrated in Guarder [30], practical implementations face limitations in memory and computation resources.

BIBOP. State-of-the-art entropy-based allocators typically build on the *Big Bag of Pages* (BIBOP) [16] mechanism with various security enhancements. BIBOP-style allocators allocate blocks of the same size class together using one or more continuous memory pages and pre-emptively allocate sub-pools for each size class. Each sub-pool is divided into several slots pre-emptively, and each allocated object takes one slot. A slot is not further divided or merged so that memory fragmentation can occur only within a slot. For a long-running program, blocks from newly mapped pages continue to be added to the pools until the program reaches a steady state, after which most allocations and deallocations involve existing blocks only. No performance degradation will happen as the program executes. A bitmap monitors the status of each slot and can be used to defend against double or invalid frees. Bag metadata is stored separately to avoid metadata-based attacks [29], and UAF only occurs within the same size class.

Extended secure features. Prior works[27,30,20] have introduced a diverse set of security enhancements over the basic BIBOP-style allocation, including:

- *Guard page*: A guard page is a single-page virtual memory block not mapped to physical memory. Therefore, any attempt to dereference an address in a guard page triggers a segmentation fault. Guard pages can be placed after each bag or randomly within bags to protect against out-of-bound accesses.
- *Heap canary*: This is a small object set to a secure value and placed at the end of each slot. Heap overflow can be detected if the canary value is modified.
- *Random allocation*: This guarantees that slots within each bag are not allocated sequentially (i.e., linear allocations). Instead, each allocated slot is randomly chosen from at least r free slots (where r is the entropy).

2.2 UAF-mitigating allocators

UAF-mitigating allocators specialize in UAF-protection only and can be broadly categorized into the following types:

A *Pointer validation on dereference* [39,11,42]: i.e., checks whether a pointer is valid and safe for a read/write operation upon dereference.
B *Pointer invalidation on free* [13,8,28,12,41]: i.e., whenever `free` is called on a pointer, it revokes its capability to access the associated virtual address.
C *Memory sweeping* [18,2,10]: i.e., checks all pointers stored in memory and either actively removes all dangling pointers or reuses a freed memory chunk with the assurance that no dangling pointer to this freed chunk exists.
D *Context-based allocation* [3,38,9]: i.e., permits the re-allocation of freed memory chunks only to objects allocated in the same "context."

Not all UAF-mitigating allocators are drop-in replacements of the system memory allocator, as some of them require recompilation of the protected programs [3,38,40], special hardware [12], or kernel modifications [13].

Security guarantee. UAF-mitigating allocators in categories A, B, and C can eliminate all UAF exploits (hence complete mitigation), although some of them might incur large overheads or require customized kernel or hardware. Context-based allocators (category D), first proposed in [9] as a heap-safety property, typically run faster but offer incomplete protection. For example, Cling [3] defines the allocation context based on the two innermost return addresses on the call stack when `malloc` is invoked. Suppose two memory allocations occur in the same context; in this case, the object allocated the second time may occupy the same memory chunk allocated the first time (if the first object is freed). TypeAfterType [38] defines the allocation context based on object types (e.g., the type `int` passed in `malloc(sizeof(int))`, and objects may be reallocated on freed memory chunks originating from the same allocation context. In summary, UAF is still possible in context-based allocators.

However, unlike entropy-based allocators (including S2MALLOC), which also mitigate UAF imperfectly, the protection from context-based allocators is deterministic. Specifically, if the dangling pointer and the target object are allocated in different contexts, there is zero chance of success in corrupting the target object, regardless of the attackers' strategies. On the other hand, context-based allocators provide no protection if the dangling pointer and the target object fall in the same context. Entropy-based allocators provide probabilistic protection in both cases. We will discuss the implications via CVEs in §5.3.

Porting for UAF-write defense only. Allocators that instrument runtime checks for pointer validity (i.e., category A) can opt to trade protections of UAF-read attacks for performance by checking the pointer validity on write accesses only. In fact, as reported in [31], only 5% to 25% of memory accesses are write accesses in the SPEC 2017 benchmark, indicating a potential reduction in overhead for category-A allocators. On the other hand, for allocators in categories B, C, and D, splitting UAF-read and UAF-write protection is inherently hard as their designs do not differentiate between read and write accesses.

2.3 Widely deployed secure allocators

Secure allocators have been adopted in industrial codebases as well. Scudo [22] is a solution from LLVM [21] that balances security and performance and supports

random allocation and delayed free-list (the latter is disabled by default), a subset of security features provided by entropy-based allocators [30,20]. Hardened_malloc [15] is the default allocator of GrapheneOS [14], a privacy and security-focused Android-based OS. Hardened_malloc incorporates all security features discussed in §2.1 and focuses on UAF-write protections. Similar to other entropy-based allocators, Hardened_malloc suffers from reduced randomness. It also incurs increased overheads for larger blocks, which limits its use scenarios.

3 Motivation

While effective in defending against various heap exploits, entropy-based allocators are not ideally suited to protect against UAF attacks. Specifically, easy-to-identify blind spots exist that drastically reduce the efficacy of defending UAF using random allocation and increase attackers' confidence in launching a successful attack without trying to predict the next allocation slot.

3.1 Adversary model

We assume that the attacker can analyze the source code and binary executable to determine the implementation details of the target program, including vulnerabilities and other relevant information, such as the size and layout of critical data structures. We also assume that the attacker can identify when a victim object is allocated or de-allocated and is aware that the target runs with S2MALLOC.

We assume that the underlying OS kernel and hardware are trusted, and an attacker cannot utilize a data leakage channel, such as `/proc/$pid/maps`, to discover the location of the heap allocator's metadata. The attacker cannot compromise the random number generator, nor can they take control of the heap allocator. Exploiting bugs of the allocator itself is out of scope. These assumptions are similar to that of other entropy-based allocators [30,20].

We allow attackers to use any existing heap feng-shui [32] technique (e.g., heap spray) to prepare or manipulate heap layout to facilitate UAF exploits. Attackers can retry an exploit if previous attempts fail silently. These assumptions make our adversary model more robust than those assumed in entropy-based allocators and on par with the adversary models in UAF-mitigating allocators [2,41].

3.2 Challenge 1: entropy loss

Entropy-based allocators thwart UAF by avoiding instant memory reuse, as that makes where a new object is allocated more predictable. However, if 1) the attacker can continue to retry the attack when the previous trial fails, or 2) heap memory can be spoofed with either dangling pointers or victim objects, it is guaranteed that the attack would eventually succeed without the victim's notice.

Figure 1 is an example, abstracted from mRuby issue 4001 [34], a UAF vulnerability in the Ruby compiler. The function `mrb_io_initialize_copy` is called when opening a file. It first frees the existing data pointer of the `copy`

```
1  mrb_io_initialize_copy(mrb_state *mrb, mrb_value copy) {
2    mrb_value orig;
3    struct mrb_io *fptr_copy, *fptr_orig;
4    fptr_copy = (struct mrb_io *)DATA_PTR(copy);
5    if (fptr_copy != NULL) {
6      mrb_free(mrb, fptr_copy);
7    }
8    fptr_copy = (struct mrb_io *)mrb_io_alloc(mrb);
9    fptr_orig = io_get_open_fptr(mrb, orig);
10   DATA_PTR(copy) = fptr_copy;
11 }
```

Fig. 1: Example UAF attack based on mRuby issue 4001 [34].

object (`DATA_PTR(copy)`) (line 7) and allocates a new object to it (line 9 and 11). If an invalid argument is given, calling `io_get_open_fptr` will throw an exception (line 10), causing an early return, which makes `DATA_PTR(copy)` a dangling pointer. The attacker can then close the file in Ruby, which will set the first word of the pointed memory to `INF`. If a string object takes this memory, its length will be overwritten to `INF`, enabling arbitrary memory reads and writes.

In the above scenario, random (i.e., non-sequential) allocation or delayed free-lists available in existing entropy-based allocators [27,30,20] merely increase the attack difficulty: as long as the attacker can wait, the memory chunk referred to by the dangling pointer will eventually be reallocated, allowing the UAF exploit to proceed after some delay. Furthermore, the entropy diminishes if an attacker can repeat the same attack *without penalty* (e.g., when a failed attempt does not crash the target program or trigger attention). Similarly, if attackers can spray the heap with either dangling pointers of victim objects, the probability of success increases significantly.

This UAF exploit strategy motivates us to design an allocator that 1) actively searches for UAF attempts and raises signals if evidence is found and 2) stops the attacker from locating critical information in memory even if the attacker manages to obtain a dangling pointer. In this example, we can stop the attacker from locating the string length deterministically even if the attacker manages to obtain a dangling pointer for a string object initially pointed to by `DATA_PTR(copy)`. In addition, any attempts to write to unallocated memory will be detected with a high probability. If an attacker attempts to spray the heap with arbitrary write to increase success rates, we can raise a signal on or even before the UAF happens.

3.3 Challenge 2: information leak

Existing entropy-based allocators [30,20] create a memory pool for each block size range, resulting in the leakage of block size via their address, possibly revealing the victim program's internal state to the attacker. For example, each BIBOP bag is assigned an 8GB virtual memory pool in SlimGuard [20]. Any objects belonging to this bag will be allocated from this pool. As a result, obtaining a known-sized block will be sufficient to infer the size of any blocks sharing the upper 32-bit address. Further, block addresses in Guarder [30]are size-aligned, allowing the attacker to infer the block size based on its address. For example, a block with an address value ending with `0x10` is of size 1 to 16 bytes, and a block with an address value ending with `0x300` is highly likely of size 129 to 256 bytes.

We mitigate this threat by dividing bags into sub-bags, limiting the size leakage only if the attacker-controlled block resides in the same sub-bag as the victim block. Furthermore, we assign random guard pages within sub-bags to make their boundaries unpredictable.

4 Design and Implementation

4.1 Architectural overview

At its core, S2MALLOC adopts BIBOP to manage memory blocks. Figure 2 shows an overview of S2MALLOC. S2MALLOC maintains per-thread metadata (Ⓐ) stored in a memory chunk requested directly from the kernel. Huge blocks are obtained or released from the OS directly (Ⓑ) and stored using a linked list. Small objects are maintained using bags, claiming memory indirectly from a segregated memory pool (Ⓒ). Each **regular bag** maintains the metadata of blocks of a size range, including the number of free slots and a list of **sub-bags**. We take **sub-bag** as the basic unit of a group of slots (§4.3).

The data field points to a memory chunk requested from the memory pool to store the objects allocated to this sub-bag. The bitmap indicates whether the current slot is taken or not. If the current slot is taken, the offset table cell stores an offset indicating where the data stored in the bag starts (§4.2). Otherwise, the slot is free, and the offset table stores the location of the FBC (§4.5).

When a `free()` call is received by S2MALLOC (step ① → ②), S2MALLOC checks whether: (1) the bitmap indicates the current block is taken; (2) the offset stored in the offset table matches with the freed pointer address; and (3) the canary value is not modified. If all checks pass, the block will be freed: the bitmap cell will be set to free, and an FBC will be put at a random location in the current block. The offset table will be updated to store the location of the FBC.

When a `malloc()` call is received (step ② → ③), S2MALLOC randomly selects one free block of the corresponding size and checks the FBC of current and nearby free blocks. A random offset will be generated, indicating where the data starts within the block, and the offset table will be updated accordingly. The heap canary will be set after the last data byte of the current block, and the bitmap will be updated. This example assumes that each block stores at most 7 bytes of data. The heap canary is initially set to the 9th byte as the offset is one and then set to the 10th after reallocation as the offset is changed to two.

4.2 Randomized in-slot offset (RIO)

In S2MALLOC, an object is stored with a random offset p, and the first p bytes of the assigned slot will be left empty. After the slot is freed and allocated to another object, the offset p will be re-computed. Thus, the relative offset between these two objects cannot be accurately predicted, and attackers cannot reuse a dangling pointer to read or write the newly allocated object with predictable alignment. This design aligns with the BIBOP concept and does not cause fragmentation within large memory chunks.

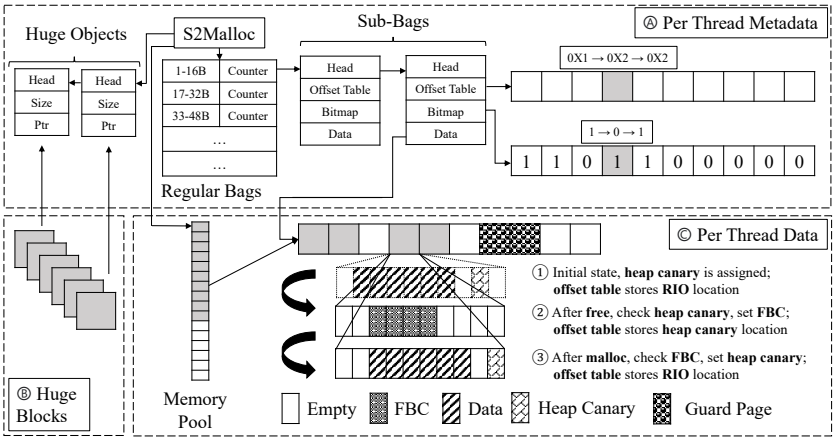

Fig. 2: Overview of S2MALLOC with an example of free and malloc. Ⓐ, Ⓑ, and Ⓒ show three S2MALLOC segments, stored in segregated memory. ①, ②, and ③ show how an allocated bag slot is freed and then allocated.

We define k to be the RIO entropy. For each bag with blocks of b bytes, $e = b/k$ bytes are not used to take data, and each block can take at most $b - e$ bytes of data. We refer to these extra bytes as entropy bytes.

Suppose this block is `malloc`-ed with an object of s bytes ($s < b - e$). Before this block is allocated, $p \in (0, b - s)$ is computed to decide the starting byte of the data object. p is 16-byte aligned following the minimum default alignment of GNU C implementation [37]. It is compatible with special data structures, such as atomic structs which must be aligned with registers and cache lines. The offset of each block will be stored separately in an offset table.

4.3 Random bag layout (RBL)

As with existing secure allocators, S2MALLOC employs BIBOP to manage small-size blocks. Blocks larger than 64KB are mapped and unmapped directly from the OS and are managed using a linked list. Smaller blocks are further classified as small, medium, and large blocks to decrease the number of size classes. Small bags contain blocks smaller or equal to 1KB, and a bag is created every 16 bytes (16 bytes granularity). Medium bags contain blocks within the range of 1KB and 8KB with a granularity of 512 bytes; large bags contain blocks within the range of 8KB and 64KB with a granularity of 4KB.

S2MALLOC does not link block sizes to block addresses. Instead of allocating a dedicated virtual memory pool for each bag (as shown in prior works [30,20]), we create a single virtual memory address pool for all bags. We further divide each bag into sub-bags, each containing 256 slots. Each bag creates new sub-bags upon need, and a newly created sub-bag will request corresponding memory from the pool. We use a bump pointer to track the available memory in the pool and sequentially allocate pool memory to sub-bags.

Secure Allocators	Guard Pages	Rand. Alloc.	Segre. MD	Heap Canary	Ptr Inval.	UAF Mitigate	UAF Detect	Overheads Memory	Runtime
DieHarder	Y	Y	Y	N	N	◐	N	21.3%	2.1%
Guarde	Y	Y	Y	Y	N	◐	N	58.1%	2.4%
SlimGuard	Y	N	N	Y	N	◐	N	22.5%	4.4%
S2MALLOC	Y	Y	Y	Y	N	●	High prob.	26.8%	2.8%
MarkUs	N	N	N	N	Y	●	N	13.0%*	42.9%*
FFmalloc	N	N	N	N	Y	●	Y	50.5%*	33.1%*

Table 1: Overview of existing secure memory allocators and S2MALLOC to illustrate how S2MALLOC fills the gap (MD: metadata, ◐: One-shot attack only, ●: Repeated/spray attacks). Overheads are measured on PARSEC [5], detailed in §6. Overheads of MarkUs and FFmalloc (numbers marked with $*$) are reported in [41] instead of measured by us.

S2MALLOC randomly inserts guard pages within sub-bags to thwart overflow, spraying, and random pointer access. This makes memory pool allocation even less predictable. S2MALLOC does not leak the block size unless both a known and a victim block reside on the same memory page. Adjacent blocks may not be within the same sub-bag, and RIO guarantees block start addresses cannot be deterministically predicted. On the contrary, two blocks will likely be in the same size range in SlimGuard if their address difference is smaller than 8GB.

4.4 Hardening heap canaries

A canary is a small data block placed after the allocated memory to detect overflow. It was initially introduced in StackGuard [6] to protect the stack and has been adopted to protect the heap [26]. The canary is set to a specific value when a memory slot is allocated. This value is checked when the slot is freed, and memory overflow is detected if the canary value changes.

However, in previous entropy-based allocators, this value is set to be either globally identical [30] or bound with slots [20] so that a knowledgeable attacker can trivially break. We follow the previous secure canary designs to use the secure MAC of the memory address as the canary [19,24]. Specifically, we take the CMAC-AES-128 encrypted block address as the canary implemented using AES-NI [17] (on x86) or Neon [4] (on AARCH) to keep the canary confidential and compact. Even if the attacker learns a canary value, they can only use it to break the current object or any further objects allocated to this slot with the same RIO. In S2MALLOC, we put a ι-byte canary immediately after the last data-storage byte in the allocated slot (i.e., the $(p+b-e)$th byte), and the canary will be checked upon `free`.

4.5 Free block canaries (FBC)

Existing entropy-based allocators defend against UAF-write attacks by statistically avoiding allocating a victim object in a block pointed to by a dangling

pointer. Although a failed attack attempt only modifies a free block without causing any harm, it is not detectable either, and given that the same attack can be retried, the attacker will eventually succeed.

To detect such attempts, we put a canary of length c, also computed using CMAC-AES-128, in each free block. Its value is checked before the block is allocated and will be reset after it is freed. We also check the FBC of d nearby blocks to improve the detection rate. FBC guarantees that accessing a freed block is not risk-free. We detail its protection rate in §5. In Figure 2, we further illustrate how the two kinds of canaries (FBC and regular heap canary) are set and cleared when an allocated block is freed and then allocated again.

Initially, in S2MALLOC, we create the memory pool using the `mmap` system call with the `ANON` flag, and all allocated memories are initialized to zero in the Linux environment. We take this advantage and use the zeros as the initial FBC to avoid additional CPU and memory usage for unallocated blocks. We also use the whole slot as the canary with improved security. As encrypted canaries may be more costly than zeroing out memory, S2MALLOC chooses to zero out small blocks and checks the whole block before it is allocated to a new object, bringing both security and computation benefits.

4.6 Summary and comparison

Table 1 summarizes S2MALLOC and state-of-the-art secure allocators along the two defense lines closely related to S2MALLOC (as discussed in §2.1 and §2.2).

S2MALLOC is an entropy-based allocator. It has nearly identical heap exploitation protection features *except* for UAF protection compared with existing works [27,30,20]. S2MALLOC provides a much more robust security assurance in the presence of UAF vulnerabilities. In particular, S2MALLOC addresses the two entropy-loss cases (discussed in §3.2 and §3.3) with RIO (§4.2) and RBL (§4.3), respectively, and hence, providing much higher effectiveness on UAF mitigation. In addition, S2MALLOC is designed to monitor the integrity of the heap actively and watch for UAF attempts, including heap spraying practices that aim to prepare the heap data and layout for UAF exploits. S2MALLOC achieves this through a synergy of regular heap canaries(§4.4) and FBC (§4.5). Consequently, S2MALLOC provides a robust security guarantee and cannot be easily evaded.

Table 1 also shows a sheer contrast between entropy-based allocators and allocators specialized in complete UAF-mitigation. Notably, although providing a complete mitigation guarantee toward UAF, allocators in this line [2,41] might impair program efficiency significantly and are less suitable for deployment in time-sensitive use cases. In contrast, as will be presented in §6, S2MALLOC incurs a considerably lower overhead, which is typical for entropy-based allocators, making S2MALLOC practical and deployable on production systems if the end-user can tolerate a marginal chance of protection failure (less than 10% in the default setting of S2MALLOC, discussed in §5).

5 Security Evaluation

5.1 Formal analysis

To mathematically model how S2MALLOC provides defense against UAF, we make the following assumptions for the attacker and target program (which are consistent with our adversary model in §3.1):

① The goal of the attacker is to modify the *victim field*, a sensitive field (e.g., a function pointer or an `is_admin` flag) in a specific type of object, a.k.a., *a victim object*, via memory writes over a dangling pointer (i.e., UAF-writes).
② The attacker can obtain a dangling pointer through a bug in the program at any point of time during execution.
③ The program repetitively allocates and frees the type of objects targeted by the attacker (i.e., victim objects) during its execution. However, we do not assume that each victim object is freed before the next victim is allocated.
④ The attacker can either indirectly monitor or directly control the allocations of victim objects, i.e., the attacker knows when a victim object is allocated, but does not know the address of the allocation.
⑤ Any memory writes through the dangling pointer is conducted after the victim object is allocated.
⑥ If the intended sensitive field of a victim object is overridden, the attack succeeds; otherwise, the program continues, allowing the attacker to repeat the exploitation effort unless detected by S2MALLOC (condition ⑦).
⑦ S2MALLOC checks FBCs on each heap allocation and detects the attack if any FBC is modified.

To simplify the illustration, we assume that the above execution logic is the only code logic that involves heap management. Below, we introduce two simple and yet realistic strategies a reasonable attacker might consider:

S1: Repeat UAF-writes through the same dangling pointer. Suppose an attacker is confident that the memory block pointed to by the dangling pointer is not taken by a long-living irrelevant object. In that case, the attacker may prefer to keep reusing that dangling pointer for future attack attempts and hope that a victim object is allocated in that block.

S2: Repeat UAF-writes with freshly obtained dangling pointers. Suppose an attacker worries that irrelevant objects can be long-lasting (i.e., holding a block that may be allocated to victim objects). In that case, the attacker may prefer to use a fresh dangling pointer per each attempt. However, we note that this is only an attack strategy, and we assume that there is no heap allocation other than victim objects.

For each scenario above, we discuss the rates of whether the attacker adopts the heap spray technique separately, introducing four scenarios in total. We refer readers to our full paper for an extended discussion.

5.2 Illustrate the protection rates

We take the mRuby issue 4001 (Figure 1) as an example and show how its protection rate is computed. The size of object `mrb_io` is 16 bytes, and the

	Attack Strategy 1						Attack Strategy 2					
round	1	5	10	50	100	500	1	5	10	50	100	500
$p_{protection}$	1.4%	4.1%	7.4%	28%	43%	64%	0.8%	12%	37%	95%	95%	95%
p_{attack}	1.2%	2.6%	4.4%	15%	24%	35%	1.2%	2.6%	4.0%	5.5%	5.5%	5.5%

Table 2: Protection and attack success rates of attack rounds in mRuby issue 4001 using the two strategies.

Vulnerability	**Attack pattern**	**[30]**	**[27]**	**[20]**	**S2malloc**	**[38]**	**[3]**
CVE-2015-6831	DP → Write	◐	○	○	●	○	○
CVE-2015-6834	DP → Write	◐	○	○	●	○	○
CVE-2015-6835	DP → Write	◐	○	○	●	○	○
CVE-2015-6835	DP → Write → sleep	○	○	○	●	○	○
CVE-2020-24346	DP → Write	○	○	○	●	○	○
Python-91153	DP → Write	○	○	○	●	■	○
mruby-4001	DP → Write	○	○	○	●	■	■
CVE-2022-22620	DP → Read	○	◑	○	◉	○	○

Table 3: Summary of how different memory allocators defend against eight exploitation techniques on seven vulnerabilities. Vanilla BIBOP allocator and Scudo [22] are vulnerable to all attacks and behave similarly to Guarder [30] (DP: dangling pointer, ○: no defense, ◐: detect at the end of execution, ◑: defense via zero-out, ●: detect via FBC change, ◉: non-deterministic leak (RIO), ■: thwart the exploitation ability).

attacker's goal in each run is to overwrite 4 bytes of it. With the default settings ($r = 256$, $s = 32$ to store a 16-byte object), the attack success rate of each trial is approximately 0.002, and the probability of overwriting FBC in a free block is approximately 0.16. In Table 2, we show how the rates change as the number of attack rounds goes up. The attack is 64% likely to be detected if the attacker adopts the first attack strategy and 95% likely to be detected using the second strategy after running it 500 times.

5.3 Defending against real-world CVEs

In this section, we compare how S2malloc, Guarder, DieHarder, SlimGuard, Cling, and TypeAfterType perform in defending against seven UAF vulnerabilities. We select these vulnerabilities based on the following criteria:

- The vulnerabilities target the Linux platform and can be mitigated in the user space (i.e., not a Linux kernel bug);
- The vulnerabilities can be deterministically triggered (i.e., not racy);
- Public exploits for these vulnerabilities are available and the exploit breaks the program information integrity (i.e., not just causing DoS).

Out of the seven vulnerabilities, six are UAF-write bugs and one (CVE-2022-22620) is a UAF-read only bug. We also found seven exploits against these vulnerabilities (two exploits for CVE-2015-6835 with different attack patterns). All CVEs except Python-91153 and mruby-4001 can cause arbitrary code execution

(ACE) if properly exploited. However, a powerful attack (e.g., ACE by overriding a function pointer) can succeed when the target object is precisely allocated to a freed memory chunk that is still referred to by a dangling pointer[1]. S2MALLOC can mitigate attacks by reducing the chance that a target object is referred to by a dangling pointer. Evaluation results of the eight exploits are in Table 3.

Entropy-based allocators. S2MALLOC can thwart all UAF-write attacks evaluated, Guarder can detect three exploits by recognizing double-free attempts, but DieHarder and SlimGuard fail to thwart these exploits. For the UAF-read attack, S2MALLOC uses RIO to stop the attacker from reusing the memory chunk with accurate object alignment, causing the data read by the dangling pointer to be not sensible. DieHarder zeros out the memory after it is freed, which is effective if the attacker tries to over-read a freed block.

Context-based allocators. While we expect context-based allocators to demonstrate strong and stable protection, some of the exploits, unfortunately, hit on certain blind spots in Cling and TypeAfterType by accident.

In the case of Cling, if both the dangling pointer and the target object (i.e., the object an attacker hope to corrupt) are allocated through the same multi-layer function call sequence, they are considered to fall under the same allocation context, causing the target object to be possibly accessed by the dangling pointer. We will illustrate this limitation through the CVE-2015-6835 case study presented later. In fact, all examined CVEs, except mruby-4001, hit this limitation of Cling. Cling mitigates mruby-4001 by limiting the attacker to target objects of type `mrb_io`, which prevents the attacker from creating a powerful attack primitive.

TypeAfterType can unpack `malloc` wrappers with an arbitrary number of layers until it finds a `sizeof(T)` in the function argument, and an ID `i` is given to each allocation site of `T`. The tuple (`i`,`T`) makes the allocation context, and all memory allocations through this call sequence will be allocated from a memory pool dedicated to this context. However, if the dangling pointer and the target object share the same context in an exploit, UAF can still occur. We will illustrate this limitation through the CVE-2015-6835 case study. TypeAfterType mitigates Python-91153 by limiting the target object to be a reallocated `string`, and mruby-4001 by limiting the target object to be an `mrb_io`.

Case study: CVE-2015-6835. This CVE is a UAF bug in the PHP session deserializer, which reconstructs a session from a serialized string. (see Figure 7 in the extended version for simplified code snippets to illustrate this CVE and its exploit in details). An attacker can exploit this vulnerability to control a dangling pointer to a freed `zval` object. This is possible as the return value (a

[1] An attacker may attack blindly, e.g., overriding a code pointer through the dangling pointer regardless of whether it points to a target object or not. This will have three consequences: 1) corrupting FBC, which may cause the attack to be detected upon future `malloc`s, 2) overwriting a wrong field due to RIO which may cause the program to enter a weird state (e.g., crash), 3) a successful attack. If the program can be recovered from a weird state automatically (e.g., crash resilience), the attacker can retry the same attack and eventually case 1 or case 3 will occur. However, without the probabilistic detection on UAF attempts by S2MALLOC, only case 3 will occur.

`zval` pointer) of `php_var_unserialize` is freed in its caller without noticing that the same pointer might also be stored in a global variable `SESSION_VARS`.

The `zval` type, unfortunately, is a reference-counting wrapper over nearly all other objects involved in the PHP engine (see Figure 4 in the extended version for the type definition of `zval`). Therefore, an attacker might corrupt any `zval` object that may be reallocated to the freed slot. They can simply uses the `echo(..)` function to dump a newly allocated `zval` in the freed memory.

1) Protection from entropy-based allocators. S2MALLOC checks FBC on every `malloc()`. In this exploit, when the attacker tries to use the dangling pointer in `zend_echo_handler`, its `refcount` field is increased, causing the FBC to be modified. This enables S2MALLOC to detect the UAF attempt when the corrupted slot or a nearby slot is about to be reallocated. If the `refcount` change does not corrupt FBC (simulated by disabling the FBC check) and this corrupted block is reallocated, S2MALLOC can still stop the exploit as RIO causes misalignment between the dangling pointer and new object, causing the `type` field to have value `UNKNOWN` and prevents echo printing.

Guarder and DieHarder try to mitigate this attack by random allocation: hoping the new object will not be referred to by a dangling pointer. However, our experiment shows that Guarder fails if the attacker re-runs the attack multiple times or spray the heap with victim objects. SlimGuard fails to provide protections as it always allocates the most recently freed objects to the program. It does not implement the claimed random allocation feature and does not have any other security features that could detect UAF. DieHarder zeros out the memory chunk that stops information leakage of the freed `zval`, but it cannot prevent an attacker to corrupt the newly allocated `zval` over the freed chunk.

2) Protection from context-based allocators. In this exploit, both the dangling pointer and the target object (i.e., the object the attacker wish to dump information via `zend_echo_handler`) are allocated by the the same multi-layer `malloc` wrapper: `php_var_unserialize`→`emalloc`→`malloc`. This is critical to understand why Cling and TypeAfterType fail to mitigate this exploit.

For Cling, this `malloc` wrapper implies that the allocation of many `zval` objects will be sharing the same context (measured by the two innermost return addresses on the call stack). This leaves the dangling pointer plenty of candidate objects to refer to after several rounds of deserialization in PHP. TypeAfterType can inline `malloc` wrappers but the inlining stops at `php_var_unserialize` because it sees the `sizeof(zval)` argument in `emalloc` and hence, will allocate all `zval` objects originating from this `malloc` wrapper from the same pool. Unfortunately, the dangling pointer is also allocated this way, enabling UAF among the dangling pointer to other `zval` objects as well.

Summary. The combination of random allocation and delayed free-list provided by previous entropy-based allocators focus on one-time attacks only, Hence, repeating the same attack is a simple yet effective solution to undermine their protection. Context-based allocators, on the other hand, might fail to detect UAF among objects allocated of the same context. While these results highlight the

effectiveness of S2MALLOC, we note that information leakage through corrupted pointers might diminish this guarantee, which is discussed in §7.

6 Performance Evaluation

To evaluate its performance, we align with previous works [13,2,30,20] that run S2MALLOC on macro, micro, and real-world programs, covering computation-heavy, memory-heavy, real-world simulating, and dedicated memory allocator benchmarks. We also analyze how different parameter values influence it.

Experiment setup. Experiments are performed on both x86 and AARCH servers for macro benchmarks. Others are only on the x86 server. The x86 server is configured with 160-core 2.40GHz Intel CPUs (x86 architecture) with 1TB system memory. We set up the AARCH server on Amazon Web Service (AWS) using the `im4gn.4xlarge` machine with 16 vCPU cores and 64 GB memory. Benchmarks are measured in the Docker environment with Debian 11 on both machines, and they are used exclusively for this evaluation task. We note that benchmarking within a Docker container only introduce negligible overheads [23]. We measure the overheads using the GNU `time` binary and setting the `LD_PRELOAD` environment variable to substitute the system default allocator.

We obtain SlimGuard, Guarder, and DieHarder from their GitHub repository. To provide a fair result, we reduce the allocation entropy bit of Guarder to eight (same as the default value of SlimGuard and S2MALLOC). We also disable DieHarder from zeroing out freed blocks (slightly accelerates DieHarder).

S2MALLOC is measured with the settings of checking two nearby blocks ($d = 2$), 10% random guard page, and taking 1/4 of the block size as random offset entropy ($e = 0.25b$). We zero out blocks smaller than a memory page (4096 bytes) and take the whole block as FBC. For blocks larger than a memory page, we set an 8-byte FBC ($c = 8$) in the corresponding blocks. We set the heap canary length to be one byte ($\iota = 1$) following the design of SlimGuard and Guarder. All reported times and memory usage are normalized using the baseline (glibc) output. We use geometric averages to compute average overheads and report the means of five runs. We note that the standard deviations are minimal.

6.1 Macro benchmarks

PARSEC. We use the PARSEC3 [5] benchmark. We exclude three network tests and one test (x264) that fails to be compiled in the baseline scenario, and only report the results of the remaining 12 benchmarks. Additionally, we exclude "raytrace" from execution for the AARCH server as it cannot be compiled.

SPEC CPU2017. We use SPEC CPU2017 [33] version 1.1.9. We report the results of 12 C/C++-only tests.

Results. On the AARCH machine, we exclude Guarder from analysis, noticing that Guarder relies on AES-NI [17], , which is not available on AARCH machines.

For the SPEC benchmark, on the x86 machine, S2MALLOC introduces 11.5% run-time overhead, smaller than two allocators – SlimGuard and DieHarder-

	Run-time Overhead				Memory Overhead			
	x86		**AARCH**		**x86**		**AARCH**	
	SPEC	PARSEC	SPEC	PARSEC	SPEC	PARSEC	SPEC	PARSEC
S2malloc	**12%**	**2.8%**	**16%**	**1.8%**	**37%**	**27%**	**38%**	**28%**
SlimGuard	17%	4.4%	7.7%	2.6%	57%	23%	57%	24%
DieHarder	31%	2.1%	-	2.5%	59%	21%	-	21%
Guarder	3.5%	2.4%	-	-	56%	58%	-	-

Table 4: Normalized overheads on SPEC and PARSEC benchmarks.

	Run-time Overhead	Memory Overhead
S2MALLOC	189%	343%
SlimGuard	298%	250%
DieHarder	229%	92%
Guarder	56%	980%

Table 5: Normalized run-time and memory overheads of running mimalloc-benchmark.

	Δ Memory Overhead	Δ Run-time Overhead
0 Nearby	-0.05%	-0.42%
4 Nearby	-0.13%	+0.4%
4B Random N	-12.35%	-0.33%
12B Random N	+68.23%	+0.73%
50% Entropy	+9.57%	+0.49%

Table 6: Normalized overhead changes compared with the default settings.

and the least memory overhead at 37.4%. On the AARCH machine, S2MALLOC introduces a similar 15.5% run-time overhead, larger than the SlimGuard since SlimGuard fails to run tests with frequent heap memory management operations. Running the PARSEC benchmark gives similar results, with smaller memory and run-time overheads.

6.2 Micro benchmarks

To further understand the overheads, we investigate its performance using mimalloc-bench [7], composed of real-world and calibrated programs that frequently allocate and free heap memory. Results are shown in Table 5.

All secure memory allocators incur larger overheads compared to running real-world (see §6.3) or general-purpose benchmarks as mimalloc-bench tests operate the heap memory in a biased frequent way, and some tests (e.g., "leanN" generates the largest run-time overhead with S2MALLOC) count the CPU ticks instead of seconds of finishing each call.

We take one test, `glibc-simple`, originally from glibc-bench [1], to further investigate the two most common heap object management functions – `malloc()` and `free()`. The test times the execution of allocating and freeing a large number of blocks of a given size. We modify the benchmark and monitor the execution time of calling `malloc()` and `free()` separately. To investigate the time consumption of different sizes, we vary the block size `S` to be 16B, 128B, and 1KB and change the number of allocated blocks `N` correspondingly so that the total allocation size

Fig. 3: Execution times of glibc-simple.

	Nginx			Lighttpd			Redis	
	Throughput	Memory	p50	Throughput	Memory	p50	Throughput	Memory
S2MALLOC	9705	7867	56	11050	9425	44	218460	44689
Guarder	9496	13724	52	11146	11088	44	221245	46545
SlimGuard	6159	4935	81	11153	6428	44	219986	47398
DieHarder	8769	7396	57	11128	13626	44	221734	52419
Glibc	9564	3400	52	10742	5069	44	218156	50909

Table 7: Real-world programs' memory usage (KB) and throughput (requests/second).

(`N * S`) is always 1000 MB following the benchmark design of SlimGuard [20]. Results are presented in Figure 3, and is the average of 100 runs.

Generally, S2MALLOC takes more time to execute `malloc()` than other compared allocators and takes less time to execute `free()` than DieHarder but longer time than Guarder and SlimGuard. However, a significant overhead comes with our cryptographically secure canary implementation, which all allocators should adopt. Although using hardware acceleration, the canary value is computed in each `malloc()` and `free()` call, introducing a nonnegligible computation tax. After disabling this feature and using a fixed value as the canary, following Guarder's implementation, the execution time of both calls is close to other allocators, and the increased overhead is expected as S2MALLOC introduces extra security guarantees. For example, in 16B `malloc` and `free` calls, S2MALLOC is 31% and 26% slower than SlimGuard respectively.

6.3 Performance on real-world programs

To evaluate the performance of S2MALLOC in real-world environments, we run Nginx (1.18.0), Lighttpd (1.4.71), and Redis (7.2.1) on the x86 machine. We use ApacheBench (ab) [35] 2.3 to test the throughput and delays using the Nginx default root page of 613 bytes as the requested page to minimize the I/O times. We send 500 concurrent requests, which reaches the maximum capacity of our server. On Redis, we use the same settings as mimalloc-bench [7]. Results are in Table 7. Applying S2MALLOC on these programs results in minimal throughput influence (even better throughput on Nginx and Redis). Running S2MALLOC delays the request response time on Nginx but not Lighttpd. Applying all compared allocators increases memory consumption.

6.4 Influence with different parameters

In addition to the default settings, we also measure how different parameters, namely, nearby checking range, random offset entropy, and RIO entropy, influence

the run-time and memory overheads. We take 0 and 4 blocks for the nearby checking range compared to the default setting: 2. We take 4 bytes and 12 bytes for the random allocation entropy compared to the default 8 bytes. For random offset entropy, we reserve 50% of the block size for RIO compared to the default 25%. Table 6 shows how different parameters influence the overheads.

Changing the nearby checking range does not introduce observable differences in the memory overhead. The introduced delta is likely due to server fluctuations. Using a larger nearby checking range introduces a greater run-time overhead, as S2MALLOC needs to compute and check more canary values. A larger random allocation entropy or RIO introduces larger memory and run-time overheads.

7 Discussion

Limited UAF-read protection. An attacker can always read through a dangling pointer (unless the page is unmapped) and none of the entropy-based allocators, including S2MALLOC, can prevent this. S2MALLOC mitigates UAF-read exploits based on the assumption that the attacker cannot distinguish memory content stored in target data fields from the content stored in other data fields. However, in the scenario where the target object contains a field that the attacker could craft, e.g., a marker value such as `0xdeadbeef`, the target field can be located by probing the marker via UAF read(s). Mitigation of UAF-read exploits can be achieved completely with allocators in categories A, B, C, and partially with allocators in category D (see §2.2).

Increasing canary checking frequency. As the core of detecting invalid UAF writes, S2MALLOC checks the nearby canaries when a block is freed. While this design could virtually cover all memory slots for a program with frequent heap operations, UAF attack attempts cannot be detected if there is no `malloc` call that triggers S2MALLOC to check FBC. Executing FBC checks in other memory operations (e.g., `free`) is one option to increase checking frequency. Another option is to integrate FBC checks into program logic. Both options require a careful balance between checking frequency (i.e., security) and overhead.

8 Conclusion

While statistically effective against all common heap exploitation techniques, state-of-the-art entropy-based heap allocators are not tailored to *actively detect* unsuccessful exploitation attempts. As a result, to beat a randomization-based moving-target scheme, an attacker can simply launch the same attack repeatedly, potentially with heap spraying, until success.

In this paper, we present S2MALLOC to fill the gap of exploitation attempt detection without compromising security and performance. In particular, we introduce three novel primitives to the design space of heap allocators: random in-block offset (RIO), free block canaries (FBC), and random bag layout (RBL). Combined with conventional BIBOP-style random allocation and heap canaries,

S2MALLOC can maintain at least the same level of protection against other heap exploitations (e.g., overflows) and yet still achieves 69% and 96% protection rate in two attack scenarios, respectively, against UAF exploitation attempts targeting a 64 bytes object, while only incurs marginal performance overhead, making S2MALLOC practical to even production systems.

References

1. Standalone Glibc-Benchtests. https://github.com/xCuri0/glibc-benchtests
2. Ainsworth, S., Jones, T.M.: MarkUs: Drop-in Use-After-Free Prevention for Low-level Languages. In: Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland). San Francisco, CA (2020)
3. Akritidis, P.: Cling: A memory allocator to mitigate dangling pointers. In: 19th USENIX Security Symposium. pp. 177–192 (2010)
4. ARM Developer: Neon. https://developer.arm.com/Architectures/Neon
5. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. Tech. rep. (2008)
6. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th Conference on USENIX Security Symposium (1998)
7. daanx: mimalloc-bench — Suite for benchmarking malloc implementations.
8. Dang, T.H.Y., Maniatis, P., Wagner, D.: Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In: Proceedings of the 26th USENIX Conference on Security Symposium (2017)
9. Dhurjati, D., Kowshik, S., Adve, V., Lattner, C.: Memory safety without runtime checks or garbage collection. In: Proceedings of the 2003 ACM SIGPLAN conference on language, compiler, and tool for embedded systems (2003)
10. Erdős, M., Ainsworth, S., Jones, T.M.: MineSweeper: A "Clean Sweep" for Drop-in Use-after-Free Prevention. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems
11. Farkhani, R.M., Ahmadi, M., Lu, L.: PTAuth: Temporal Memory Safety via Robust Points-to Authentication. CoRR (2020)
12. Filardo, N., Gutstein, B., Woodruff, J., Ainsworth, S., Paul-Trifu, L., Davis, B., Xia, H., Napierala, E., Richardson, A., Baldwin, J., Chisnall, D., Clark, J., Gudka, K., Joannou, A., Markettos, A., Massinghi, A., Norton, R., Roe, M., Sewell, P., Son, S., Jones, T., Moore, S., Neumann, P., Watson, R.: Cornucopia: Temporal safety for cheri heaps. In: Proceedings of the 41st IEEE Symposium on Security and Privacy. IEEE Computer Society (2020)
13. Gorter, F., Koning, K., Bos, H., Giuffrida, C.: DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In: Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)
14. GrapheneOS: GrapheneOS: the private and secure OS. https://grapheneos.org/
15. GrapheneOS: Hardened_malloc
16. Hanson, D.R.: A Portable Storage Management System for The ICON Programming Language. Software: Practice and Experience **10**(6), 489–500 (1980)
17. Intel Corporation: Intel Advanced Encryption Standard Instructions (AES-NI)
18. Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., Lee, W.: Preventing use-after-free with dangling pointers nullification. In: Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS) (2015)

19. Liljestrand, H., Gauhar, Z., Nyman, T., Ekberg, J.E., Asokan, N.: Protecting the stack with paced canaries. In: Proceedings of the 4th Workshop on System Software for Trusted Execution (2019). https://doi.org/10.1145/3342559.3365336
20. Liu, B., Olivier, P., Ravindran, B.: SlimGuard: A Secure and Memory-Efficient Heap Allocator. In: Proceedings of the 20th International Middleware Conference
21. LLVM Project: Clang: a C language family frontend for LLVM
22. LLVM Project: Scudo Hardened Allocator
23. Martin, J.P., Kandasamy, A., Chandrasekaran, K.: Exploring the support for high performance applications in the container runtime environment. Human-centric Computing and Information Sciences **8**(1) (2018)
24. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: Ccfi: Cryptographically enforced control flow integrity. CCS '15, New York, NY, USA (2015)
25. National Vulnerability Database: 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html
26. Nikiforakis, N., Piessens, F., Joosen, W.: Heapsentry: Kernel-assisted protection against heap overflows. In: Detection of Intrusions and Malware & Vulnerability Assessment (2013)
27. Novark, G., Berger, E.D.: DieHarder: Securing The Heap. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS) (2010)
28. Qiang, W., Li, W., Jin, H., Surbiryala, J.: Mpchecker: Use-after-free vulnerabilities protection based on multi-level pointers. IEEE Access **7**, 45961–45977 (2019)
29. Shellphish: shellphish/how2heap: A repository for learning various heap exploitation techniques. https://github.com/shellphish/how2heap
30. Silvestro, S., Liu, H., Liu, T., Lin, Z., Liu, T.: Guarder: A Tunable Secure Allocator. In: Proceedings of the 27th USENIX Security Symposium (Security) (2018)
31. Singh, S., Awasthi, M.: Memory centric characterization and analysis of spec cpu2017 suite. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. ICPE '19, ACM (2019)
32. Sotirov, A.: Heap Feng Shui in Javascript. Black Hat Europe (2007)
33. Standard Performance Evaluation Corporation: SPEC CPU® 2017
34. Teuchert, D. and Aschermann, C. and Frassetto, T. and Abera T.: Use after free in File#initilialize_copy. https://github.com/mruby/mruby/issues/4001 (2018)
35. The Apache Software Foundation: Apache HTTP Server Documentation: ab - Apache HTTP Server Benchmarking Tool (2023)
36. The Chromium Projects: Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/
37. The GNU Project: Aligned Memory Blocks (The GNU C Library), https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html
38. Van Der Kouwe, E., Kroes, T., Ouwehand, C., Bos, H., Giuffrida, C.: Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In: Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC) (2018)
39. Van Der Kouwe, E., Nigade, V., Giuffrida, C.: Dangsan: Scalable use-after-free detection. In: Proceedings of the Twelfth European Conference on Computer Systems. p. 405–419. Association for Computing Machinery (2017)
40. Wang, R., Xu, M., Asokan, N.: Semalloc: Semantics-informed memory allocator (2024)
41. Wickman, B., Hu, H., Yun, I., Jang, D., Lim, J., Kashyap, S., Kim, T.: Preventing Use-After-Free Attacks with Fast Forward Allocation. In: Proceedings of the 30th USENIX Security Symposium (Security)
42. Younan, Y.: Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In: Network and Distributed System Security Symposium (2015)